eCos Reference Manual

Documentation licensing terms

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).
Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.
Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Trademarks

Altera® and Excalibur™ are trademarks of Altera Corporation.
AMDi® is a registered trademark of Advanced Micro Devices, Inc.
ARM®, StrongARM®, Thumb®, ARM7™, ARM9™ is a registered trademark of Advanced RISC Machines, Ltd.
Cirrus Logic® and Maverick™ are registered trademarks of Cirrus Logic, Inc.
Cogent™ is a trademark of Cogent Computer Systems, Inc.
Compaq® is a registered trademark of the Compaq Computer Corporation.
Fujitsu® is a registered trademark of Fujitsu Limited.
IBM®, and PowerPC™ are trademarks of International Business Machines Corporation.
IDT® is a registered trademark of Integrated Device Technology Inc.
Intel®, i386™, Pentium® , StrataFlash® and XScale™ are trademarks of Intel Corporation.
Intrinsyc® and Cerf™ are trademarks of Intrinsyc Software, Inc.
Linux® is a registered trademark of Linus Torvalds.
Matsushita™ and Panasonic® are trademarks of the Matsushita Electric Industrial Corporation.
Microsoft®, Windows®, Windows NT® and Windows XP® are registered trademarks of Microsoft Corporation, Inc.
MIPS®, MIPS32™ MIPS64™, 4K™, 5K™ Atlas™ and Malta™ are trademarks of MIPS Technologies, Inc.
Motorola®, ColdFire® is a trademark of Motorola, Inc.
NEC® V800™, V850™, V850/SA1™, V850/SB1™, VR4300™, and VRC4375™ are trademarks of NEC Corporation.
PMC-Sierra® RM7000™ and Ocelot™ are trademarks of PMC-Sierra Incorporated.
Red Hat, eCos™, RedBoot™, GNUPro®, and Insight™ are trademarks of Red Hat, Inc.
Samsung® and CalmRISC™ are trademarks of Samsung, Inc.
Sharp® is a registered trademark of Sharp Electronics Corp.
SPARC® is a registered trademark of SPARC International, Inc., and is used under license by Sun Microsystems, Inc.
Sun Microsystems® and Solaris® are registered trademarks of Sun Microsystems, Inc.
SuperH™ and Renesas™ are trademarks owned by Renesas Technology Corp.
Texas Instruments®, OMAP™ and Innovator™ are trademarks of Texas Instruments Incorporated.
Toshiba® is a registered trademark of the Toshiba Corporation.
UNIX® is a registered trademark of The Open Group.
All other brand and product names, trademarks, and copyrights are the property of their respective owners.

Warranty

eCos and RedBoot are open source software, covered by a modified version of the GNU General Public Licence (http://www.gnu.org/copyleft/gpl.html), and you are welcome to change it and/or distribute copies of it under certain conditions. See http://ecos.sourceforge.org/license-overview.html for more information about the license.
eCos and RedBoot software have NO WARRANTY.
Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.
# Table of Contents

I. The eCos Kernel ........................................................................................................... xxiii  
   Kernel Overview ........................................................................................................ 25  
   SMP Support ............................................................................................................. 33  
   Thread creation .......................................................................................................... 37  
   Thread information ................................................................................................... 41  
   Thread control .......................................................................................................... 45  
   Thread termination .................................................................................................... 47  
   Thread priorities ....................................................................................................... 49  
   Per-thread data ......................................................................................................... 51  
   Thread destructors .................................................................................................... 53  
   Exception handling ................................................................................................. 55  
   Counters ................................................................................................................... 57  
   Clocks ....................................................................................................................... 59  
   Alarms ....................................................................................................................... 61  
   Mutexes ..................................................................................................................... 63  
   Condition Variables ............................................................................................... 69  
   Semaphores .............................................................................................................. 73  
   Mail boxes ................................................................................................................ 75  
   Event Flags .............................................................................................................. 77  
   Spinlocks .................................................................................................................. 81  
   Scheduler Control .................................................................................................... 83  
   Interrupt Handling .................................................................................................. 85  
   Kernel Real-time Characterization .......................................................................... 91  

II. The eCos Hardware Abstraction Layer (HAL) ............................................................ ci  
   1. Introduction .......................................................................................................... 1  
   2. Architecture, Variant and Platform .................................................................... 3  
   3. General principles .............................................................................................. 5  
   4. HAL Interfaces ................................................................................................... 7  
      Base Definitions .................................................................................................. 7  
      Byte order ........................................................................................................... 7  
      Label Translation .............................................................................................. 7  
      Base types .......................................................................................................... 7  
      Atomic types ..................................................................................................... 8  
      Architecture Characterization .......................................................................... 8  
      Register Save Format ......................................................................................... 8  
      Thread Context Initialization ............................................................................ 8  
      Thread Context Switching .................................................................................. 9  
      Bit indexing ....................................................................................................... 9  
      Idle thread activity ......................................................................................... 10  
      Reorder barrier ............................................................................................... 10  
      Breakpoint support ......................................................................................... 10  
      GDB support ..................................................................................................... 10  
      Setjmp and longjmp support ............................................................................ 11  
      Stack Sizes ........................................................................................................ 11  
      Address Translation ......................................................................................... 11  
      Global Pointer ................................................................................................. 12  
   
   Interrupt Handling .................................................................................................. 12  
      Vector numbers ................................................................................................. 12  
      Interrupt state control ..................................................................................... 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation details</td>
<td>44</td>
</tr>
<tr>
<td>New Platform Ports</td>
<td>44</td>
</tr>
<tr>
<td>New architecture ports</td>
<td>44</td>
</tr>
<tr>
<td>IO channels</td>
<td>44</td>
</tr>
<tr>
<td>Available Procedures</td>
<td>45</td>
</tr>
<tr>
<td>Usage</td>
<td>46</td>
</tr>
<tr>
<td>Compatibility</td>
<td>46</td>
</tr>
<tr>
<td>Implementation Details</td>
<td>46</td>
</tr>
<tr>
<td>New Platform Ports</td>
<td>47</td>
</tr>
<tr>
<td>HAL Coding Conventions</td>
<td>47</td>
</tr>
<tr>
<td>Implementation issues</td>
<td>48</td>
</tr>
<tr>
<td>Source code details</td>
<td>48</td>
</tr>
<tr>
<td>Nested Headers</td>
<td>49</td>
</tr>
<tr>
<td>Platform HAL Porting</td>
<td>50</td>
</tr>
<tr>
<td>HAL Platform Porting Process</td>
<td>50</td>
</tr>
<tr>
<td>Brief overview</td>
<td>50</td>
</tr>
<tr>
<td>Step-by-step</td>
<td>51</td>
</tr>
<tr>
<td>Minimal requirements</td>
<td>51</td>
</tr>
<tr>
<td>Adding features</td>
<td>52</td>
</tr>
<tr>
<td>Hints</td>
<td>53</td>
</tr>
<tr>
<td>HAL Platform CDL</td>
<td>54</td>
</tr>
<tr>
<td>eCos Database</td>
<td>54</td>
</tr>
<tr>
<td>CDL File Layout</td>
<td>55</td>
</tr>
<tr>
<td>Startup Type</td>
<td>56</td>
</tr>
<tr>
<td>Build options</td>
<td>56</td>
</tr>
<tr>
<td>Common Target Options</td>
<td>57</td>
</tr>
<tr>
<td>Platform Memory Layout</td>
<td>60</td>
</tr>
<tr>
<td>Layout Files</td>
<td>60</td>
</tr>
<tr>
<td>Reserved Regions</td>
<td>61</td>
</tr>
<tr>
<td>Platform Serial Device Support</td>
<td>61</td>
</tr>
<tr>
<td>Variant HAL Porting</td>
<td>62</td>
</tr>
<tr>
<td>HAL Variant Porting Process</td>
<td>63</td>
</tr>
<tr>
<td>HAL Variant CDL</td>
<td>63</td>
</tr>
<tr>
<td>Cache Support</td>
<td>65</td>
</tr>
<tr>
<td>Architecture HAL Porting</td>
<td>65</td>
</tr>
<tr>
<td>HAL Architecture Porting Process</td>
<td>65</td>
</tr>
<tr>
<td>CDL Requirements</td>
<td>70</td>
</tr>
<tr>
<td>7. Future developments</td>
<td>75</td>
</tr>
<tr>
<td>III. The ISO Standard C and Math Libraries</td>
<td>77</td>
</tr>
<tr>
<td>8. C and math library overview</td>
<td>79</td>
</tr>
<tr>
<td>Included non-ISO functions</td>
<td>79</td>
</tr>
<tr>
<td>Math library compatibility modes</td>
<td>80</td>
</tr>
<tr>
<td>matherr()</td>
<td>80</td>
</tr>
<tr>
<td>Thread-safety and re-entrancy</td>
<td>82</td>
</tr>
<tr>
<td>Some implementation details</td>
<td>82</td>
</tr>
<tr>
<td>Thread safety</td>
<td>83</td>
</tr>
<tr>
<td>C library startup</td>
<td>84</td>
</tr>
</tbody>
</table>
IV. I/O Package (Device Drivers) ........................................................................................................... 87

9. Introduction ........................................................................................................................................... 89
10. User API ............................................................................................................................................... 91
11. Serial driver details ............................................................................................................................... 93
    Raw Serial Driver ................................................................................................................................. 93
        Runtime Configuration ....................................................................................................................... 93
    API Details ............................................................................................................................................. 94
        cyg_io_write ................................................................................................................................. 95
        cyg_io_read ................................................................................................................................. 95
        cyg_io_get_config ...................................................................................................................... 95
        cyg_io_set_config ...................................................................................................................... 97
    TTY driver ........................................................................................................................................... 98
        Runtime configuration ................................................................................................................ 99
        API details ................................................................................................................................ 99
12. How to Write a Driver .......................................................................................................................... 101
    How to Write a Serial Hardware Interface Driver ........................................................................... 102
        DevTab Entry .............................................................................................................................. 102
        Serial Channel Structure ........................................................................................................... 103
        Serial Functions Structure .......................................................................................................... 104
        Callbacks ................................................................................................................................... 105
        Serial testing with ser_filter .................................................................................................... 106
            Rationale ................................................................................................................................. 106
            The Protocol ............................................................................................................................. 106
            The Serial Tests ....................................................................................................................... 107
            Serial Filter Usage .................................................................................................................. 108
            A Note on Failures ................................................................................................................... 109
            Debugging ............................................................................................................................... 109
13. Device Driver Interface to the Kernel ................................................................................................... 111
    Interrupt Model ................................................................................................................................. 111
    Synchronization ................................................................................................................................. 111
    SMP Support ...................................................................................................................................... 112
    Device Driver Models ....................................................................................................................... 112
    Synchronization Levels .................................................................................................................... 113
    The API ............................................................................................................................................... 114
        cyg_drv_isr_lock .......................................................................................................................... 114
        cyg_drv_isr_unlock ....................................................................................................................... 114
        cyg_drv_spinlock_init .................................................................................................................... 115
        cyg_drv_spinlock_destroy .......................................................................................................... 115
        cyg_drv_spinlock_spin ............................................................................................................... 116
        cyg_drv_spinlock_clear ............................................................................................................... 116
        cyg_drv_spinlock_try .................................................................................................................... 116
        cyg_drv_spinlock_test ................................................................................................................... 117
        cyg_drv_spinlock_spin_intsave ................................................................................................. 117
        cyg_drv_spinlock_clear_intsave ............................................................................................... 118
        cyg_drv_dsr_lock ........................................................................................................................ 118
        cyg_drv_dsr_unlock ...................................................................................................................... 119
        cyg_drv_mutex_init ....................................................................................................................... 119
        cyg_drv_mutex_destroy ............................................................................................................... 120
        cyg_drv_mutex_lock ..................................................................................................................... 120
        cyg_drv_mutex_trylock ................................................................................................................ 120
        cyg_drv_mutex_unlock .................................................................................................................. 121
VII. FLASH Library ............................................................................................................ 171

26. The eCos FLASH Library ................................................................................................. 173
   Notes on using the FLASH library ...................................................................................... 173
   Danger, Will Robinson! Danger! ......................................................................................... 173

27. The Version 2 eCos FLASH API ...................................................................................... 175
   FLASH user API .................................................................................................................. 175
      Initializing the FLASH library ......................................................................................... 175
      Retrieving information about FLASH devices .............................................................. 175
      Reading from FLASH ...................................................................................................... 175
      Erasing areas of FLASH ................................................................................................. 176
      Programming the FLASH ............................................................................................... 176
      Locking and unlocking blocks ....................................................................................... 176
      Locking FLASH mutexes ............................................................................................... 176
      Configuring diagnostic output ....................................................................................... 177
      Return values and errors ............................................................................................... 177

   FLASH device API............................................................................................................. 177
      The FLASH device Structure .......................................................................................... 178

28. The legacy Version 1 eCos FLASH API ........................................................................... 179
   FLASH user API ................................................................................................................ 179
      Initializing the FLASH library ........................................................................................ 179
      Retrieving information about the FLASH ..................................................................... 179
      Reading from FLASH .................................................................................................... 180
      Erasing areas of FLASH ............................................................................................... 180
      Programming the FLASH ............................................................................................. 180
      Locking and unlocking blocks ..................................................................................... 180
      Return values and errors .............................................................................................. 180
      Notes on using the FLASH library ................................................................................ 181

   FLASH device API............................................................................................................. 181
      The flash_info structure ............................................................................................... 181
      Initializing the device driver ........................................................................................ 182
      Querying the FLASH ..................................................................................................... 182
      Erasing a block of FLASH ............................................................................................ 182
      Programming a region of FLASH ................................................................................ 182
      Reading a region from FLASH ..................................................................................... 182
      Locking and unlocking FLASH blocks ........................................................................ 183
      Mapping FLASH error codes to FLASH IO error codes ............................................ 183
      Determining if code is in FLASH .................................................................................. 183
      Implementation Notes .................................................................................................. 183

29. FLASH I/O devices ........................................................................................................ 185
   Overview and CDL Configuration .................................................................................... 185
   Using FLASH I/O devices ............................................................................................... 186

VIII. SPI Support .................................................................................................................. 189
   Overview .......................................................................................................................... 191
   SPI Interface .................................................................................................................... 195
   Porting to New Hardware ............................................................................................... 199
<table>
<thead>
<tr>
<th>IX. I2C Support</th>
<th>201</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>203</td>
</tr>
<tr>
<td>I2C Interface</td>
<td>205</td>
</tr>
<tr>
<td>Porting to New Hardware</td>
<td>209</td>
</tr>
<tr>
<td>X. CAN Support</td>
<td>215</td>
</tr>
<tr>
<td>30. Overview</td>
<td>217</td>
</tr>
<tr>
<td>Description</td>
<td>217</td>
</tr>
<tr>
<td>eCos Support for CAN</td>
<td>217</td>
</tr>
<tr>
<td>31. User API</td>
<td>219</td>
</tr>
<tr>
<td>32. CAN driver details</td>
<td>221</td>
</tr>
<tr>
<td>Description</td>
<td>221</td>
</tr>
<tr>
<td>API Details</td>
<td>221</td>
</tr>
<tr>
<td>cyg_io_write</td>
<td>221</td>
</tr>
<tr>
<td>CAN Messages</td>
<td>221</td>
</tr>
<tr>
<td>cyg_io_read</td>
<td>222</td>
</tr>
<tr>
<td>CAN Events</td>
<td>223</td>
</tr>
<tr>
<td>cyg_io_get_config</td>
<td>224</td>
</tr>
<tr>
<td>cyg_io_set_config</td>
<td>225</td>
</tr>
<tr>
<td>Runtime Configuration</td>
<td>225</td>
</tr>
<tr>
<td>Device configuration</td>
<td>225</td>
</tr>
<tr>
<td>Timeout configuration</td>
<td>226</td>
</tr>
<tr>
<td>Reading buffer configuration</td>
<td>226</td>
</tr>
<tr>
<td>Reading hardware description information</td>
<td>227</td>
</tr>
<tr>
<td>Reading hardware message buffer configuration</td>
<td>228</td>
</tr>
<tr>
<td>Reading state of CAN hardware</td>
<td>228</td>
</tr>
<tr>
<td>Changing mode of CAN hardware</td>
<td>228</td>
</tr>
<tr>
<td>Flush or drain buffers</td>
<td>229</td>
</tr>
<tr>
<td>Configuring blocking/non-blocking calls</td>
<td>229</td>
</tr>
<tr>
<td>Message buffer management</td>
<td>230</td>
</tr>
<tr>
<td>Remote frame response buffer configuration</td>
<td>231</td>
</tr>
<tr>
<td>Message filter configuration</td>
<td>232</td>
</tr>
<tr>
<td>Message filter deactivation</td>
<td>234</td>
</tr>
<tr>
<td>Configuring a callback on events</td>
<td>234</td>
</tr>
<tr>
<td>33. Configuration</td>
<td>237</td>
</tr>
<tr>
<td>34. Writing a CAN hardware device driver</td>
<td>239</td>
</tr>
<tr>
<td>How to Write a CAN Hardware Interface Driver</td>
<td>239</td>
</tr>
<tr>
<td>DevTab Entry</td>
<td>239</td>
</tr>
<tr>
<td>CAN Channel Structure</td>
<td>240</td>
</tr>
<tr>
<td>CAN Lowlevel Functions Structure</td>
<td>241</td>
</tr>
<tr>
<td>Callbacks</td>
<td>242</td>
</tr>
<tr>
<td>XI. ADC Support</td>
<td>243</td>
</tr>
<tr>
<td>Overview</td>
<td>245</td>
</tr>
<tr>
<td>ADC Device Drivers</td>
<td>249</td>
</tr>
<tr>
<td>XII. Framebuffer Support</td>
<td>253</td>
</tr>
<tr>
<td>Overview</td>
<td>255</td>
</tr>
<tr>
<td>Framebuffer Parameters</td>
<td>259</td>
</tr>
<tr>
<td>Framebuffer Control Operations</td>
<td>263</td>
</tr>
<tr>
<td>Framebuffer Colours</td>
<td>269</td>
</tr>
<tr>
<td>Framebuffer Drawing Primitives</td>
<td>275</td>
</tr>
<tr>
<td>Framebuffer Pixel Manipulation</td>
<td>281</td>
</tr>
</tbody>
</table>
Notes ........................................................................................................................................ 306
Thread Management [POSIX Section 16] ..................................................................................... 306
  Functions Implemented .............................................................................................................. 306
  Functions Omitted ..................................................................................................................... 306
  Notes ....................................................................................................................................... 307
Thread-Specific Data [POSIX Section 17] .................................................................................... 307
  Functions Implemented .............................................................................................................. 307
  Functions Omitted ..................................................................................................................... 307
  Notes ....................................................................................................................................... 307
Thread Cancellation [POSIX Section 18] ..................................................................................... 308
  Functions Implemented .............................................................................................................. 308
  Functions Omitted ..................................................................................................................... 308
  Notes ....................................................................................................................................... 308
Non-POSIX Functions .................................................................................................................. 308
  General I/O Functions .............................................................................................................. 308
  Socket Functions ....................................................................................................................... 308
  Notes ....................................................................................................................................... 309
References and Bibliography ........................................................................................................ 311
XIV. μITRON .................................................................................................................................. 311
  36. μITRON API .......................................................................................................................... 313
     Introduction to μITRON .......................................................................................................... 313
     μITRON and eCos ................................................................................................................... 313
     Task Management Functions ................................................................................................ 314
       Error checking ....................................................................................................................... 315
     Task-Dependent Synchronization Functions ........................................................................ 315
       Error checking ....................................................................................................................... 316
     Synchronization and Communication Functions ..................................................................... 316
       Error checking ....................................................................................................................... 318
     Extended Synchronization and Communication Functions .................................................. 319
     Interrupt management functions ........................................................................................... 319
       Error checking ....................................................................................................................... 320
     Memory pool Management Functions ..................................................................................... 320
       Error checking ....................................................................................................................... 321
     Time Management Functions ................................................................................................ 321
       Error checking ....................................................................................................................... 322
     System Management Functions .............................................................................................. 323
       Error checking ....................................................................................................................... 324
     Network Support Functions ................................................................................................... 324
     μITRON Configuration FAQ ................................................................................................... 324
XV. TCP/IP Stack Support for eCos ............................................................................................. 329
  37. Ethernet Driver Design ......................................................................................................... 331
  38. Sample Code ........................................................................................................................ 333
  39. Configuring IP Addresses ....................................................................................................... 335
  40. Tests and Demonstrations ...................................................................................................... 337
    Loopback tests ......................................................................................................................... 337
    Building the Network Tests .................................................................................................... 337
    Standalone Tests ...................................................................................................................... 337
    Performance Test ..................................................................................................................... 338
    Interactive Tests ...................................................................................................................... 339
    Maintenance Tools .................................................................................................................. 340
  41. Support Features .................................................................................................................. 341
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFTP</td>
<td>341</td>
</tr>
<tr>
<td>DHCP</td>
<td>343</td>
</tr>
<tr>
<td>42. TCP/IP Library Reference</td>
<td>345</td>
</tr>
<tr>
<td>getdomainname</td>
<td>345</td>
</tr>
<tr>
<td>gethostname</td>
<td>346</td>
</tr>
<tr>
<td>byteorder</td>
<td>347</td>
</tr>
<tr>
<td>ethers</td>
<td>348</td>
</tr>
<tr>
<td>getaddrinfo</td>
<td>350</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>355</td>
</tr>
<tr>
<td>getifaddrs</td>
<td>357</td>
</tr>
<tr>
<td>getnameinfo</td>
<td>359</td>
</tr>
<tr>
<td>getnetent</td>
<td>362</td>
</tr>
<tr>
<td>getprotoent</td>
<td>363</td>
</tr>
<tr>
<td>getrsetbyname</td>
<td>365</td>
</tr>
<tr>
<td>getservent</td>
<td>366</td>
</tr>
<tr>
<td>if_nametoindex</td>
<td>368</td>
</tr>
<tr>
<td>inet</td>
<td>369</td>
</tr>
<tr>
<td>inet6_option_space</td>
<td>372</td>
</tr>
<tr>
<td>inet6_rhdr_space</td>
<td>376</td>
</tr>
<tr>
<td>inet_net</td>
<td>379</td>
</tr>
<tr>
<td>ipx</td>
<td>380</td>
</tr>
<tr>
<td>iso_addr</td>
<td>381</td>
</tr>
<tr>
<td>link_addr</td>
<td>382</td>
</tr>
<tr>
<td>net_addrcmp</td>
<td>383</td>
</tr>
<tr>
<td>ns</td>
<td>384</td>
</tr>
<tr>
<td>resolver</td>
<td>385</td>
</tr>
<tr>
<td>accept</td>
<td>388</td>
</tr>
<tr>
<td>bind</td>
<td>389</td>
</tr>
<tr>
<td>connect</td>
<td>391</td>
</tr>
<tr>
<td>getpeername</td>
<td>392</td>
</tr>
<tr>
<td>getsockname</td>
<td>394</td>
</tr>
<tr>
<td>getsockopt</td>
<td>395</td>
</tr>
<tr>
<td>ioctl</td>
<td>398</td>
</tr>
<tr>
<td>poll</td>
<td>399</td>
</tr>
<tr>
<td>select</td>
<td>401</td>
</tr>
<tr>
<td>send</td>
<td>403</td>
</tr>
<tr>
<td>shutdown</td>
<td>406</td>
</tr>
<tr>
<td>socket</td>
<td>406</td>
</tr>
</tbody>
</table>

**XVI. FreeBSD TCP/IP Stack port for eCos**

43. Networking Stack Features
44. FreeBSD TCP/IP stack port
   44.1. Targets
   44.2. Building the Network Stack
45. APIs
   45.1. Standard networking
   45.2. Enhanced Select()
XVIII. DNS for eCos and RedBoot ................................................................. 439

49. DNS .......................................................................................... 441
   DNS API................................................................................. 441
   DNS Client Testing............................................................... 442

XIX. IPSEC for eCos ............................................................................. 445

50. Installation and Configuration.......................................................... 447
51. libipsec Reference ........................................................................ 449
      ipsec_set_policy .................................................................. 449
      ipsec_strerror .................................................................... 451

XX. eCos PPP User Guide ........................................................................ 453

52. Features .................................................................................... 455
53. Using PPP .................................................................................. 457
54. PPP Interface ............................................................................... 461
      cyg_ppp_options_init() ..................................................... 461
      cyg_ppp_up() .................................................................... 465
      cyg_ppp_down() ............................................................... 467
      cyg_ppp_wait_up() .......................................................... 469
      cyg_ppp_wait_down() ...................................................... 471
      cyg_ppp_chat() ................................................................. 473
55. Installing and Configuring PPP ....................................................... 475
      Including PPP in a Configuration ..................................... 475
      Configuring PPP .............................................................. 475
56. CHAT Scripts ............................................................................... 479
      Chat Script ....................................................................... 479
      ABORT Strings .................................................................. 480
      TIMEOUT ........................................................................... 480
      Sending EOT ...................................................................... 481
      Escape Sequences ............................................................ 481
57. PPP Enabled Device Drivers ........................................................... 483
58. Testing ....................................................................................... 485
      Test Programs .................................................................... 485
      Test Script ......................................................................... 486
XXI. Ethernet Device Drivers ..................................................................................................................... 489

59. Generic Ethernet Device Driver ........................................................................................................ 491
Generic Ethernet API ............................................................................................................................ 491
Review of the functions .......................................................................................................................... 493
  Init function........................................................................................................................................ 493
  Start function ..................................................................................................................................... 494
  Stop function ...................................................................................................................................... 494
  Control function ............................................................................................................................... 494
  Can-send function ........................................................................................................................... 495
  Send function ..................................................................................................................................... 495
  Deliver function .................................................................................................................................. 496
  Receive function .................................................................................................................................. 497
  Poll function ....................................................................................................................................... 497
  Interrupt-vector function .................................................................................................................. 497
Upper Layer Functions .......................................................................................................................... 498
  Callback Init function ....................................................................................................................... 498
  Callback Tx-Done function ............................................................................................................... 498
  Callback Receive function ............................................................................................................... 498
Calling graph for Transmission and Reception ..................................................................................... 499
  Transmission ...................................................................................................................................... 499
  Receive ............................................................................................................................................... 499

XXII. Ethernet PHY Device Support ....................................................................................................... 501

60. Ethernet PHY Device Support ........................................................................................................ 503
  Ethernet PHY Device API ................................................................................................................. 503

XXIII. SNMP .............................................................................................................................................. 505

61. SNMP for eCos ................................................................................................................................. 507
  Version ............................................................................................................................................... 507
  SNMP packages in the eCos source repository ................................................................................. 507
  MIBs supported ............................................................................................................................... 507
  Changes to eCos sources .................................................................................................................. 508
  Starting the SNMP Agent ................................................................................................................. 508
Configuring eCos .................................................................................................................................... 509
  Version usage (v1, v2 or v3) ............................................................................................................... 509
  Traps ................................................................................................................................................ 510
  snmpd.conf file ............................................................................................................................... 510
Test cases .............................................................................................................................................. 511
  SNMP clients and package use ......................................................................................................... 511
  Unimplemented features ................................................................................................................ 512
  MIB Compiler .................................................................................................................................. 512
  snmpd.conf ..................................................................................................................................... 513

XXIV. Embedded HTTP Server ............................................................................................................. 523

62. Embedded HTTP Server .................................................................................................................. 525
  Introduction ...................................................................................................................................... 525
  Server Organization ......................................................................................................................... 525
Server Configuration ........................................................................................................................... 526
  CYGNUM_HTTPD_SERVER_PORT .................................................................................................... 526
  CYGDAT_HTTPD_SERVER_ID ............................................................................................................ 526
  CYGNUM_HTTPD_THREAD_COUNT ................................................................................................. 526
  CYGNUM_HTTPD_THREAD_PRIORITY ............................................................................................. 526
  CYGNUM_HTTPD_THREAD_STACK_SIZE ......................................................................................... 526
FTP Client API
Variable Size Allocation Pools
CYNUM_HTTPD_SERVER_BUFFER_SIZE
CYNUM_HTTPD_SERVER_AUTO_START
CYNUM_HTTPD_SERVER_DELAY
Support Functions and Macros
HTTP Support
General HTML Support
Table Support
Forms Support
Predefined Handlers
System Monitor

XXV. FTP Client for eCos TCP/IP Stack

63. FTP Client Features
FTP Client API
ftp_get
ftp_put
ftpcient_print

XXVI. Simple Network Time Protocol Client

64. The SNTP Client
Starting the SNTP client
What it does
Configuring the unicast list of NTP servers
Warning: timestamp wrap around
The SNTP test program

XXVII. Another Tiny HTTP Server for eCos

65. The ATHTTP Server
Features
Starting the server
MIME types
MIME Types for Chunked Frames
C language callback functions
CGI
CGI via objloader
CGI via the simple tcl interpreter
start_chunked
write_chunked
der_chunked
tcl hello world example
Authentication
Directory Listing
Form Variables
Internal Resources

XXVIII. Memory Allocation

66. eCos Memory Pools
eCos Memory pools
Variable Size Allocation Pools
Fixed Size Allocation Pools
stdlib malloc Pools
XXXVI. eCos Synthetic Target .................................................................................................................. 671
  Overview ............................................................................................................................................... 673
  Installation ........................................................................................................................................... 677
  Running a Synthetic Target Application ............................................................................................... 679
  The I/O Auxiliary’s User Interface .......................................................................................................... 685
  The Console Device ................................................................................................................................. 691
  System Calls .......................................................................................................................................... 693
  Writing New Devices - target .................................................................................................................. 695
  Writing New Devices - host ...................................................................................................................... 701
  Porting ................................................................................................................................................... 711

XXXVII. M68000 Architectural Support ...................................................................................................... 715
  Overview ............................................................................................................................................... 717
  Configuration ........................................................................................................................................ 719
  The HAL Port ....................................................................................................................................... 721

XXXVIII. Freescale MCFxxxx Variant Support .......................................................................................... 729
  MCFxxxx ColdFire Processors ............................................................................................................... 731

XXXIX. Freescale MCF5272 Processor Support .......................................................................................... 737
  The MCF5272 ColdFire Processor ......................................................................................................... 739

XL. Freescale M5272C3 Board Support .................................................................................................... 743
  Overview ............................................................................................................................................... 745
  Setup .................................................................................................................................................... 747
  Configuration ...................................................................................................................................... 751
  The HAL Port ..................................................................................................................................... 755

XLI. SuperH SH4-202 MicroDev Board Support ....................................................................................... 757
  Overview ............................................................................................................................................... 759
  Setup .................................................................................................................................................... 761
  Configuration ...................................................................................................................................... 767
  The HAL Port ..................................................................................................................................... 769

XLII. SA11X0 USB Device Driver ............................................................................................................. 771
  SA11X0 USB Device Driver .................................................................................................................... 773

XLIII. NEC uPD985xx USB Device Driver ................................................................................................ 777
  NEC uPD985xx USB Device Driver ........................................................................................................ 779

XLIV. Synthetic Target Ethernet Driver .................................................................................................... 783
  Synthetic Target Ethernet Driver ........................................................................................................... 785

XLV. Freescale ColdFire Ethernet Driver .................................................................................................. 791
  Freescale ColdFire Ethernet Driver ........................................................................................................ 793

XLVI. Freescale MCFxxxx Serial Driver ..................................................................................................... 795
  MCFxxxx Serial Driver ........................................................................................................................... 797

XLVII. Synthetic Target Watchdog Device .................................................................................................. 799
  Synthetic Target Watchdog Device .......................................................................................................... 801

XLVIII. Dallas DS1307 Wallclock Device Driver ....................................................................................... 805
  Dallas DS1307 Wallclock Device Driver ................................................................................................ 807

XLIX. MMC MultiMedia Card Disk Driver ................................................................................................ 809
  Device Driver for MMC MultiMedia Cards ............................................................................................. 811

L. Synthetic Target Framebuffer Device .................................................................................................... 813
  Synthetic Target Framebuffer Device ...................................................................................................... 815
LII. AMD AM29xxxx Flash Device Driver ................................................................................................. 819
    Overview ........................................................................................................................................ 821
    Instantiating an AM29xxxx Device ................................................................................................. 823

LII. Intel Strata Flash Device Driver ...................................................................................................... 831
    Overview ........................................................................................................................................ 833
    Instantiating a Strata Device ........................................................................................................... 835
    Strata-Specific Functions ................................................................................................................ 845

LIII. Motorola MCF52xx ColdFire I'C Bus Driver .................................................................................... 847
    Motorola MCF52xx Coldfire I'C Bus Driver ..................................................................................... 849
List of Tables

8-1. Behavior of math exception handling................................................................................. 81
I. The eCos Kernel
Kernel Overview

Name
Kernel — Overview of the eCos Kernel

Description
The kernel is one of the key packages in all of eCos. It provides the core functionality needed for developing multi-threaded applications:

1. The ability to create new threads in the system, either during startup or when the system is already running.
2. Control over the various threads in the system, for example manipulating their priorities.
3. A choice of schedulers, determining which thread should currently be running.
4. A range of synchronization primitives, allowing threads to interact and share data safely.
5. Integration with the system’s support for interrupts and exceptions.

In some other operating systems the kernel provides additional functionality. For example the kernel may also provide memory allocation functionality, and device drivers may be part of the kernel as well. This is not the case for eCos. Memory allocation is handled by a separate package. Similarly each device driver will typically be a separate package. Various packages are combined and configured using the eCos configuration technology to meet the requirements of the application.

The eCos kernel package is optional. It is possible to write single-threaded applications which do not use any kernel functionality, for example RedBoot. Typically such applications are based around a central polling loop, continually checking all devices and taking appropriate action when I/O occurs. A small amount of calculation is possible every iteration, at the cost of an increased delay between an I/O event occurring and the polling loop detecting the event. When the requirements are straightforward it may well be easier to develop the application using a polling loop, avoiding the complexities of multiple threads and synchronization between threads. As requirements get more complicated a multi-threaded solution becomes more appropriate, requiring the use of the kernel. In fact some of the more advanced packages in eCos, for example the TCP/IP stack, use multi-threading internally. Therefore if the application uses any of those packages then the kernel becomes a required package, not an optional one.

The kernel functionality can be used in one of two ways. The kernel provides its own C API, with functions like \texttt{cyg\_thread\_create} and \texttt{cyg\_mutex\_lock}. These can be called directly from application code or from other packages. Alternatively there are a number of packages which provide compatibility with existing API’s, for example POSIX threads or \texttt{µITRON}. These allow application code to call standard functions such as \texttt{pthread\_create}, and those functions are implemented using the basic functionality provided by the eCos kernel. Using compatibility packages in an eCos application can make it much easier to reuse code developed in other environments, and to share code.

Although the different compatibility packages have similar requirements on the underlying kernel, for example the ability to create a new thread, there are differences in the exact semantics. For example, strict \texttt{µITRON} compliance requires that kernel timeslicing is disabled. This is achieved largely through the configuration technology. The kernel provides a number of configuration options that control the exact semantics that are provided, and the various compatibility packages require particular settings for those options. This has two important consequences. First, it is not usually possible to have two different compatibility packages in one eCos configuration because they will have conflicting requirements on the underlying kernel. Second, the semantics of the kernel’s own API are only loosely defined because of the many configuration options. For
Kernel Overview

example `cyg_mutex_lock` will always attempt to lock a mutex, but various configuration options determine the behaviour when the mutex is already locked and there is a possibility of priority inversion.

The optional nature of the kernel package presents some complications for other code, especially device drivers. Wherever possible a device driver should work whether or not the kernel is present. However there are some parts of the system, especially those related to interrupt handling, which should be implemented differently in multi-threaded environments containing the eCos kernel and in single-threaded environments without the kernel. To cope with both scenarios the common HAL package provides a driver API, with functions such as `cyg_drv_interrupt_attach`. When the kernel package is present these driver API functions map directly on to the equivalent kernel functions such as `cyg_interrupt_attach`, using macros to avoid any overheads. When the kernel is absent the common HAL package implements the driver API directly, but this implementation is simpler than the one in the kernel because it can assume a single-threaded environment.

Schedulers

When a system involves multiple threads, a scheduler is needed to determine which thread should currently be running. The eCos kernel can be configured with one of two schedulers, the bitmap scheduler and the multi-level queue (MLQ) scheduler. The bitmap scheduler is somewhat more efficient, but has a number of limitations. Most systems will instead use the MLQ scheduler. Other schedulers may be added in the future, either as extensions to the kernel package or in separate packages.

Both the bitmap and the MLQ scheduler use a simple numerical priority to determine which thread should be running. The number of priority levels is configurable via the option `CYGNUM_KERNEL_SCHED_PRIORITIES`, but a typical system will have up to 32 priority levels. Therefore thread priorities will be in the range 0 to 31, with 0 being the highest priority and 31 the lowest. Usually only the system’s idle thread will run at the lowest priority. Thread priorities are absolute, so the kernel will only run a lower-priority thread if all higher-priority threads are currently blocked.

The bitmap scheduler only allows one thread per priority level, so if the system is configured with 32 priority levels then it is limited to only 32 threads — still enough for many applications. A simple bitmap can be used to keep track of which threads are currently runnable. Bitmaps can also be used to keep track of threads waiting on a mutex or other synchronization primitive. Identifying the highest-priority runnable or waiting thread involves a simple operation on the bitmap, and an array index operation can then be used to get hold of the thread data structure itself. This makes the bitmap scheduler fast and totally deterministic.

The MLQ scheduler allows multiple threads to run at the same priority. This means that there is no limit on the number of threads in the system, other than the amount of memory available. However operations such as finding the highest priority runnable thread are a little bit more expensive than for the bitmap scheduler.

Optionally the MLQ scheduler supports timeslicing, where the scheduler automatically switches from one runnable thread to another when some number of clock ticks have occurred. Timeslicing only comes into play when there are two runnable threads at the same priority and no higher priority runnable threads. If timeslicing is disabled then a thread will not be preempted by another thread of the same priority, and will continue running until either it explicitly yields the processor or until it blocks by, for example, waiting on a synchronization primitive. The configuration options `CYGSEM_KERNEL_SCHED_TIMESLICE` and `CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS` control timeslicing. The bitmap scheduler does not provide timeslicing support. It only allows one thread per priority level, so it is not possible to preempt the current thread in favour of another one with the same priority.

Another important configuration option that affects the MLQ scheduler is `CYGIMP_KERNEL_SCHED_SORTED_QUEUES`. This determines what happens when a thread blocks, for example by waiting on a semaphore which has no pending events. The default behaviour of the system is last-in-first-out queuing. For example if several threads are waiting on a semaphore and an event is posted, the thread that gets woken up is the last one that called `cyg_semaphore_wait`. This allows for a simple and fast
implementation of both the queue and dequeue operations. However if there are several queued threads with
different priorities, it may not be the highest priority one that gets woken up. In practice this is rarely a
problem: usually there will be at most one thread waiting on a queue, or when there are several threads they
will be of the same priority. However if the application does require strict priority queueing then the option
CYGIMP_KERNEL_SCHED_SORTED_QUEUES should be enabled. There are disadvantages: more work is needed
whenever a thread is queued, and the scheduler needs to be locked for this operation so the system’s dispatch
latency is worse. If the bitmap scheduler is used then priority queueing is automatic and does not involve any
penalties.

Some kernel functionality is currently only supported with the MLQ scheduler, not the bitmap scheduler. This
includes support for SMP systems, and protection against priority inversion using either mutex priority ceilings
or priority inheritance.

Synchronization Primitives

The eCos kernel provides a number of different synchronization primitives: mutexes, condition variables,
counting semaphores, mail boxes and event flags.

Mutexes serve a very different purpose from the other primitives. A mutex allows multiple threads to share a
resource safely: a thread locks a mutex, manipulates the shared resource, and then unlocks the mutex again. The
other primitives are used to communicate information between threads, or alternatively from a DSR associated
with an interrupt handler to a thread.

When a thread that has locked a mutex needs to wait for some condition to become true, it should use a
condition variable. A condition variable is essentially just a place for a thread to wait, and which another
thread, or DSR, can use to wake it up. When a thread waits on a condition variable it releases the mutex
before waiting, and when it wakes up it reacquires it before proceeding. These operations are atomic so that
synchronization race conditions cannot be introduced.

A counting semaphore is used to indicate that a particular event has occurred. A consumer thread can wait
for this event to occur, and a producer thread or a DSR can post the event. There is a count associated with
the semaphore so if the event occurs multiple times in quick succession this information is not lost, and the
appropriate number of semaphore wait operations will succeed.

Mail boxes are also used to indicate that a particular event has occurred, and allows for one item of data to be
exchanged per event. Typically this item of data would be a pointer to some data structure. Because of the need
to store this extra data, mail boxes have a finite capacity. If a producer thread generates mail box events faster
than they can be consumed then, to avoid overflow, it will be blocked until space is again available in the mail
box. This means that mail boxes usually cannot be used by a DSR to wake up a thread. Instead mail boxes are
typically only used between threads.

Event flags can be used to wait on some number of different events, and to signal that one or several of these
events have occurred. This is achieved by associating bits in a bit mask with the different events. Unlike a
counting semaphore no attempt is made to keep track of the number of events that have occurred, only the fact
that an event has occurred at least once. Unlike a mail box it is not possible to send additional data with the
event, but this does mean that there is no possibility of an overflow and hence event flags can be used between
a DSR and a thread as well as between threads.

The eCos common HAL package provides its own device driver API which contains some of the above syn-
chronization primitives. These allow the DSR for an interrupt handler to signal events to higher-level code. If
the configuration includes the eCos kernel package then the driver API routines map directly on to the equiva-
 lent kernel routines, allowing interrupt handlers to interact with threads. If the kernel package is not included
and the application consists of just a single thread running in polled mode then the driver API is implemented
entirely within the common HAL, and with no need to worry about multiple threads the implementation can
obviously be rather simpler.
Threads and Interrupt Handling

During normal operation the processor will be running one of the threads in the system. This may be an application thread, a system thread running inside say the TCP/IP stack, or the idle thread. From time to time a hardware interrupt will occur, causing control to be transferred briefly to an interrupt handler. When the interrupt has been completed the system’s scheduler will decide whether to return control to the interrupted thread or to some other runnable thread.

Threads and interrupt handlers must be able to interact. If a thread is waiting for some I/O operation to complete, the interrupt handler associated with that I/O must be able to inform the thread that the operation has completed. This can be achieved in a number of ways. One very simple approach is for the interrupt handler to set a volatile variable. A thread can then poll continuously until this flag is set, possibly sleeping for a clock tick in between. Polling continuously means that the cpu time is not available for other activities, which may be acceptable for some but not all applications. Polling once every clock tick imposes much less overhead, but means that the thread may not detect that the I/O event has occurred until an entire clock tick has elapsed. In typical systems this could be as long as 10 milliseconds. Such a delay might be acceptable for some applications, but not all.

A better solution would be to use one of the synchronization primitives. The interrupt handler could signal a condition variable, post to a semaphore, or use one of the other primitives. The thread would perform a wait operation on the same primitive. It would not consume any cpu cycles until the I/O event had occurred, and when the event does occur the thread can start running again immediately (subject to any higher priority threads that might also be runnable).

Synchronization primitives constitute shared data, so care must be taken to avoid problems with concurrent access. If the thread that was interrupted was just performing some calculations then the interrupt handler could manipulate the synchronization primitive quite safely. However if the interrupted thread happened to be inside some kernel call then there is a real possibility that some kernel data structure will be corrupted. One way of avoiding such problems would be for the kernel functions to disable interrupts when executing any critical region. On most architectures this would be simple to implement and very fast, but it would mean that interrupts would be disabled often and for quite a long time. For some applications that might not matter, but many embedded applications require that the interrupt handler run as soon as possible after the hardware interrupt has occurred. If the kernel relied on disabling interrupts then it would not be able to support such applications.

Instead the kernel uses a two-level approach to interrupt handling. Associated with every interrupt vector is an Interrupt Service Routine or ISR, which will run as quickly as possible so that it can service the hardware. However an ISR can make only a small number of kernel calls, mostly related to the interrupt subsystem, and it cannot make any call that would cause a thread to wake up. If an ISR detects that an I/O operation has completed and hence that a thread should be woken up, it can cause the associated Deferred Service Routine or DSR to run. A DSR is allowed to make more kernel calls, for example it can signal a condition variable or post to a semaphore.

Disabling interrupts prevents ISRs from running, but very few parts of the system disable interrupts and then only for short periods of time. The main reason for a thread to disable interrupts is to manipulate some state that is shared with an ISR. For example if a thread needs to add another buffer to a linked list of free buffers and the ISR may remove a buffer from this list at any time, the thread would need to disable interrupts for the few instructions needed to manipulate the list. If the hardware raises an interrupt at this time, it remains pending until interrupts are reenabled.

Analogous to interrupts being disabled or enabled, the kernel has a scheduler lock. The various kernel functions such as cyg_mutex_lock and cyg_semaphore_post will claim the scheduler lock, manipulate the kernel data structures, and then release the scheduler lock. If an interrupt results in a DSR being requested and the scheduler is currently locked, the DSR remains pending. When the scheduler lock is released any pending DSRs will run. These may post events to synchronization primitives, causing other higher priority threads to be woken up.
For an example, consider the following scenario. The system has a high-priority thread A, responsible for processing some data coming from an external device. This device will raise an interrupt when data is available. There are two other threads B and C which spend their time performing calculations and occasionally writing results to a display of some sort. This display is a shared resource, so a mutex is used to control access.

At a particular moment in time thread A is likely to be blocked, waiting on a semaphore or another synchronization primitive until data is available. Thread B might be running performing some calculations, and thread C is runnable waiting for its next timeslice. Interrupts are enabled, and the scheduler is unlocked because none of the threads are in the middle of a kernel operation. At this point, the device raises an interrupt. The hardware transfers control to a low-level interrupt handler provided by eCos, which works out exactly which interrupt occurs, and then the corresponding ISR is run. This ISR manipulates the hardware as appropriate, determines that there is now data available, and wants to wake up thread A by posting to the semaphore. However, ISR’s are not allowed to call `cyg_semaphore_post` directly, so instead the ISR requests that its associated DSR be run and returns. There are no more interrupts to be processed, so the kernel next checks for DSR’s. One DSR is pending, and the scheduler is currently unlocked, so the DSR can run immediately and post the semaphore. This will have the effect of making thread A runnable again, so the scheduler’s data structures are adjusted accordingly. When the DSR returns, thread B is no longer the highest priority runnable thread so it will be suspended, and instead thread A gains control over the CPU.

In the above example, no kernel data structures were being manipulated at the exact moment that the interrupt happened. However, that cannot be assumed. Suppose that thread B had finished its current set of calculations and wanted to write the results to the display. It would claim the appropriate mutex and manipulate the display. Now suppose that thread B was timesliced in favour of thread C, and that thread C also finished its calculations and wanted to write the results to the display. It would call `cyg_mutex_lock`. This kernel call locks the scheduler, examines the current state of the mutex, discovers that the mutex is already owned by another thread, suspends the current thread, and switches control to another runnable thread. Another interrupt happens in the middle of this `cyg_mutex_lock` call, causing the ISR to run immediately. The ISR decides that thread A should be woken up so it requests that its DSR be run and returns back to the kernel. At this point there is a pending DSR, but the scheduler is still locked by the call to `cyg_mutex_lock` so the DSR cannot run immediately. Instead, the call to `cyg_mutex_lock` is allowed to continue, which at some point involves unlocking the scheduler. The pending DSR can now run, safely post the semaphore, and thus wake up thread A.

If the ISR had called `cyg_semaphore_post` directly rather than leaving it to a DSR, it is likely that there would have been some sort of corruption of a kernel data structure. For example, the kernel might have completely lost track of one of the threads, and that thread would never have run again. The two-level approach to interrupt handling, ISR’s and DSR’s, prevents such problems with no need to disable interrupts.

**Calling Contexts**

eCos defines a number of contexts. Only certain calls are allowed from inside each context, for example most operations on threads or synchronization primitives are not allowed from ISR context. The different contexts are initialization, thread, ISR, and DSR.

When eCos starts up, it goes through a number of phases, including setting up the hardware and invoking C++ static constructors. During this time, interrupts are disabled and the scheduler is locked. When a configuration includes the kernel package, the final operation is a call to `cyg Scheduler_start`. At this point, interrupts are enabled, the scheduler is unlocked, and control is transferred to the highest priority runnable thread. If the configuration also includes the C library package, then usually the C library startup package will have created a thread which will call the application’s main entry point.

Some application code can also run before the scheduler is started, and this code runs in initialization context. If the application is written partly or completely in C++, then the constructors for any static objects will be run. Alternatively, application code can define a function `cyg user start` which gets called after any C++ static constructors. This allows applications to be written entirely in C.
Kernel Overview

void
cyg_user_start(void)
{
    /* Perform application-specific initialization here */
}

It is not necessary for applications to provide a cyg_user_start function since the system will provide a default implementation which does nothing.

Typical operations that are performed from inside static constructors or cyg_user_start include creating threads, synchronization primitives, setting up alarms, and registering application-specific interrupt handlers. In fact for many applications all such creation operations happen at this time, using statically allocated data, avoiding any need for dynamic memory allocation or other overheads.

Code running in initialization context runs with interrupts disabled and the scheduler locked. It is not permitted to reenable interrupts or unlock the scheduler because the system is not guaranteed to be in a totally consistent state at this point. A consequence is that initialization code cannot use synchronization primitives such as cyg_semaphore_wait to wait for an external event. It is permitted to lock and unlock a mutex: there are no other threads running so it is guaranteed that the mutex is not yet locked, and therefore the lock operation will never block; this is useful when making library calls that may use a mutex internally.

At the end of the startup sequence the system will call cyg_scheduler_start and the various threads will start running. In thread context nearly all of the kernel functions are available. There may be some restrictions on interrupt-related operations, depending on the target hardware. For example the hardware may require that interrupts be acknowledged in the ISR or DSR before control returns to thread context, in which case cyg_interrupt_acknowledge should not be called by a thread.

At any time the processor may receive an external interrupt, causing control to be transferred from the current thread. Typically a VSR provided by eCos will run and determine exactly which interrupt occurred. Then the VSR will switch to the appropriate ISR, which can be provided by a HAL package, a device driver, or by the application. During this time the system is running at ISR context, and most of the kernel function calls are disallowed. This includes the various synchronization primitives, so for example an ISR is not allowed to post to a semaphore to indicate that an event has happened. Usually the only operations that should be performed from inside an ISR are ones related to the interrupt subsystem itself, for example masking an interrupt or acknowledging that an interrupt has been processed. On SMP systems it is also possible to use spinlocks from ISR context.

When an ISR returns it can request that the corresponding DSR be run as soon as it is safe to do so, and that will run in DSR context. This context is also used for running alarm functions, and threads can switch temporarily to DSR context by locking the scheduler. Only certain kernel functions can be called from DSR context, although more than in ISR context. In particular it is possible to use any synchronization primitives which cannot block. These include cyg_semaphore_post, cyg_cond_signal, cyg_cond_broadcast, cyg_flag_setbits, and cyg_mbox_tryput. It is not possible to use any primitives that may block such as cyg_semaphore_wait, cyg_mutex_lock, or cyg_mbox_put. Calling such functions from inside a DSR may cause the system to hang.

The specific documentation for the various kernel functions gives more details about valid contexts.

Error Handling and Assertions

In many APIs each function is expected to perform some validation of its parameters and possibly of the current state of the system. This is supposed to ensure that each function is used correctly, and that application code is not attempting to perform a semaphore operation on a mutex or anything like that. If an error is detected then a suitable error code is returned, for example the POSIX function pthread_mutex_lock can return various
error codes including `EINVAL` and `EDEADLK`. There are a number of problems with this approach, especially in the context of deeply embedded systems:

1. Performing these checks inside the mutex lock and all the other functions requires extra cpu cycles and adds significantly to the code size. Even if the application is written correctly and only makes system function calls with sensible arguments and under the right conditions, these overheads still exist.

2. Returning an error code is only useful if the calling code detects these error codes and takes appropriate action. In practice the calling code will often ignore any errors because the programmer “knows” that the function is being used correctly. If the programmer is mistaken then an error condition may be detected and reported, but the application continues running anyway and is likely to fail some time later in mysterious ways.

3. If the calling code does always check for error codes, that adds yet more cpu cycles and code size overhead.

4. Usually there will be no way to recover from certain errors, so if the application code detected an error such as `EINVAL` then all it could do is abort the application somehow.

The approach taken within the eCos kernel is different. Functions such as `cyg_mutex_lock` will not return an error code. Instead they contain various assertions, which can be enabled or disabled. During the development process assertions are normally left enabled, and the various kernel functions will perform parameter checks and other system consistency checks. If a problem is detected then an assertion failure will be reported and the application will be terminated. In a typical debug session a suitable breakpoint will have been installed and the developer can now examine the state of the system and work out exactly what is going on. Towards the end of the development cycle assertions will be disabled by manipulating configuration options within the eCos infrastructure package, and all assertions will be eliminated at compile-time. The assumption is that by this time the application code has been mostly debugged: the initial version of the code might have tried to perform a semaphore operation on a mutex, but any problems like that will have been fixed some time ago. This approach has a number of advantages:

1. In the final application there will be no overheads for checking parameters and other conditions. All that code will have been eliminated at compile-time.

2. Because the final application will not suffer any overheads, it is reasonable for the system to do more work during the development process. In particular the various assertions can test for more error conditions and more complicated errors. When an error is detected it is possible to give a text message describing the error rather than just return an error code.

3. There is no need for application programmers to handle error codes returned by various kernel function calls. This simplifies the application code.

4. If an error is detected then an assertion failure will be reported immediately and the application will be halted. There is no possibility of an error condition being ignored because application code did not check for an error code.

Although none of the kernel functions return an error code, many of them do return a status condition. For example the function `cyg_semaphore_timed_wait` waits until either an event has been posted to a semaphore, or until a certain number of clock ticks have occurred. Usually the calling code will need to know whether the wait operation succeeded or whether a timeout occurred. `cyg_semaphore_timed_wait` returns a boolean: a return value of zero or false indicates a timeout, a non-zero return value indicates that the wait succeeded.

In conventional APIs one common error conditions is lack of memory. For example the POSIX function `pthread_create` usually has to allocate some memory dynamically for the thread stack and other per-thread data. If the target hardware does not have enough memory to meet all demands, or more commonly if the application contains a memory leak, then there may not be enough memory available and the function call would fail. The eCos kernel avoids such problems by never performing any dynamic memory allocation. Instead it is the responsibility of the application code to provide all the memory required for kernel data structures and
other needs. In the case of `cyg_thread_create` this means a `cyg_thread` data structure to hold the thread
details, and a char array for the thread stack.

In many applications this approach results in all data structures being allocated statically rather than dynami-
cally. This has several advantages. If the application is in fact too large for the target hardware's memory then
there will be an error at link-time rather than at run-time, making the problem much easier to diagnose. Static
allocation does not involve any of the usual overheads associated with dynamic allocation, for example there
is no need to keep track of the various free blocks in the system, and it may be possible to eliminate `malloc`
from the system completely. Problems such as fragmentation and memory leaks cannot occur if all data is
allocated statically. However, some applications are sufficiently complicated that dynamic memory allocation
is required, and the various kernel functions do not distinguish between statically and dynamically allocated
memory. It still remains the responsibility of the calling code to ensure that sufficient memory is available, and
passing null pointers to the kernel will result in assertions or system failure.
SMP Support

Name
SMP — Support Symmetric Multiprocessing Systems

Description
eCos contains support for limited Symmetric Multi-Processing (SMP). This is only available on selected architectures and platforms. The implementation has a number of restrictions on the kind of hardware supported. These are described in the Section called SMP Support in Chapter 4.

The following sections describe the changes that have been made to the eCos kernel to support SMP operation.

System Startup
The system startup sequence needs to be somewhat different on an SMP system, although this is largely transparent to application code. The main startup takes place on only one CPU, called the primary CPU. All other CPUs, the secondary CPUs, are either placed in suspended state at reset, or are captured by the HAL and put into a spin as they start up. The primary CPU is responsible for copying the DATA segment and zeroing the BSS (if required), calling HAL variant and platform initialization routines and invoking constructors. It then calls `cyg_start` to enter the application. The application may then create extra threads and other objects.

It is only when the application calls `cyg_scheduler_start` that the secondary CPUs are initialized. This routine scans the list of available secondary CPUs and invokes `HAL_SMP_CPU_START` to start each CPU. Finally it calls an internal function `Cyg_Scheduler::start_cpu` to enter the scheduler for the primary CPU.

Each secondary CPU starts in the HAL, where it completes any per-CPU initialization before calling into the kernel at `cyg_kernel_cpu_startup`. Here it claims the scheduler lock and calls `Cyg_Scheduler::start_cpu`.

`Cyg_Scheduler::start_cpu` is common to both the primary and secondary CPUs. The first thing this code does is to install an interrupt object for this CPU’s inter-CPU interrupt. From this point on the code is the same as for the single CPU case: an initial thread is chosen and entered.

From this point on the CPUs are all equal, eCos makes no further distinction between the primary and secondary CPUs. However, the hardware may still distinguish between them as far as interrupt delivery is concerned.

Scheduling
To function correctly an operating system kernel must protect its vital data structures, such as the run queues, from concurrent access. In a single CPU system the only concurrent activities to worry about are asynchronous interrupts. The kernel can easily guard its data structures against these by disabling interrupts. However, in a multi-CPU system, this is inadequate since it does not block access by other CPUs.

The eCos kernel protects its vital data structures using the scheduler lock. In single CPU systems this is a simple counter that is atomically incremented to acquire the lock and decremented to release it. If the lock is decremented to zero then the scheduler may be invoked to choose a different thread to run. Because interrupts may continue to be serviced while the scheduler lock is claimed, ISRs are not allowed to access kernel data structures, or call kernel routines that can. Instead all such operations are deferred to an associated DSR routine that is run during the lock release operation, when the data structures are in a consistent state.


SMP Support

By choosing a kernel locking mechanism that does not rely on interrupt manipulation to protect data structures, it is easier to convert eCos to SMP than would otherwise be the case. The principal change needed to make eCos SMP-safe is to convert the scheduler lock into a nestable spin lock. This is done by adding a spinlock and a CPU id to the original counter.

The algorithm for acquiring the scheduler lock is very simple. If the scheduler lock’s CPU id matches the current CPU then it can just increment the counter and continue. If it does not match, the CPU must spin on the spinlock, after which it may increment the counter and store its own identity in the CPU id.

To release the lock, the counter is decremented. If it goes to zero the CPU id value must be set to NONE and the spinlock cleared.

To protect these sequences against interrupts, they must be performed with interrupts disabled. However, since these are very short code sequences, they will not have an adverse effect on the interrupt latency.

Beyond converting the scheduler lock, further preparing the kernel for SMP is a relatively minor matter. The main changes are to convert various scalar housekeeping variables into arrays indexed by CPU id. These include the current thread pointer, the need_reschedule flag and the timeslice counter.

At present only the Multi-Level Queue (MLQ) scheduler is capable of supporting SMP configurations. The main change made to this scheduler is to cope with having several threads in execution at the same time. Running threads are marked with the CPU that they are executing on. When scheduling a thread, the scheduler skips past any running threads until it finds a thread that is pending. While not a constant-time algorithm, as in the single CPU case, this is still deterministic, since the worst case time is bounded by the number of CPUs in the system.

A second change to the scheduler is in the code used to decide when the scheduler should be called to choose a new thread. The scheduler attempts to keep the n CPUs running the n highest priority threads. Since an event or interrupt on one CPU may require a reschedule on another CPU, there must be a mechanism for deciding this. The algorithm currently implemented is very simple. Given a thread that has just been awakened (or had its priority changed), the scheduler scans the CPUs, starting with the one it is currently running on, for a current thread that is of lower priority than the new one. If one is found then a reschedule interrupt is sent to that CPU and the scan continues, but now using the current thread of the rescheduled CPU as the candidate thread. In this way the new thread gets to run as quickly as possible, hopefully on the current CPU, and the remaining CPUs will pick up the remaining highest priority threads as a consequence of processing the reschedule interrupt.

The final change to the scheduler is in the handling of timeslicing. Only one CPU receives timer interrupts, although all CPUs must handle timeslicing. To make this work, the CPU that receives the timer interrupt decrements the timeslice counter for all CPUs, not just its own. If the counter for a CPU reaches zero, then it sends a timeslice interrupt to that CPU. On receiving the interrupt the destination CPU enters the scheduler and looks for another thread at the same priority to run. This is somewhat more efficient than distributing clock ticks to all CPUs, since the interrupt is only needed when a timeslice occurs.

All existing synchronization mechanisms work as before in an SMP system. Additional synchronization mechanisms have been added to provide explicit synchronization for SMP, in the form of spinlocks.

SMP Interrupt Handling

The main area where the SMP nature of a system requires special attention is in device drivers and especially interrupt handling. It is quite possible for the ISR, DSR and thread components of a device driver to execute on different CPUs. For this reason it is much more important that SMP-capable device drivers use the interrupt-related functions correctly. Typically a device driver would use the driver API rather than call the kernel directly, but it is unlikely that anybody would attempt to use a multiprocessor system without the kernel package.
Two new functions have been added to the Kernel API to do interrupt routing: `cyg_interrupt_set_cpu` and `cyg_interrupt_get_cpu`. Although not currently supported, special values for the `cpu` argument may be used in future to indicate that the interrupt is being routed dynamically or is CPU-local. Once a vector has been routed to a new CPU, all other interrupt masking and configuration operations are relative to that CPU, where relevant.

There are more details of how interrupts should be handled in SMP systems in the Section called SMP Support in Chapter 13.
SMP Support
Thread creation

Name
cyg_thread_create — Create a new thread

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_thread_create(cyg_addrword_t sched_info, cyg_thread_entry_t* entry, 
cyg_addrword_t entry_data, char* name, void* stack_base, cyg_ucount32 stack_size, 
cyg_handle_t* handle, cyg_thread* thread);

Description

The cyg_thread_create function allows application code and eCos packages to create new threads. In many applications this only happens during system initialization and all required data is allocated statically. However additional threads can be created at any time, if necessary. A newly created thread is always in suspended state and will not start running until it has been resumed via a call to cyg_thread_resume. Also, if threads are created during system initialization then they will not start running until the eCos scheduler has been started.

The name argument is used primarily for debugging purposes, making it easier to keep track of which cyg_thread structure is associated with which application-level thread. The kernel configuration option CYGVAR_KERNEL_THREADS_NAME controls whether or not this name is actually used.

On creation each thread is assigned a unique handle, and this will be stored in the location pointed at by the handle argument. Subsequent operations on this thread including the required cyg_thread_resume should use this handle to identify the thread.

The kernel requires a small amount of space for each thread, in the form of a cyg_thread data structure, to hold information such as the current state of that thread. To avoid any need for dynamic memory allocation within the kernel this space has to be provided by higher-level code, typically in the form of a static variable. The thread argument provides this space.

Thread Entry Point

The entry point for a thread takes the form:

void
thread_entry_function(cyg_addrword_t data)
{
    ...
}

The second argument to cyg_thread_create is a pointer to such a function. The third argument entry_data is used to pass additional data to the function. Typically this takes the form of a pointer to some static data, or a small integer, or 0 if the thread does not require any additional data.
Thread creation

If the thread entry function ever returns then this is equivalent to the thread calling \texttt{cyg\_thread\_exit}. Even though the thread will no longer run again, it remains registered with the scheduler. If the application needs to re-use the \texttt{cyg\_thread} data structure then a call to \texttt{cyg\_thread\_delete} is required first.

Thread Priorities

The \texttt{sched\_info} argument provides additional information to the scheduler. The exact details depend on the scheduler being used. For the bitmap and mlqueue schedulers it is a small integer, typically in the range 0 to 31, with 0 being the highest priority. The lowest priority is normally used only by the system’s idle thread. The exact number of priorities is controlled by the kernel configuration option \texttt{CYGNUM\_KERNEL\_SCHED\_PRIORITIES}.

It is the responsibility of the application developer to be aware of the various threads in the system, including those created by eCos packages, and to ensure that all threads run at suitable priorities. For threads created by other packages the documentation provided by those packages should indicate any requirements.

The functions \texttt{cyg\_thread\_set\_priority}, \texttt{cyg\_thread\_get\_priority}, and \texttt{cyg\_thread\_get\_current\_priority} can be used to manipulate a thread’s priority.

Stacks and Stack Sizes

Each thread needs its own stack for local variables and to keep track of function calls and returns. Again it is expected that this stack is provided by the calling code, usually in the form of static data, so that the kernel does not need any dynamic memory allocation facilities. \texttt{cyg\_thread\_create} takes two arguments related to the stack, a pointer to the base of the stack and the total size of this stack. On many processors stacks actually descend from the top down, so the kernel will add the stack size to the base address to determine the starting location.

The exact stack size requirements for any given thread depend on a number of factors. The most important is of course the code that will be executed in the context of this code: if this involves significant nesting of function calls, recursion, or large local arrays, then the stack size needs to be set to a suitably high value. There are some architectural issues, for example the number of cpu registers and the calling conventions will have some effect on stack usage. Also, depending on the configuration, it is possible that some other code such as interrupt handlers will occasionally run on the current thread’s stack. This depends in part on configuration options such as \texttt{CYGIMP\_HAL\_COMMON\_INTERRUPTS\_USE\_INTERRUPT\_STACK} and \texttt{CYGSEM\_HAL\_COMMON\_INTERRUPTS\_ALLOW\_NESTING}.

Determining an application’s actual stack size requirements is the responsibility of the application developer, since the kernel cannot know in advance what code a given thread will run. However, the system does provide some hints about reasonable stack sizes in the form of two constants: \texttt{CYGNUM\_HAL\_STACK\_SIZE\_MINIMUM} and \texttt{CYGNUM\_HAL\_STACK\_SIZE\_TYPICAL}. These are defined by the appropriate HAL package. The \texttt{MINIMUM} value is appropriate for a thread that just runs a single function and makes very simple system calls. Trying to create a thread with a smaller stack than this is illegal. The \texttt{TYPICAL} value is appropriate for applications where application calls are nested no more than half a dozen or so levels, and there are no large arrays on the stack.

If the stack sizes are not estimated correctly and a stack overflow occurs, the probably result is some form of memory corruption. This can be very hard to track down. The kernel does contain some code to help detect stack overflows, controlled by the configuration option \texttt{CYGFUN\_KERNEL\_THREADS\_STACK\_CHECKING}: a small amount of space is reserved at the stack limit and filled with a special signature: every time a thread context switch occurs this signature is checked, and if invalid that is a good indication (but not absolute proof) that a stack overflow has occurred. This form of stack checking is enabled by default when the system is built with debugging enabled. A related configuration option is \texttt{CYGFUN\_KERNEL\_THREADS\_STACK\_MEASUREMENT}: enabling this option means that a thread can call the function \texttt{cyg\_thread\_measure\_stack\_usage} to find out
the maximum stack usage to date. Note that this is not necessarily the true maximum because, for example, it is possible that in the current run no interrupt occurred at the worst possible moment.

**Valid contexts**

cyg_thread_create may be called during initialization and from within thread context. It may not be called from inside a DSR.

**Example**

A simple example of thread creation is shown below. This involves creating five threads, one producer and four consumers or workers. The threads are created in the system’s cyg_user_start: depending on the configuration it might be more appropriate to do this elsewhere, for example inside main.

```c
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.h>

// These numbers depend entirely on your application
#define NUMBER_OF_WORKERS 4
#define PRODUCER_PRIORITY 10
#define WORKER_PRIORITY 11
#define PRODUCER_STACKSIZE CYGNUM_HAL_STACK_SIZE_TYPICAL
#define WORKER_STACKSIZE (CYGNUM_HAL_STACK_SIZE_MINIMUM + 1024)

static unsigned char producer_stack[PRODUCER_STACKSIZE];
static unsigned char worker_stacks[NUMBER_OF_WORKERS][WORKER_STACKSIZE];
static cyg_handle_t producer_handle, worker_handles[NUMBER_OF_WORKERS];
static cyg_thread producer_thread, worker_threads[NUMBER_OF_WORKERS];

static void producer(cyg_addrword_t data)
{
    ...
}

static void worker(cyg_addrword_t data)
{
    ...
}

void cyg_user_start(void)
{
    int i;

    cyg_thread_create(PRODUCER_PRIORITY, &producer, 0, "producer",
                      producer_stack, PRODUCER_STACKSIZE,
                      &producer_handle, &producer_thread);
    cyg_thread_resume(producer_handle);
    for (i = 0; i < NUMBER_OF_WORKERS; i++) {
        cyg_thread_create(WORKER_PRIORITY, &worker, i, "worker",
                          worker_stacks[i], WORKER_STACKSIZE,
                          &worker_handles[i], &worker_threads[i]);
        cyg_thread_resume(worker_handles[i]);
```

```
Thread Entry Points and C++

For code written in C++ the thread entry function must be either a static member function of a class or an ordinary function outside any class. It cannot be a normal member function of a class because such member functions take an implicit additional argument `this`, and the kernel has no way of knowing what value to use for this argument. One way around this problem is to make use of a special static member function, for example:

```cpp
class fred {
  public:
    void thread_function();
    static void static_thread_aux(cyg_addrword_t);
  
  void fred::static_thread_aux(cyg_addrword_t objptr)
  { 
    fred* object = static_cast<fred*>(objptr);
    object->thread_function();
  }

  static fred instance;

  extern "C" void
cyg_start( void )
  { 
    ... 
    cyg_thread_create( ..., 
      &fred::static_thread_aux,
      reinterpret_cast<cyg_addrword_t>(&instance),
      ...);
    ...
  }
```

Effectively this uses the `entry_data` argument to `cyg_thread_create` to hold the `this` pointer. Unfortunately this approach does require the use of some C++ casts, so some of the type safety that can be achieved when programming in C++ is lost.
Thread information

Name

cyg_thread_self, cyg_thread_idle_thread, cyg_thread_get_stack_base,
cyg_thread_get_stack_size, cyg_thread_measure_stack_usage,
cyg_thread_get_next, cyg_thread_get_info, cyg_thread_get_id,
cyg_thread_find — Get basic thread information

Synopsis

#include <cyg/kernel/kapi.h>

cyg_handle_t cyg_thread_self(void);
cyg_handle_t cyg_thread_idle_thread(void);
cyg_addrword_t cyg_thread_get_stack_base(cyg_handle_t thread);
cyg_uint32 cyg_thread_get_stack_size(cyg_handle_t thread);
cyg_uint32 cyg_thread_measure_stack_usage(cyg_handle_t thread);
cyg_bool cyg_thread_get_next(cyg_handle_t *thread, cyg_uint16 *id);
cyg_bool cyg_thread_get_info(cyg_handle_t thread, cyg_uint16 id, cyg_thread_info *info);
cyg_uint16 cyg_thread_get_id(cyg_handle_t thread);
cyg_handle_t cyg_thread_find(cyg_uint16 id);

Description

These functions can be used to obtain some basic information about various threads in the system. Typically
they serve little or no purpose in real applications, but they can be useful during debugging.

cyg_thread_self returns a handle corresponding to the current thread. It will be the same as the value filled in
by cyg_thread_create when the current thread was created. This handle can then be passed to other functions
such as cyg_thread_get_priority.

cyg_thread_idle_thread returns the handle corresponding to the idle thread. This thread is created auto-
matically by the kernel, so application-code has no other way of getting hold of this information.

cyg_thread_get_stack_base and cyg_thread_get_stack_size return information about a specific
thread’s stack. The values returned will match the values passed to cyg_thread_create when this thread was
created.

cyg_thread_measure_stack_usage is only available if the configuration option
CYGFUN_KERNEL_THREADS_STACK_MEASUREMENT is enabled. The return value is the maximum number of
bytes of stack space used so far by the specified thread. Note that this should not be considered a true upper
bound, for example it is possible that in the current test run the specified thread has not yet been interrupted at
the deepest point in the function call graph. Never the less the value returned can give some useful indication
of the thread’s stack requirements.

cyg_thread_get_next is used to enumerate all the current threads in the system. It should be called initially
with the locations pointed to by thread and id set to zero. On return these will be set to the handle and ID of
the first thread. On subsequent calls, these parameters should be left set to the values returned by the previous
call. The handle and ID of the next thread in the system will be installed each time, until a false return value
indicates the end of the list.
Thread information

cyg_thread_get_info fills in the cyg_thread_info structure with information about the thread described by the thread and id arguments. The information returned includes the thread’s handle and id, its state and name, priorities and stack parameters. If the thread does not exist the function returns false.

The cyg_thread_info structure is defined as follows by <cyg/kernel/kapi.h>, but may be extended in future with additional members, and so its size should not be relied upon:

typedef struct
{
    cyg_handle_t    handle;
    cyg_uint16      id;
    cyg_uint32      state;
    char*           name;
    cyg_priority_t  set_pri;
    cyg_priority_t  cur_pri;
    cyg_addrword_t  stack_base;
    cyg_uint32      stack_size;
    cyg_uint32      stack_used;
} cyg_thread_info;

cyg_thread_get_id returns the unique thread ID for the thread identified by thread.

cyg_thread_find returns a handle for the thread whose ID is id. If no such thread exists, a zero handle is returned.

Valid contexts

cyg_thread_self may only be called from thread context. cyg_thread_idle_thread may be called from thread or DSR context, but only after the system has been initialized. cyg_thread_get_stack_base, cyg_thread_get_stack_size and cyg_thread_measure_stack_usage may be called any time after the specified thread has been created, but measuring stack usage involves looping over at least part of the thread’s stack so this should normally only be done from thread context. cyg_thread_get_id may be called from any context as long as the caller can guarantee that the supplied thread handle remains valid.

Examples

A simple example of the use of the cyg_thread_get_next and cyg_thread_get_info follows:

#include <cyg/kernel/kapi.h>
#include <stdio.h>

void show_threads(void)
{
    cyg_handle_t thread = 0;
    cyg_uint16 id = 0;

    while( cyg_thread_get_next( &thread, &id ) )
    {
        cyg_thread_info info;

        if( !cyg_thread_get_info( thread, id, &info ) )
            break;

        printf("ID: %04x name: %10s pri: %d\n",
            thread, id, info.state, info.set_pri, info.cur_pri, info.stack_base, info.stack_size, info.stack_used,
        );
    }
}
Thread information

info.id, info.name?info.name:"----", info.set_pri );
Thread information
Thread control

Name

cyg_thread_yield, cyg_thread_delay, cyg_thread_suspend,
cyg_thread_resume, cyg_thread_release — Control whether or not a thread is running

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_thread_yield (void);
void cyg_thread_delay (cyg_tick_count_t delay);
void cyg_thread_suspend (cyg_handle_t thread);
void cyg_thread_resume (cyg_handle_t thread);
void cyg_thread_release (cyg_handle_t thread);

Description

These functions provide some control over whether or not a particular thread can run. Apart from the required use of cyg_thread_resume to start a newly-created thread, application code should normally use proper synchronization primitives such as condition variables or mail boxes.

Yield

cyg_thread_yield allows a thread to relinquish control of the processor to some other runnable thread which has the same priority. This can have no effect on any higher-priority thread since, if such a thread were runnable, the current thread would have been preempted in its favour. Similarly it can have no effect on any lower-priority thread because the current thread will always be run in preference to those. As a consequence this function is only useful in configurations with a scheduler that allows multiple threads to run at the same priority, for example the mlqueue scheduler. If instead the bitmap scheduler was being used then cyg_thread_yield() would serve no purpose.

Even if a suitable scheduler such as the mlqueue scheduler has been configured, cyg_thread_yield will still rarely prove useful: instead timeslicing will be used to ensure that all threads of a given priority get a fair slice of the available processor time. However it is possible to disable timeslicing via the configuration option CYGSEM_KERNEL_SCHED_TIMESLICE, in which case cyg_thread_yield can be used to implement a form of cooperative multitasking.

Delay

cyg_thread_delay allows a thread to suspend until the specified number of clock ticks have occurred. For example, if a value of 1 is used and the system clock runs at a frequency of 100Hz then the thread will sleep for up to 10 milliseconds. This functionality depends on the presence of a real-time system clock, as controlled by the configuration option CYGVAR_KERNEL_COUNTERS_CLOCK.
Thread control

If the application requires delays measured in milliseconds or similar units rather than in clock ticks, some calculations are needed to convert between these units as described in Clocks. Usually these calculations can be done by the application developer, or at compile-time. Performing such calculations prior to every call to cyg_thread_delay adds unnecessary overhead to the system.

Suspend and Resume

Associated with each thread is a suspend counter. When a thread is first created this counter is initialized to 1. cyg_thread_suspend can be used to increment the suspend counter, and cyg_thread_resume decrements it. The scheduler will never run a thread with a non-zero suspend counter. Therefore a newly created thread will not run until it has been resumed.

An occasional problem with the use of suspend and resume functionality is that a thread gets suspended more times than it is resumed and hence never becomes runnable again. This can lead to very confusing behaviour. To help with debugging such problems the kernel provides a configuration option CYGNUM_KERNEL_MAX_SUSPEND_COUNT_ASSERT which imposes an upper bound on the number of suspend calls without matching resumes, with a reasonable default value. This functionality depends on infrastructure assertions being enabled.

Releasing a Blocked Thread

When a thread is blocked on a synchronization primitive such as a semaphore or a mutex, or when it is waiting for an alarm to trigger, it can be forcibly woken up using cyg_thread_release. Typically this will call the affected synchronization primitive to return false, indicating that the operation was not completed successfully. This function has to be used with great care, and in particular it should only be used on threads that have been designed appropriately and check all return codes. If instead it were to be used on, say, an arbitrary thread that is attempting to claim a mutex then that thread might not bother to check the result of the mutex lock operation - usually there would be no reason to do so. Therefore the thread will now continue running in the false belief that it has successfully claimed a mutex lock, and the resulting behaviour is undefined. If the system has been built with assertions enabled then it is possible that an assertion will trigger when the thread tries to release the mutex it does not actually own.

The main use of cyg_thread_release is in the POSIX compatibility layer, where it is used in the implementation of per-thread signals and cancellation handlers.

Valid contexts

cyg_thread_yield can only be called from thread context. A DSR must always run to completion and cannot yield the processor to some thread. cyg_thread_suspend, cyg_thread_resume, and cyg_thread_release may be called from thread or DSR context.
Thread termination

Name
cyg_thread_exit, cyg_thread_kill, cyg_thread_delete — Allow threads to terminate

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_thread_exit(void);
void cyg_thread_kill(cyg_handle_t thread);
cyg_bool_t cyg_thread_delete(cyg_handle_t thread);

Description

In many embedded systems the various threads are allocated statically, created during initialization, and never need to terminate. This avoids any need for dynamic memory allocation or other resource management facilities. However if a given application does have a requirement that some threads be created dynamically, must terminate, and their resources such as the stack be reclaimed, then the kernel provides the functions cyg_thread_exit, cyg_thread_kill, and cyg_thread_delete.

cyg_thread_exit allows a thread to terminate itself, thus ensuring that it will not be run again by the scheduler. However the cyg_thread data structure passed to cyg_thread_create remains in use, and the handle returned by cyg_thread_create remains valid. This allows other threads to perform certain operations on the terminated thread, for example to determine its stack usage via cyg_thread_measure_stack_usage. When the handle and cyg_thread structure are no longer required, cyg_thread_delete should be called to release these resources. If the stack was dynamically allocated then this should not be freed until after the call to cyg_thread_delete.

Alternatively, one thread may use cyg_thread_kill on another. This has much the same effect as the affected thread calling cyg_thread_exit. However killing a thread is generally rather dangerous because no attempt is made to unlock any synchronization primitives currently owned by that thread or release any other resources that thread may have claimed. Therefore use of this function should be avoided, and cyg_thread_exit is preferred. cyg_thread_kill cannot be used by a thread to kill itself.

cyg_thread_delete should be used on a thread after it has exited and is no longer required. After this call the thread handle is no longer valid, and both the cyg_thread structure and the thread stack can be re-used or freed. If cyg_thread_delete is invoked on a thread that is still running then there is an implicit call to cyg_thread_kill. This function returns true if the delete was successful, and false if the delete did not happen. The delete may not happen for example if the thread being destroyed is a lower priority thread than the running thread, and will thus not wake up in order to exit until it is rescheduled.

Valid contexts
cyg_thread_exit, cyg_thread_kill and cyg_thread_delete can only be called from thread context.
Thread termination
Thread priorities

Name

cyg_thread_get_priority, cyg_thread_get_current_priority, cyg_thread_set_priority — Examine and manipulate thread priorities

Synopsis

#include <cyg/kernel/kapi.h>

cyg_priority_t cyg_thread_get_priority (cyg_handle_t thread);
cyg_priority_t cyg_thread_get_current_priority (cyg_handle_t thread);
void cyg_thread_set_priority (cyg_handle_t thread, cyg_priority_t priority);

Description

Typical schedulers use the concept of a thread priority to determine which thread should run next. Exactly what this priority consists of will depend on the scheduler, but a typical implementation would be a small integer in the range 0 to 31, with 0 being the highest priority. Usually only the idle thread will run at the lowest priority. The exact number of priority levels available depends on the configuration, typically the option CYGNUM_KERNEL_SCHED_PRIORITIES.

cyg_thread_get_priority can be used to determine the priority of a thread, or more correctly the value last used in a cyg_thread_set_priority call or when the thread was first created. In some circumstances it is possible that the thread is actually running at a higher priority. For example, if it owns a mutex and priority ceilings or inheritance is being used to prevent priority inversion problems, then the thread’s priority may have been boosted temporarily. cyg_thread_get_current_priority returns the real current priority.

In many applications appropriate thread priorities can be determined and allocated statically. However, if it is necessary for a thread’s priority to change at run-time then the cyg_thread_set_priority function provides this functionality.

Valid contexts

cyg_thread_get_priority and cyg_thread_get_current_priority can be called from thread or DSR context, although the latter is rarely useful. cyg_thread_set_priority should also only be called from thread context.
Per-thread data

Name

cyg_thread_new_data_index, cyg_thread_free_data_index,
cyg_thread_get_data, cyg_thread_get_data_ptr, cyg_thread_set_data—
Manipulate per-thread data

Synopsis

#include <cyg/kernel/kapi.h>

cyg_ucount32 cyg_thread_new_data_index(void);
void cyg_thread_free_data_index(cyg_ucount32 index);
cyg_addrword_t cyg_thread_get_data(cyg_ucount32 index);
cyg_addrword_t* cyg_thread_get_data_ptr(cyg_ucount32 index);
void cyg_thread_set_data(cyg_ucount32 index, cyg_addrword_t data);

Description

In some applications and libraries it is useful to have some data that is specific to each thread. For example,
many of the functions in the POSIX compatibility package return -1 to indicate an error and store additional
information in what appears to be a global variable errno. However, if multiple threads make concurrent calls
into the POSIX library and if errno were really a global variable then a thread would have no way of knowing
whether the current errno value really corresponded to the last POSIX call it made, or whether some other
thread had run in the meantime and made a different POSIX call which updated the variable. To avoid such
confusion errno is instead implemented as a per-thread variable, and each thread has its own instance.
The support for per-thread data can be disabled via the configuration option CYGVAR_KERNEL_THREADS_DATA.
If enabled, each cyg_thread data structure holds a small array of words. The size of this array is determined
by the configuration option CYGNUM_KERNEL_THREADS_DATA_MAX. When a thread is created the array is filled
with zeroes.

If an application needs to use per-thread data then it needs an index into this array which has not yet been
allocated to other code. This index can be obtained by calling cyg_thread_new_data_index, and then used
in subsequent calls to cyg_thread_get_data. Typically indices are allocated during system initialization and
stored in static variables. If for some reason a slot in the array is no longer required and can be re-used then it
can be released by calling cyg_thread_free_data_index.

The current per-thread data in a given slot can be obtained using cyg_thread_get_data. This
implicitly operates on the current thread, and its single argument should be an index as returned by
cyg_thread_new_data_index. The per-thread data can be updated using cyg_thread_set_data. If a
particular item of per-thread data is needed repeatedly then cyg_thread_get_data_ptr can be used to
obtain the address of the data, and indirecing through this pointer allows the data to be examined and updated
efficiently.

Some packages, for example the error and POSIX packages, have pre-allocated slots in the array of per-thread
data. These slots should not normally be used by application code, and instead slots should be allocated during
initialization by a call to cyg_thread_new_data_index. If it is known that, for example, the configuration
will never include the POSIX compatibility package then application code may instead decide to re-use the
Per-thread data

slot allocated to that package, CYGNUM_KERNEL_THREADS_DATA_POSIX, but obviously this does involve a risk of strange and subtle bugs if the application’s requirements ever change.

Valid contexts

Typically cyg_thread_new_data_index is only called during initialization, but may also be called at any time in thread context. cyg_thread_free_data_index, if used at all, can also be called during initialization or from thread context. cyg_thread_get_data, cyg_thread_get_data_ptr, and cyg_thread_set_data may only be called from thread context because they implicitly operate on the current thread.
Thread destructors

Name

cyg_thread_add_destructor, cyg_thread_rem_destructor — Call functions on thread termination

Synopsis

```c
#include <cyg/kernel/kapi.h>

typedef void (*cyg_thread_destructor_fn)(cyg_addrword_t);

cyg_bool_t cyg_thread_add_destructor(cyg_thread_destructor_fn fn, cyg_addrword_t data);
cyg_bool_t cyg_thread_rem_destructor(cyg_thread_destructor_fn fn, cyg_addrword_t data);
```

Description

These functions are provided for cases when an application requires a function to be automatically called when a thread exits. This is often useful when, for example, freeing up resources allocated by the thread.

This support must be enabled with the configuration option `CYGPKG_KERNEL_THREADS_DESTRUCTORS`. When enabled, you may register a function of type `cyg_thread_destructor_fn` to be called on thread termination using `cyg_thread_add_destructor`. You may also provide it with a piece of arbitrary information in the `data` argument which will be passed to the destructor function `fn` when the thread terminates. If you no longer wish to call a function previously registered with `cyg_thread_add_destructor`, you may call `cyg_thread_rem_destructor` with the same parameters used to register the destructor function. Both these functions return `true` on success and `false` on failure.

By default, thread destructors are per-thread, which means that registering a destructor function only registers that function for the current thread. In other words, each thread has its own list of destructors. Alternatively you may disable the configuration option `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` in which case any registered destructors will be run when any threads exit. In other words, the thread destructor list is global and all threads have the same destructors.

There is a limit to the number of destructors which may be registered, which can be controlled with the `CYGNUM_KERNEL_THREADS_DESTRUCTORS` configuration option. Increasing this value will very slightly increase the amount of memory in use, and when `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is enabled, the amount of memory used per thread will increase. When the limit has been reached, `cyg_thread_add_destructor` will return `false`.

Valid contexts

When `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is enabled, these functions must only be called from a thread context as they implicitly operate on the current thread. When `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is disabled, these functions may be called from thread or DSR context, or at initialization time.
Thread destructors
Exception handling

**Name**

cyg_exception_set_handler, cyg_exception_clear_handler,
cyg_exception_call_handler — Handle processor exceptions

**Synopsis**

```c
#include <cyg/kernel/kapi.h>

void cyg_exception_set_handler(cyg_code_t exception_number, cyg_exception_handler_t* new_handler, cyg_addrword_t new_data, cyg_exception_handler_t** old_handler, cyg_addrword_t* old_data);
void cyg_exception_clear_handler(cyg_code_t exception_number);
void cyg_exception_call_handler(cyg_handle_t thread, cyg_code_t exception_number, cyg_addrword_t exception_info);
```

**Description**

Sometimes code attempts operations that are not legal on the current hardware, for example dividing by zero, or accessing data through a pointer that is not properly aligned. When this happens the hardware will raise an exception. This is very similar to an interrupt, but happens synchronously with code execution rather than asynchronously and hence can be tied to the thread that is currently running.

The exceptions that can be raised depend very much on the hardware, especially the processor. The corresponding documentation should be consulted for more details. Alternatively the architectural HAL header file hal_intr.h, or one of the variant or platform header files it includes, will contain appropriate definitions. The details of how to handle exceptions, including whether or not it is possible to recover from them, also depend on the hardware.

Exception handling is optional, and can be disabled through the configuration option CYGPKG_KERNEL_EXCEPTIONS. If an application has been exhaustively tested and is trusted never to raise a hardware exception then this option can be disabled and code and data sizes will be reduced somewhat. If exceptions are left enabled then the system will provide default handlers for the various exceptions, but these do nothing. Even the specific type of exception is ignored, so there is no point in attempting to decode this and distinguish between say a divide-by-zero and an unaligned access. If the application installs its own handlers and wants details of the specific exception being raised then the configuration option CYGSEM_KERNEL_EXCEPTIONS_DECODE has to be enabled.

An alternative handler can be installed using cyg_exception_set_handler. This requires a code for the exception, a function pointer for the new exception handler, and a parameter to be passed to this handler. Details of the previously installed exception handler will be returned via the remaining two arguments, allowing that handler to be reinstated, or null pointers can be used if this information is of no interest. An exception handling function should take the following form:

```c
void
my_exception_handler(cyg_addrword_t data, cyg_code_t exception, cyg_addrword_t info)
{
    ...
}
```
Exception handling

The data argument corresponds to the `new_data` parameter supplied to `cyg_exception_set_handler`. The exception code is provided as well, in case a single handler is expected to support multiple exceptions. The `info` argument will depend on the hardware and on the specific exception.

`cyg_exception_clear_handler` can be used to restore the default handler, if desired. It is also possible for software to raise an exception and cause the current handler to be invoked, but generally this is useful only for testing.

By default the system maintains a single set of global exception handlers. However, since exceptions occur synchronously it is sometimes useful to handle them on a per-thread basis, and have a different set of handlers for each thread. This behaviour can be obtained by disabling the configuration option `CYGSEM_KERNEL_EXCEPTIONS_GLOBAL`. If per-thread exception handlers are being used then `cyg_exception_set_handler` and `cyg_exception_clear_handler` apply to the current thread. Otherwise they apply to the global set of handlers.

---

**Caution**

In the current implementation `cyg_exception_call_handler` can only be used on the current thread. There is no support for delivering an exception to another thread.

---

**Note:** Exceptions at the eCos kernel level refer specifically to hardware-related events such as unaligned accesses to memory or division by zero. There is no relation with other concepts that are also known as exceptions, for example the `throw` and `catch` facilities associated with C++.

---

**Valid contexts**

If the system is configured with a single set of global exception handlers then `cyg_exception_set_handler` and `cyg_exception_clear_handler` may be called during initialization or from thread context. If instead per-thread exception handlers are being used then it is not possible to install new handlers during initialization because the functions operate implicitly on the current thread, so they can only be called from thread context. `cyg_exception_call_handler` should only be called from thread context.
Counters

Name

cyg_counter_create, cyg_counter_delete, cyg_counter_current_value,
cyg_counter_set_value, cyg_counter_tick — Count event occurrences

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_counter_create(cyg_handle_t* handle, cyg_counter* counter);
void cyg_counter_delete(cyg_handle_t counter);
cyg_tick_count_t cyg_counter_current_value(cyg_handle_t counter);
void cyg_counter_set_value(cyg_handle_t counter, cyg_tick_count_t new_value);
void cyg_counter_tick(cyg_handle_t counter);

Description

Kernel counters can be used to keep track of how many times a particular event has occurred. Usually this event is an external signal of some sort. The most common use of counters is in the implementation of clocks, but they can be useful with other event sources as well. Application code can attach alarms to counters, causing a function to be called when some number of events have occurred.

A new counter is initialized by a call to cyg_counter_create. The first argument is used to return a handle to the new counter which can be used for subsequent operations. The second argument allows the application to provide the memory needed for the object, thus eliminating any need for dynamic memory allocation within the kernel. If a counter is no longer required and does not have any alarms attached then cyg_counter_delete can be used to release the resources, allowing the cyg_counter data structure to be re-used.

Initializing a counter does not automatically attach it to any source of events. Instead some other code needs to call cyg_counter_tick whenever a suitable event occurs, which will cause the counter to be incremented and may cause alarms to trigger. The current value associated with the counter can be retrieved using cyg_counter_current_value and modified with cyg_counter_set_value. Typically the latter function is only used during initialization, for example to set a clock to wallclock time, but it can be used to reset a counter if necessary. However cyg_counter_set_value will never trigger any alarms. A newly initialized counter has a starting value of 0.

The kernel provides two different implementations of counters. The default is CYGIMP_KERNEL_COUNTERS_SINGLE_LIST which stores all alarms attached to the counter on a single list. This is simple and usually efficient. However when a tick occurs the kernel code has to traverse this list, typically at DSR level, so if there are a significant number of alarms attached to a single counter this will affect the system’s dispatch latency. The alternative implementation, CYGIMP_KERNEL_COUNTERS_MULTI_LIST, stores each alarm in one of an array of lists such that at most one of the lists needs to be searched per clock tick. This involves extra code and data, but can improve real-time responsiveness in some circumstances. Another configuration option that is relevant here is CYGIMP_KERNEL_COUNTERS_SORT_LIST, which is disabled by default. This provides a trade off between doing work whenever a new alarm is added to a counter and doing work whenever a tick occurs. It is application-dependent which of these is more appropriate.
Counters

Valid contexts

cyg_counter_create is typically called during system initialization but may also be called in thread context. Similarly cyg_counter_delete may be called during initialization or in thread context. cyg_counter_current_value, cyg_counter_set_value and cyg_counter_tick may be called during initialization or from thread or DSR context. In fact, cyg_counter_tick is usually called from inside a DSR in response to an external event of some sort.
Clocks

Name

cyg_clock_create, cyg_clock_delete, cyg_clock_to_counter,
cyg_clock_set_resolution, cyg_clock_get_resolution, cyg_real_time_clock,
cyg_current_time — Provide system clocks

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_clock_create(cyg_resolution_t resolution, cyg_handle_t* handle, cyg_clock* clock);
void cyg_clock_delete(cyg_handle_t clock);
void cyg_clock_to_counter(cyg_handle_t clock, cyg_handle_t* counter);
void cyg_clock_set_resolution(cyg_handle_t clock, cyg_resolution_t resolution);
cyg_resolution_t cyg_clock_get_resolution(cyg_handle_t clock);
cyg_handle_t cyg_real_time_clock(void);
cyg_tick_count_t cyg_current_time(void);

Description

In the eCos kernel clock objects are a special form of counter objects. They are attached to a specific type of hardware, clocks that generate ticks at very specific time intervals, whereas counters can be used with any event source.

In a default configuration the kernel provides a single clock instance, the real-time clock. This gets used for timeslicing and for operations that involve a timeout, for example cyg_semaphore_timed_wait. If this functionality is not required it can be removed from the system using the configuration option CYGVAR_KERNEL_COUNTERS_CLOCK. Otherwise the real-time clock can be accessed by a call to cyg_real_time_clock, allowing applications to attach alarms, and the current counter value can be obtained using cyg_current_time.

Applications can create and destroy additional clocks if desired, using cyg_clock_create and cyg_clock_delete. The first argument to cyg_clock_create specifies the resolution this clock will run at. The second argument is used to return a handle for this clock object, and the third argument provides the kernel with the memory needed to hold this object. This clock will not actually tick by itself. Instead it is the responsibility of application code to initialize a suitable hardware timer to generate interrupts at the appropriate frequency, install an interrupt handler for this, and call cyg_counter_tick from inside the DSR. Associated with each clock is a kernel counter, a handle for which can be obtained using cyg_clock_to_counter.

Clock Resolutions and Ticks

At the kernel level all clock-related operations including delays, timeouts and alarms work in units of clock ticks, rather than in units of seconds or milliseconds. If the calling code, whether the application or some other package, needs to operate using units such as milliseconds then it has to convert from these units to clock ticks.
The main reason for this is that it accurately reflects the hardware: calling something like \texttt{nanosleep} with a delay of ten nanoseconds will not work as intended on any real hardware because timer interrupts simply will not happen that frequently; instead calling \texttt{cyg_thread_delay} with the equivalent delay of 0 ticks gives a much clearer indication that the application is attempting something inappropriate for the target hardware. Similarly, passing a delay of five ticks to \texttt{cyg_thread_delay} makes it fairly obvious that the current thread will be suspended for somewhere between four and five clock periods, as opposed to passing \texttt{50000000} to \texttt{nanosleep} which suggests a granularity that is not actually provided.

A secondary reason is that conversion between clock ticks and units such as milliseconds can be somewhat expensive, and whenever possible should be done at compile-time or by the application developer rather than at run-time. This saves code size and cpu cycles.

The information needed to perform these conversions is the clock resolution. This is a structure with two fields, a dividend and a divisor, and specifies the number of nanoseconds between clock ticks. For example a clock that runs at 100Hz will have 10 milliseconds between clock ticks, or 10000000 nanoseconds. The ratio between the resolution’s dividend and divisor will therefore be \(10000000\) to \(1\), and typical values for these might be \(1000000000\) and \(100\). If the clock runs at a different frequency, say 60Hz, the numbers could be \(1000000000\) and \(60\) respectively. Given a delay in nanoseconds, this can be converted to clock ticks by multiplying with the the divisor and then dividing by the dividend. For example a delay of 50 milliseconds corresponds to 50000000 nanoseconds, and with a clock frequency of 100Hz this can be converted to \((50000000 \times 100) / 1000000000\) = 5 clock ticks. Given the large numbers involved this arithmetic normally has to be done using 64-bit precision and the long long data type, but allows code to run on hardware with unusual clock frequencies.

The default frequency for the real-time clock on any platform is usually about 100Hz, but platform-specific documentation should be consulted for this information. Usually it is possible to override this default by configuration options, but again this depends on the capabilities of the underlying hardware. The resolution for any clock can be obtained using \texttt{cyg_clock_get_resolution}. For clocks created by application code, there is also a function \texttt{cyg_clock_set_resolution}. This does not affect the underlying hardware timer in any way, it merely updates the information that will be returned in subsequent calls to \texttt{cyg_clock_get_resolution}; changing the actual underlying clock frequency will require appropriate manipulation of the timer hardware.

**Valid contexts**

\texttt{cyg_clock_create} is usually only called during system initialization (if at all), but may also be called from thread context. The same applies to \texttt{cyg_clock_delete}. The remaining functions may be called during initialization, from thread context, or from DSR context, although it should be noted that there is no locking between \texttt{cyg_clock_get_resolution} and \texttt{cyg_clock_set_resolution} so theoretically it is possible that the former returns an inconsistent data structure.
Alarms

Name

cyg_alarm_create, cyg_alarm_delete, cyg_alarm_initialize,
cyg_alarm_enable, cyg_alarm_disable — Run an alarm function when a number of events have occurred

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_alarm_create(cyg_handle_t counter, cyg_alarm_t* alarmfn, cyg_addrword_t data,
                      cyg_handle_t* handle, cyg_alarm* alarm);
void cyg_alarm_delete(cyg_handle_t alarm);
void cyg_alarm_initialize(cyg_handle_t alarm, cyg_tick_count_t trigger,
                          cyg_tick_count_t interval);
void cyg_alarm_enable(cyg_handle_t alarm);
void cyg_alarm_disable(cyg_handle_t alarm);

Description

Kernel alarms are used together with counters and allow for action to be taken when a certain number of events have occurred. If the counter is associated with a clock then the alarm action happens when the appropriate number of clock ticks have occurred, in other words after a certain period of time.

Setting up an alarm involves a two-step process. First the alarm must be created with a call to cyg_alarm_create. This takes five arguments. The first identifies the counter to which the alarm should be attached. If the alarm should be attached to the system’s real-time clock then cyg_real_time_clock and cyg_clock_to_counter can be used to get hold of the appropriate handle. The next two arguments specify the action to be taken when the alarm is triggered, in the form of a function pointer and some data. This function should take the form:

void
amalarm_handler(cyg_handle_t alarm, cyg_addrword_t data)
{
    ...
}

The data argument passed to the alarm function corresponds to the third argument passed to cyg_alarm_create. The fourth argument to cyg_alarm_create is used to return a handle to the newly-created alarm object, and the final argument provides the memory needed for the alarm object and thus avoids any need for dynamic memory allocation within the kernel.

Once an alarm has been created a further call to cyg_alarm_initialize is needed to activate it. The first argument specifies the alarm. The second argument indicates the number of events, for example clock ticks, that need to occur before the alarm triggers. If the third argument is 0 then the alarm will only trigger once. A non-zero value specifies that the alarm should trigger repeatedly, with an interval of the specified number of events.
Alarms can be temporarily disabled and reenabled using `cyg_alarm_disable` and `cyg_alarm_enable`. Alternatively another call to `cyg_alarm_initialize` can be used to modify the behaviour of an existing alarm. If an alarm is no longer required then the associated resources can be released using `cyg_alarm_delete`.

The alarm function is invoked when a counter tick occurs, in other words when there is a call to `cyg_counter_tick`, and will happen in the same context. If the alarm is associated with the system’s real-time clock then this will be DSR context, following a clock interrupt. If the alarm is associated with some other application-specific counter then the details will depend on how that counter is updated.

If two or more alarms are registered for precisely the same counter tick, the order of execution of the alarm functions is unspecified.

**Valid contexts**

`cyg_alarm_create` `cyg_alarm_initialize` is typically called during system initialization but may also be called in thread context. The same applies to `cyg_alarm_delete`. `cyg_alarm_initialize`, `cyg_alarm_disable` and `cyg_alarm_enable` may be called during initialization or from thread or DSR context, but `cyg_alarm_enable` and `cyg_alarm_initialize` may be expensive operations and should only be called when necessary.
Mutexes

Name

cyg_mutex_init, cyg_mutex_destroy, cyg_mutex_lock, cyg_mutex_trylock, cyg_mutex_unlock, cyg_mutex_release, cyg_mutex_set_ceiling, cyg_mutex_set_protocol — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_mutex_init(cyg_mutex_t* mutex);
void cyg_mutex_destroy(cyg_mutex_t* mutex);
cyg_bool_t cyg_mutex_lock(cyg_mutex_t* mutex);
cyg_bool_t cyg_mutex_trylock(cyg_mutex_t* mutex);
void cyg_mutex_unlock(cyg_mutex_t* mutex);
void cyg_mutex_release(cyg_mutex_t* mutex);
void cyg_mutex_set_ceiling(cyg_mutex_t* mutex, cyg_priority_t priority);
void cyg_mutex_set_protocol(cyg_mutex_t* mutex, enum cyg_mutex_protocol protocol/);

Description

The purpose of mutexes is to let threads share resources safely. If two or more threads attempt to manipulate a data structure with no locking between them then the system may run for quite some time without apparent problems, but sooner or later the data structure will become inconsistent and the application will start behaving strangely and is quite likely to crash. The same can apply even when manipulating a single variable or some other resource. For example, consider:

static volatile int counter = 0;

void
process_event(void)
{
    ...
    counter++;
}

Assume that after a certain period of time counter has a value of 42, and two threads A and B running at the same priority call process_event. Typically thread A will read the value of counter into a register, increment this register to 43, and write this updated value back to memory. Thread B will do the same, so usually counter will end up with a value of 44. However if thread A is timesliced after reading the old value 42 but before writing back 43, thread B will still read back the old value and will also write back 43. The net result is that the counter only gets incremented once, not twice, which depending on the application may prove disastrous.

Sections of code like the above which involve manipulating shared data are generally known as critical regions. Code should claim a lock before entering a critical region and release the lock when leaving. Mutexes provide an appropriate synchronization primitive for this.
A mutex must be initialized before it can be used, by calling `cyg_mutex_init`. This takes a pointer to a `cyg_mutex_t` data structure which is typically statically allocated, and may be part of a larger data structure. If a mutex is no longer required and there are no threads waiting on it then `cyg_mutex_destroy` can be used.

The main functions for using a mutex are `cyg_mutex_lock` and `cyg_mutex_unlock`. In normal operation `cyg_mutex_lock` will return success after claiming the mutex lock, blocking if another thread currently owns the mutex. However the lock operation may fail if other code calls `cyg_mutex_release` or `cyg_thread_release`, so if these functions may get used then it is important to check the return value. The current owner of a mutex should call `cyg_mutex_unlock` when a lock is no longer required. This operation must be performed by the owner, not by another thread.

`cyg_mutex_trylock` is a variant of `cyg_mutex_lock` that will always return immediately, returning success or failure as appropriate. This function is rarely useful. Typical code locks a mutex just before entering a critical region, so if the lock cannot be claimed then there may be nothing else for the current thread to do. Use of this function may also cause a form of priority inversion if the owner runs at a lower priority, because the priority inheritance code will not be triggered. Instead the current thread continues running, preventing the owner from getting any cpu time, completing the critical region, and releasing the mutex.

`cyg_mutex_release` can be used to wake up all threads that are currently blocked inside a call to `cyg_mutex_lock` for a specific mutex. These lock calls will return failure. The current mutex owner is not affected.

**Priority Inversion**

The use of mutexes gives rise to a problem known as priority inversion. In a typical scenario this requires three threads A, B, and C, running at high, medium and low priority respectively. Thread A and thread B are temporarily blocked waiting for some event, so thread C gets a chance to run, needs to enter a critical region, and locks a mutex. At this point threads A and B are woken up - the exact order does not matter. Thread A needs to claim the same mutex but has to wait until C has left the critical region and can release the mutex. Meanwhile thread B works on something completely different and can continue running without problems. Because thread C is running a lower priority than B it will not get a chance to run until B blocks for some reason, and hence thread A cannot run either. The overall effect is that a high-priority thread A cannot proceed because of a lower priority thread B, and priority inversion has occurred.

In simple applications it may be possible to arrange the code such that priority inversion cannot occur, for example by ensuring that a given mutex is never shared by threads running at different priority levels. However this may not always be possible even at the application level. In addition mutexes may be used internally by underlying code, for example the memory allocation package, so careful analysis of the whole system would be needed to be sure that priority inversion cannot occur. Instead it is common practice to use one of two techniques: priority ceilings and priority inheritance.
Mutexes

Priority ceilings involve associating a priority with each mutex. Usually this will match the highest priority thread that will ever lock the mutex. When a thread running at a lower priority makes a successful call to `cyg_mutex_lock` or `cyg_mutex_trylock` its priority will be boosted to that of the mutex. For example, given the previous example the priority associated with the mutex would be that of thread A, so for as long as it owns the mutex thread C will run in preference to thread B. When C releases the mutex its priority drops to the normal value again, allowing A to run and claim the mutex. Setting the priority for a mutex involves a call to `cyg_mutex_set_ceiling`, which is typically called during initialization. It is possible to change the ceiling dynamically but this will only affect subsequent lock operations, not the current owner of the mutex.

Priority ceilings are very suitable for simple applications, where for every thread in the system it is possible to work out which mutexes will be accessed. For more complicated applications this may prove difficult, especially if thread priorities change at run-time. An additional problem occurs for any mutexes outside the application, for example used internally within eCos packages. A typical eCos package will be unaware of the details of the various threads in the system, so it will have no way of setting suitable ceilings for its internal mutexes. If those mutexes are not exported to application code then using priority ceilings may not be viable. The kernel does provide a configuration option `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY` that can be used to set the default priority ceiling for all mutexes, which may prove sufficient.

The alternative approach is to use priority inheritance: if a thread calls `cyg_mutex_lock` for a mutex that it currently owned by a lower-priority thread, then the owner will have its priority raised to that of the current thread. Often this is more efficient than priority ceilings because priority boosting only happens when necessary, not for every lock operation, and the required priority is determined at run-time rather than by static analysis. However there are complications when multiple threads running at different priorities try to lock a single mutex, or when the current owner of a mutex then tries to lock additional mutexes, and this makes the implementation significantly more complicated than priority ceilings.

There are a number of configuration options associated with priority inversion. First, if after careful analysis it is known that priority inversion cannot arise then the component `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL` can be disabled. More commonly this component will be enabled, and one of either `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT` or `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING` will be selected, so that one of the two protocols is available for all mutexes. It is possible to select multiple protocols, so that some mutexes can have priority ceilings while others use priority inheritance or no priority inversion protection at all. Obviously this flexibility will add to the code size and to the cost of mutex operations. The default for all mutexes is controlled by `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT`, and can be changed at run-time using `cyg_mutex_set_protocol`.

Priority inversion problems can also occur with other synchronization primitives such as semaphores. For example there could be a situation where a high-priority thread A is waiting on a semaphore, a low-priority thread B is running and preventing C from making progress. However a semaphore does not have the concept of an owner, so there is no way for the system to know that it is thread C which would next post to the semaphore. Hence there is no way for the system to boost the priority of C automatically and prevent the priority inversion. Instead situations like this have to be detected by application developers and appropriate precautions have to be taken, for example making sure that all the threads run at suitable priorities at all times.
Mutexes

Warning
The current implementation of priority inheritance within the eCos kernel does not handle certain exceptional circumstances completely correctly. Problems will only arise if a thread owns one mutex, then attempts to claim another mutex, and there are other threads attempting to lock these same mutexes. Although the system will continue running, the current owners of the various mutexes involved may not run at the priority they should. This situation never arises in typical code because a mutex will only be locked for a small critical region, and there is no need to manipulate other shared resources inside this region. A more complicated implementation of priority inheritance is possible but would add significant overhead and certain operations would no longer be deterministic.

Warning
Support for priority ceilings and priority inheritance is not implemented for all schedulers. In particular neither priority ceilings nor priority inheritance are currently available for the bitmap scheduler.

Alternatives
In nearly all circumstances, if two or more threads need to share some data then protecting this data with a mutex is the correct thing to do. Mutexes are the only primitive that combine a locking mechanism and protection against priority inversion problems. However this functionality is achieved at a cost, and in exceptional circumstances such as an application’s most critical inner loop it may be desirable to use some other means of locking.

When a critical region is very very small it is possible to lock the scheduler, thus ensuring that no other thread can run until the scheduler is unlocked again. This is achieved with calls to cyg_scheduler_lock and cyg_scheduler_unlock. If the critical region is sufficiently small then this can actually improve both performance and dispatch latency because cyg_mutex_lock also locks the scheduler for a brief period of time. This approach will not work on SMP systems because another thread may already be running on a different processor and accessing the critical region.

Another way of avoiding the use of mutexes is to make sure that all threads that access a particular critical region run at the same priority and configure the system with timeslicing disabled (CYGSEM_KERNEL_SCHED_TIMESLICE). Without timeslicing a thread can only be preempted by a higher-priority one, or if it performs some operation that can block. This approach requires that none of the operations in the critical region can block, so for example it is not legal to call cyg_semaphore_wait. It is also vulnerable to any changes in the configuration or to the various thread priorities: any such changes may now have unexpected side effects. It will not work on SMP systems.

Recursive Mutexes
The implementation of mutexes within the eCos kernel does not support recursive locks. If a thread has locked a mutex and then attempts to lock the mutex again, typically as a result of some recursive call in a complicated call graph, then either an assertion failure will be reported or the thread will deadlock. This behaviour is deliberate. When a thread has just locked a mutex associated with some data structure, it can assume that that data structure is in a consistent state. Before unlocking the mutex again it must ensure that the data structure is again in a consistent state. Recursive mutexes allow a thread to make arbitrary changes to a data structure, then in a recursive call lock the mutex again while the data structure is still inconsistent. The net result is that
Mutexes

code can no longer make any assumptions about data structure consistency, which defeats the purpose of using mutexes.

Valid contexts

cyg_mutex_init, cyg_mutex_set_ceiling and cyg_mutex_set_protocol are normally called during initialization but may also be called from thread context. The remaining functions should only be called from thread context. Mutexes serve as a mutual exclusion mechanism between threads, and cannot be used to synchronize between threads and the interrupt handling subsystem. If a critical region is shared between a thread and a DSR then it must be protected using cyg_scheduler_lock and cyg_scheduler_unlock. If a critical region is shared between a thread and an ISR, it must be protected by disabling or masking interrupts. Obviously these operations must be used with care because they can affect dispatch and interrupt latencies.
Mutexes
Condition Variables

Name

cyg_cond_init, cyg_cond_destroy, cyg_cond_wait, cyg_cond_timed_wait, cyg_cond_signal, cyg_cond_broadcast — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_cond_init(cyg_cond_t* cond, cyg_mutex_t* mutex);
void cyg_cond_destroy(cyg_cond_t* cond);
cyg_bool_t cyg_cond_wait(cyg_cond_t* cond);
cyg_bool_t cyg_cond_timed_wait(cyg_cond_t* cond, cyg_tick_count_t abstime);
void cyg_cond_signal(cyg_cond_t* cond);
void cyg_cond_broadcast(cyg_cond_t* cond);

Description

Condition variables are used in conjunction with mutexes to implement long-term waits for some condition to become true. For example consider a set of functions that control access to a pool of resources:

cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;

    cyg_mutex_lock(&res_lock); // lock the mutex

    if( res_count == 0 ) // check for free resource
        res = RES_NONE; // return RES_NONE if none
    else
    {
        res_count--;
        res = res_pool[res_count]; // allocate a resources
    }

    cyg_mutex_unlock(&res_lock); // unlock the mutex

    return res;
}
void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock); // lock the mutex
    res_pool[res_count] = res; // free the resource
    res_count++;
    cyg_mutex_unlock(&res_lock); // unlock the mutex
}

These routines use the variable res_count to keep track of the resources available. If there are none then
res_allocate returns RES_NONE, which the caller must check for and take appropriate error handling actions.
Now suppose that we do not want to return RES_NONE when there are no resources, but want to wait for one to
become available. This is where a condition variable can be used:

cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    cyg_cond_init(&res_wait, &res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock); // lock the mutex
    while( res_count == 0 ) // wait for a resources
        cyg_cond_wait(&res_wait);
    res_count--; // allocate a resource
    res = res_pool[res_count];
    cyg_mutex_unlock(&res_lock); // unlock the mutex
    return res;
}

void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock); // lock the mutex
    res_pool[res_count] = res; // free the resource
    res_count++;
    cyg_cond_signal(&res_wait); // wake up any waiting allocators
    cyg_mutex_unlock(&res_lock); // unlock the mutex
}
Condition Variables

In this version of the code, when `res_allocate` detects that there are no resources it calls `cyg_cond_wait`. This does two things: it unlocks the mutex, and puts the calling thread to sleep on the condition variable. When `res_free` is eventually called, it puts a resource back into the pool and calls `cyg_cond_signal` to wake up any thread waiting on the condition variable. When the waiting thread eventually gets to run again, it will re-lock the mutex before returning from `cyg_cond_wait`.

There are two important things to note about the way in which this code works. The first is that the mutex unlock and wait in `cyg_cond_wait` are atomic: no other thread can run between the unlock and the wait. If this were not the case then a call to `res_free` by that thread would release the resource but the call to `cyg_cond_signal` would be lost, and the first thread would end up waiting when there were resources available.

The second feature is that the call to `cyg_cond_wait` is in a `while` loop and not a simple `if` statement. This is because of the need to re-lock the mutex in `cyg_cond_wait` when the signalled thread reawakens. If there are other threads already queued to claim the lock then this thread must wait. Depending on the scheduler and the queue order, many other threads may have entered the critical section before this one gets to run. So the condition that it was waiting for may have been rendered false. Using a loop around all condition variable wait operations is the only way to guarantee that the condition being waited for is still true after waiting.

Before a condition variable can be used it must be initialized with a call to `cyg_cond_init`. This requires two arguments, memory for the data structure and a pointer to an existing mutex. This mutex will not be initialized by `cyg_cond_init`, instead a separate call to `cyg_mutex_init` is required. If a condition variable is no longer required and there are no threads waiting on it then `cyg_cond_destroy` can be used.

When a thread needs to wait for a condition to be satisfied it can call `cyg_cond_wait`. The thread must have already locked the mutex that was specified in the `cyg_cond_init` call. This mutex will be unlocked and the current thread will be suspended in an atomic operation. When some other thread performs a signal or broadcast operation the current thread will be woken up and automatically reclaim ownership of the mutex again, allowing it to examine global state and determine whether or not the condition is now satisfied.

The kernel supplies a variant of this function, `cyg_cond_timed_wait`, which can be used to wait on the condition variable or until some number of clock ticks have occurred. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the `cyg_current_time()` function should be added to determine the absolute number of ticks. The mutex will always be reclaimed before `cyg_cond_timed_wait` returns, regardless of whether it was a result of a signal operation or a timeout.

There is no `cyg_cond_trywait` function because this would not serve any purpose. If a thread has locked the mutex and determined that the condition is satisfied, it can just release the mutex and return. There is no need to perform any operation on the condition variable.

When a thread changes shared state that may affect some other thread blocked on a condition variable, it should call either `cyg_cond_signal` or `cyg_cond_broadcast`. These calls do not require ownership of the mutex, but usually the mutex will have been claimed before updating the shared state. A signal operation only wakes up the first thread that is waiting on the condition variable, while a broadcast wakes up all the threads. If there are no threads waiting on the condition variable at the time, then the signal or broadcast will have no effect: past signals are not counted up or remembered in any way. Typically a signal should be used when all threads will check the same condition and at most one thread can continue running. A broadcast should be used if threads check slightly different conditions, or if the change to the global state might allow multiple threads to proceed.

Valid contexts

cyg_cond_init is typically called during system initialization but may also be called in thread context. The same applies to cyg_cond_delete. cyg_cond_wait and cyg_cond_timedwait may only be called from thread context since they may block. cyg_cond_signal and cyg_cond_broadcast may be called from thread or DSR context.
Condition Variables
Semaphores

Name

cyg_semaphore_init, cyg_semaphore_destroy, cyg_semaphore_wait,
cyg_semaphore_timed_wait, cyg_semaphore_post, cyg_semaphore_peek —
Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_semaphore_init(cyg_sem_t* sem, cyg_count32 val);
void cyg_semaphore_destroy(cyg_sem_t* sem);
cyg_bool_t cyg_semaphore_wait(cyg_sem_t* sem);
cyg_bool_t cyg_semaphore_timed_wait(cyg_sem_t* sem, cyg_tick_count_t abstime);
cyg_bool_t cyg_semaphore_trywait(cyg_sem_t* sem);
void cyg_semaphore_post(cyg_sem_t* sem);
void cyg_semaphore_peek(cyg_sem_t* sem, cyg_count32* val);

Description

Counting semaphores are a synchronization primitive that allow threads to wait until an event has occurred. The event may be generated by a producer thread, or by a DSR in response to a hardware interrupt. Associated with each semaphore is an integer counter that keeps track of the number of events that have not yet been processed. If this counter is zero, an attempt by a consumer thread to wait on the semaphore will block until some other thread or a DSR posts a new event to the semaphore. If the counter is greater than zero then an attempt to wait on the semaphore will consume one event, in other words decrement the counter, and return immediately. Posting to a semaphore will wake up the first thread that is currently waiting, which will then resume inside the semaphore wait operation and decrement the counter again.

Another use of semaphores is for certain forms of resource management. The counter would correspond to how many of a certain type of resource are currently available, with threads waiting on the semaphore to claim a resource and posting to release the resource again. In practice condition variables are usually much better suited for operations like this.

cyg_semaphore_init is used to initialize a semaphore. It takes two arguments, a pointer to a cyg_sem_t structure and an initial value for the counter. Note that semaphore operations, unlike some other parts of the kernel API, use pointers to data structures rather than handles. This makes it easier to embed semaphores in a larger data structure. The initial counter value can be any number, zero, positive or negative, but typically a value of zero is used to indicate that no events have occurred yet.

cyg_semaphore_wait is used by a consumer thread to wait for an event. If the current counter is greater than 0, in other words if the event has already occurred in the past, then the counter will be decremented and the call will return immediately. Otherwise the current thread will be blocked until there is a cyg_semaphore_post call.

cyg_semaphore_post is called when an event has occurs. This increments the counter and wakes up the first thread waiting on the semaphore (if any). Usually that thread will then continue running inside cyg_semaphore_wait and decrement the counter again. However other scenarios are possible. For example the thread calling cyg_semaphore_post may be running at high priority, some other thread running at
Semaphores

Medium priority may be about to call \texttt{cyg_semaphore_wait} when it next gets a chance to run, and a low priority thread may be waiting on the semaphore. What will happen is that the current high priority thread continues running until it is descheduled for some reason, then the medium priority thread runs and its call to \texttt{cyg_semaphore_wait} succeeds immediately, and later on the low priority thread runs again, discovers a counter value of 0, and blocks until another event is posted. If there are multiple threads blocked on a semaphore then the configuration option \texttt{CYGIMP KERNEL SCHED_SORTED_QUEUES} determines which one will be woken up by a post operation.

\texttt{cyg_semaphore_wait} returns a boolean. Normally it will block until it has successfully decremented the counter, retrying as necessary, and return success. However the wait operation may be aborted by a call to \texttt{cyg_thread_release}, and \texttt{cyg_semaphore_wait} will then return false.

\texttt{cyg_semaphore_timed_wait} is a variant of \texttt{cyg_semaphore_wait}. It can be used to wait until either an event has occurred or a number of clock ticks have happened. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the \texttt{cyg_current_time()} function should be added to determine the absolute number of ticks. The function returns success if the semaphore wait operation succeeded, or false if the operation timed out or was aborted by \texttt{cyg_thread_release}. If support for the real-time clock has been removed from the current configuration then this function will not be available. \texttt{cyg_semaphore_trywait} is another variant which will always return immediately rather than block, again returning success or failure. If \texttt{cyg_semaphore_timedwait} is given a timeout in the past, it operates like \texttt{cyg_semaphore_trywait}.

\texttt{cyg_semaphore.peek} can be used to get hold of the current counter value. This function is rarely useful except for debugging purposes since the counter value may change at any time if some other thread or a DSR performs a semaphore operation.

Valid contexts

\texttt{cyg_semaphore_init} is normally called during initialization but may also be called from thread context. \texttt{cyg_semaphore_wait} and \texttt{cyg_semaphore_timed_wait} may only be called from thread context because these operations may block. \texttt{cyg_semaphore_trywait}, \texttt{cyg_semaphore_post} and \texttt{cyg_semaphore.peek} may be called from thread or DSR context.
Mail boxes

Name
cyg_mbox_create, cyg_mbox_delete, cyg_mbox_get, cyg_mbox_timed_get,
cyg_mbox_tryget, cyg_mbox_peek_item, cyg_mbox_put, cyg_mbox_timed_put,
cyg_mbox_tryput, cyg_mbox Peek, cyg_mbox_waiting_to_get,
cyg_mbox_waiting_to_put — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_mbox_create(cyg_handle_t* handle, cyg_mbox* mbox);
void cyg_mbox_delete(cyg_handle_t mbox);
void* cyg_mbox_get(cyg_handle_t mbox);
void* cyg_mbox_timed_get(cyg_handle_t mbox, cyg_tick_count_t abstime);
void cyg_mbox_tryget(cyg_handle_t mbox);
cyg_count32 cyg_mbox_peek(cyg_handle_t mbox);
void* cyg_mbox_peek_item(cyg_handle_t mbox);
cyg_bool_t cyg_mbox_put(cyg_handle_t mbox, void* item);
cyg_bool_t cyg_mbox_timed_put(cyg_handle_t mbox, void* item, cyg_tick_count_t abstime);
cyg_bool_t cyg_mbox_tryput(cyg_handle_t mbox, void* item);
cyg_bool_t cyg_mbox_waiting_to_get(cyg_handle_t mbox);
cyg_bool_t cyg_mbox_waiting_to_put(cyg_handle_t mbox);

Description

Mail boxes are a synchronization primitive. Like semaphores they can be used by a consumer thread to wait until a certain event has occurred, but the producer also has the ability to transmit some data along with each event. This data, the message, is normally a pointer to some data structure. It is stored in the mail box itself, so the producer thread that generates the event and provides the data usually does not have to block until some consumer thread is ready to receive the event. However a mail box will only have a finite capacity, typically ten slots. Even if the system is balanced and events are typically consumed at least as fast as they are generated, a burst of events can cause the mail box to fill up and the generating thread will block until space is available again. This behaviour is very different from semaphores, where it is only necessary to maintain a counter and hence an overflow is unlikely.

Before a mail box can be used it must be created with a call to cyg_mbox_create. Each mail box has a unique handle which will be returned via the first argument and which should be used for subsequent operations. cyg_mbox_create also requires an area of memory for the kernel structure, which is provided by the cyg_mbox second argument. If a mail box is no longer required then cyg_mbox_delete can be used. This will simply discard any messages that remain posted.

The main function for waiting on a mail box is cyg_mbox_get. If there is a pending message because of a call to cyg_mbox_put then cyg_mbox_get will return immediately with the message that was put into the mail box. Otherwise this function will block until there is a put operation. Exceptionally the thread can instead be unblocked by a call to cyg_thread_release, in which case cyg_mbox_get will return a null pointer. It is assumed that there will never be a call to cyg_mbox_put with a null pointer, because it would not be possible
Mail boxes
to distinguish between that and a release operation. Messages are always retrieved in the order in which they were put into the mail box, and there is no support for messages with different priorities.

There are two variants of cyg_mbox_get. The first, cyg_mbox_timed_get will wait until either a message is available or until a number of clock ticks have occurred. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. If no message is posted within the timeout then a null pointer will be returned. cyg_mbox_tryget is a non-blocking operation which will either return a message if one is available or a null pointer.

New messages are placed in the mail box by calling cyg_mbox_put or one of its variants. The main put function takes two arguments, a handle to the mail box and a pointer for the message itself. If there is a spare slot in the mail box then the new message can be placed there immediately, and if there is a waiting thread it will be woken up so that it can receive the message. If the mail box is currently full then cyg_mbox_put will block until there has been a get operation and a slot is available. The cyg_mbox_timed_put variant imposes a time limit on the put operation, returning false if the operation cannot be completed within the specified number of clock ticks and as for cyg_mbox_timed_get this is an absolute tick count. The cyg_mbox_tryput variant is non-blocking, returning false if there are no free slots available and the message cannot be posted without blocking.

There are a further four functions available for examining the current state of a mailbox. The results of these functions must be used with care because usually the state can change at any time as a result of activity within other threads, but they may prove occasionally useful during debugging or in special situations. cyg_mbox_peek returns a count of the number of messages currently stored in the mail box. cyg_mbox_peek_item retrieves the first message, but it remains in the mail box until a get operation is performed. cyg_mbox_waiting_to_get and cyg_mbox_waiting_to_put indicate whether or not there are currently threads blocked in a get or a put operation on a given mail box.

The number of slots in each mail box is controlled by a configuration option CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE, with a default value of 10. All mail boxes are the same size.

Valid contexts
cyg_mbox_create is typically called during system initialization but may also be called in thread context. The remaining functions are normally called only during thread context. Of special note is cyg_mbox_put which can be a blocking operation when the mail box is full, and which therefore must never be called from DSR context. It is permitted to call cyg_mbox_tryput, cyg_mbox_tryget, and the information functions from DSR context but this is rarely useful.
Event Flags

Name

cyg_flag_init, cyg_flag_destroy, cyg_flag_setbits, cyg_flag_maskbits,
cyg_flag_wait, cyg_flag_timed_wait, cyg_flag_poll, cyg_flag_peek,
cyg_flag_waiting — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_flag_init(cyg_flag_t* flag);
void cyg_flag_destroy(cyg_flag_t* flag);
void cyg_flag_setbits(cyg_flag_t* flag, cyg_flag_value_t value);
void cyg_flag_maskbits(cyg_flag_t* flag, cyg_flag_value_t value);
cyg_flag_value_t cyg_flag_wait(cyg_flag_t* flag, cyg_flag_value_t pattern,
cyg_flag_mode_t mode);
cyg_flag_value_t cyg_flag_timed_wait(cyg_flag_t* flag, cyg_flag_value_t pattern,
cyg_flag_mode_t mode, cyg_tick_count_t abstime);
cyg_flag_value_t cyg_flag_poll(cyg_flag_t* flag, cyg_flag_value_t pattern,
cyg_flag_mode_t mode);
cyg_flag_value_t cyg_flag_peek(cyg_flag_t* flag);
cyg_bool_t cyg_flag_waiting(cyg_flag_t* flag);

Description

Event flags allow a consumer thread to wait for one of several different types of event to occur. Alternatively it is possible to wait for some combination of events. The implementation is relatively straightforward. Each event flag contains a 32-bit integer. Application code associates these bits with specific events, so for example bit 0 could indicate that an I/O operation has completed and data is available, while bit 1 could indicate that the user has pressed a start button. A producer thread or a DSR can cause one or more of the bits to be set, and a consumer thread currently waiting for these bits will be woken up.

Unlike semaphores no attempt is made to keep track of event counts. It does not matter whether a given event occurs once or multiple times before being consumed, the corresponding bit in the event flag will change only once. However semaphores cannot easily be used to handle multiple event sources. Event flags can often be used as an alternative to condition variables, although they cannot be used for completely arbitrary conditions and they only support the equivalent of condition variable broadcasts, not signals.

Before an event flag can be used it must be initialized by a call to cyg_flag_init. This takes a pointer to a cyg_flag_t data structure, which can be part of a larger structure. All 32 bits in the event flag will be set to 0, indicating that no events have yet occurred. If an event flag is no longer required it can be cleaned up with a call to cyg_flag_destroy, allowing the memory for the cyg_flag_t structure to be re-used.

A consumer thread can wait for one or more events by calling cyg_flag_wait. This takes three arguments. The first identifies a particular event flag. The second is some combination of bits, indicating which events are of interest. The final argument should be one of the following:
Event Flags

CYG_FLAG_WAITMODE_AND

The call to cyg_flag_wait will block until all the specified event bits are set. The event flag is not cleared when the wait succeeds, in other words all the bits remain set.

CYG_FLAG_WAITMODE_OR

The call will block until at least one of the specified event bits is set. The event flag is not cleared on return.

CYG_FLAG_WAITMODE_AND | CYG_FLAG_WAITMODE_CLR

The call will block until all the specified event bits are set, and the entire event flag is cleared when the call succeeds. Note that if this mode of operation is used then a single event flag cannot be used to store disjoint sets of events, even though enough bits might be available. Instead each disjoint set of events requires its own event flag.

CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR

The call will block until at least one of the specified event bits is set, and the entire flag is cleared when the call succeeds.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.

A call to cyg_flag_wait normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If cyg_thread_release is used to unblock a thread that is currently in a wait operation, the cyg_flag_wait call will instead return 0.

cyg_flag_timed_wait is a variant of cyg_flag_wait which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. cyg_flag_poll is a non-blocking variant: if the wait operation can succeed immediately it acts like cyg_flag_wait, otherwise it returns immediately with a value of 0.
**Valid contexts**

cyg_flag_init is typically called during system initialization but may also be called in thread context. The same applies to cyg_flag_destroy, cyg_flag_wait and cyg_flag_timed_wait may only be called from thread context. The remaining functions may be called from thread or DSR context.
Spinlocks

Name

cyg_spinlock_create, cyg_spinlock_destroy, cyg_spinlock_spin,
cyg_spinlock_clear, cyg_spinlock_test, cyg_spinlock_spin_intsave,
cyg_spinlock_clear_intsave — Low-level Synchronization Primitive

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_spinlock_init(cyg_spinlock_t* lock, cyg_bool_t locked);
void cyg_spinlock_destroy(cyg_spinlock_t* lock);
void cyg_spinlock_clear(cyg_spinlock_t* lock);
cyg_bool_t cyg_spinlock_try(cyg_spinlock_t* lock);
cyg_bool_t cyg_spinlock_test(cyg_spinlock_t* lock);
void cyg_spinlock_spin_intsave(cyg_spinlock_t* lock, cyg_addrword_t* istate);
void cyg_spinlock_clear_intsave(cyg_spinlock_t* lock, cyg_addrword_t istate);

Description

Spinlocks provide an additional synchronization primitive for applications running on SMP systems. They operate at a lower level than the other primitives such as mutexes, and for most purposes the higher-level primitives should be preferred. However there are some circumstances where a spinlock is appropriate, especially when interrupt handlers and threads need to share access to hardware, and on SMP systems the kernel implementation itself depends on spinlocks.

Essentially a spinlock is just a simple flag. When code tries to claim a spinlock it checks whether or not the flag is already set. If not then the flag is set and the operation succeeds immediately. The exact implementation of this is hardware-specific, for example it may use a test-and-set instruction to guarantee the desired behaviour even if several processors try to access the spinlock at the exact same time. If it is not possible to claim a spinlock then the current thread spins in a tight loop, repeatedly checking the flag until it is clear. This behaviour is very different from other synchronization primitives such as mutexes, where contention would cause a thread to be suspended. The assumption is that a spinlock will only be held for a very short time. If claiming a spinlock could cause the current thread to be suspended then spinlocks could not be used inside interrupt handlers, which is not acceptable.

This does impose a constraint on any code which uses spinlocks. Specifically it is important that spinlocks are held only for a short period of time, typically just some dozens of instructions. Otherwise another processor could be blocked on the spinlock for a long time, unable to do any useful work. It is also important that a thread which owns a spinlock does not get preempted because that might cause another processor to spin for a whole timeslice period, or longer. One way of achieving this is to disable interrupts on the current processor, and the function cyg_spinlock_spin_intsave is provided to facilitate this.

Spinlocks should not be used on single-processor systems. Consider a high priority thread which attempts to claim a spinlock already held by a lower priority thread: it will just loop forever and the lower priority thread will never get another chance to run and release the spinlock. Even if the two threads were running at the same priority, the one attempting to claim the spinlock would spin until it was timesliced and a lot of cpu time would
Spinlocks

be wasted. If an interrupt handler tried to claim a spinlock owned by a thread, the interrupt handler would loop forever. Therefore spinlocks are only appropriate for SMP systems where the current owner of a spinlock can continue running on a different processor.

Before a spinlock can be used it must be initialized by a call to `cyg_spinlock_init`. This takes two arguments, a pointer to a `cyg_spinlock_t` data structure, and a flag to specify whether the spinlock starts off locked or unlocked. If a spinlock is no longer required then it can be destroyed by a call to `cyg_spinlock_destroy`.

There are two routines for claiming a spinlock: `cyg_spinlock_spin` and `cyg_spinlock_spin_intsave`. The former can be used when it is known the current code will not be preemption, for example because it is running in an interrupt handler or because interrupts are disabled. The latter will disable interrupts in addition to claiming the spinlock, so is safe to use in all circumstances. The previous interrupt state is returned via the second argument, and should be used in a subsequent call to `cyg_spinlock_clear_intsave`.

Similarly there are two routines for releasing a spinlock: `cyg_spinlock_clear` and `cyg_spinlock_clear_intsave`. Typically the former will be used if the spinlock was claimed by a call to `cyg_spinlock_spin`, and the latter when `cyg_spinlock_intsave` was used.

There are two additional routines. `cyg_spinlock_try` is a non-blocking version of `cyg_spinlock_spin`: if possible the lock will be claimed and the function will return `true`; otherwise the function will return immediately with failure. `cyg_spinlock_test` can be used to find out whether or not the spinlock is currently locked. This function must be used with care because, especially on a multiprocessor system, the state of the spinlock can change at any time.

Spinlocks should only be held for a short period of time, and attempting to claim a spinlock will never cause a thread to be suspended. This means that there is no need to worry about priority inversion problems, and concepts such as priority ceilings and inheritance do not apply.

Valid contexts

All of the spinlock functions can be called from any context, including ISR and DSR context. Typically `cyg_spinlock_init` is only called during system initialization.
Scheduler Control

Name

cyg_scheduler_start, cyg_scheduler_lock, cyg_scheduler_unlock,
cyg_scheduler_safe_lock, cyg_scheduler_read_lock — Control the state of the scheduler

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_scheduler_start(void);
void cyg_scheduler_lock(void);
void cyg_scheduler_unlock(void);
cyg_ucount32 cyg_scheduler_read_lock(void);

Description

cyg_scheduler_start should only be called once, to mark the end of system initialization. In typical configurations it is called automatically by the system startup, but some applications may bypass the standard startup in which case cyg_scheduler_start will have to be called explicitly. The call will enable system interrupts, allowing I/O operations to commence. Then the scheduler will be invoked and control will be transferred to the highest priority runnable thread. The call will never return.

The various data structures inside the eCos kernel must be protected against concurrent updates. Consider a call to cyg_semaphore_post which causes a thread to be woken up: the semaphore data structure must be updated to remove the thread from its queue; the scheduler data structure must also be updated to mark the thread as runnable; it is possible that the newly runnable thread has a higher priority than the current one, in which case preemption is required. If in the middle of the semaphore post call an interrupt occurred and the interrupt handler tried to manipulate the same data structures, for example by making another thread runnable, then it is likely that the structures will be left in an inconsistent state and the system will fail.

To prevent such problems the kernel contains a special lock known as the scheduler lock. A typical kernel function such as cyg_semaphore_post will claim the scheduler lock, do all its manipulation of kernel data structures, and then release the scheduler lock. The current thread cannot be preempted while it holds the scheduler lock. If an interrupt occurs and a DSR is supposed to run to signal that some event has occurred, that DSR is postponed until the scheduler unlock operation. This prevents concurrent updates of kernel data structures.

The kernel exports three routines for manipulating the scheduler lock. cyg_scheduler_lock can be called to claim the lock. On return it is guaranteed that the current thread will not be preempted, and that no other code is manipulating any kernel data structures. cyg_scheduler_unlock can be used to release the lock, which may cause the current thread to be preempted. cyg_scheduler_read_lock can be used to query the current state of the scheduler lock. This function should never be needed because well-written code should always know whether or not the scheduler is currently locked, but may prove useful during debugging.

The implementation of the scheduler lock involves a simple counter. Code can call cyg_scheduler_lock multiple times, causing the counter to be incremented each time, as long as cyg_scheduler_unlock is called the same number of times. This behaviour is different from mutexes where an attempt by a thread to lock a mutex multiple times will result in deadlock or an assertion failure.
Typical application code should not use the scheduler lock. Instead other synchronization primitives such as mutexes and semaphores should be used. While the scheduler is locked the current thread cannot be preempted, so any higher priority threads will not be able to run. Also no DSRs can run, so device drivers may not be able to service I/O requests. However there is one situation where locking the scheduler is appropriate: if some data structure needs to be shared between an application thread and a DSR associated with some interrupt source, the thread can use the scheduler lock to prevent concurrent invocations of the DSR and then safely manipulate the structure. It is desirable that the scheduler lock is held for only a short period of time, typically some tens of instructions. In exceptional cases there may also be some performance-critical code where it is more appropriate to use the scheduler lock rather than a mutex, because the former is more efficient.

Valid contexts

_cyg_scheduler_start_ can only be called during system initialization, since it marks the end of that phase. The remaining functions may be called from thread or DSR context. Locking the scheduler from inside the DSR has no practical effect because the lock is claimed automatically by the interrupt subsystem before running DSRs, but allows functions to be shared between normal thread code and DSRs.
Interrupt Handling

Name

cyg_interrupt_create, cyg_interrupt_delete, cyg_interrupt_attach,
cyg_interrupt_detach, cyg_interrupt_configure, cyg_interrupt_acknowledge,
cyg_interrupt_enable, cyg_interrupt_disable, cyg_interrupt_mask,
ocyg_interrupt_mask_intunsafe, cyg_interrupt_unmask,
cyg_interrupt_unmask_intunsafe, cyg_interrupt_set_cpu,
cyg_interrupt_get_cpu, cyg_interrupt_get_vsr, cyg_interrupt_set_vsr—
Manage interrupt handlers

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_interrupt_create(cyg_vector_t vector, cyg_priority_t priority, cyg_addrword_t data, cyg_ISR_t* isr, cyg_DSR_t* dsr, cyg_handle_t* handle, cyg_interrupt* intr);
void cyg_interrupt_delete(cyg_handle_t interrupt);
void cyg_interrupt_attach(cyg_handle_t interrupt);
void cyg_interrupt_detach(cyg_handle_t interrupt);
void cyg_interrupt_configure(cyg_vector_t vector, cyg_bool_t level, cyg_bool_t up);
void cyg_interrupt_acknowledge(cyg_vector_t vector);
void cyg_interrupt_disable(void);
void cyg_interrupt_enable(void);
void cyg_interrupt_mask(cyg_vector_t vector);
void cyg_interrupt_mask_intunsafe(cyg_vector_t vector);
void cyg_interrupt_unmask(cyg_vector_t vector);
void cyg_interrupt_unmask_intunsafe(cyg_vector_t vector);
void cyg_interrupt_set_cpu(cyg_vector_t vector, cyg_cpu_t cpu);
cyg_cpu_t cyg_interrupt_get_cpu(cyg_vector_t vector);
void cyg_interrupt_get_vsr(cyg_vector_t vector, cyg_VSR_t** vsr);
void cyg_interrupt_set_vsr(cyg_vector_t vector, cyg_VSR_t* vsr);

Description

The kernel provides an interface for installing interrupt handlers and controlling when interrupts occur. This functionality is used primarily by eCos device drivers and by any application code that interacts directly with hardware. However in most cases it is better to avoid using this kernel functionality directly, and instead the device driver API provided by the common HAL package should be used. Use of the kernel package is optional, and some applications such as RedBoot work with no need for multiple threads or synchronization primitives. Any code which calls the kernel directly rather than the device driver API will not function in such a configuration. When the kernel package is present the device driver API is implemented as #define's to the equivalent kernel calls, otherwise it is implemented inside the common HAL package. The latter implementation can be simpler than the kernel one because there is no need to consider thread preemption and similar issues.

The exact details of interrupt handling vary widely between architectures. The functionality provided by the kernel abstracts away from many of the details of the underlying hardware, thus simplifying application development. However this is not always successful. For example, if some hardware does not provide any support at all for masking specific interrupts then calling cyg_interrupt_mask may not behave as intended: instead
Interrupt Handling

of masking just the one interrupt source it might disable all interrupts, because that is as close to the desired behaviour as is possible given the hardware restrictions. Another possibility is that masking a given interrupt source also affects all lower-priority interrupts, but still allows higher-priority ones. The documentation for the appropriate HAL packages should be consulted for more information about exactly how interrupts are handled on any given hardware. The HAL header files will also contain useful information.

Interrupt Handlers

Interrupt handlers are created by a call to \texttt{cyg\_interrupt\_create}. This takes the following arguments:

\begin{itemize}
  \item \texttt{cyg\_vector\_t \textit{vector}}
  \begin{itemize}
    \item The interrupt vector, a small integer, identifies the specific interrupt source. The appropriate hardware documentation or HAL header files should be consulted for details of which vector corresponds to which device.
  \end{itemize}
  \item \texttt{cyg\_priority\_t \textit{priority}}
  \begin{itemize}
    \item Some hardware may support interrupt priorities, where a low priority interrupt handler can in turn be interrupted by a higher priority one. Again hardware-specific documentation should be consulted for details about what the valid interrupt priority levels are.
  \end{itemize}
  \item \texttt{cyg\_addrword\_t \textit{data}}
  \begin{itemize}
    \item When an interrupt occurs eCos will first call the associated interrupt service routine or ISR, then optionally a deferred service routine or DSR. The \textit{data} argument to \texttt{cyg\_interrupt\_create} will be passed to both these functions. Typically it will be a pointer to some data structure.
  \end{itemize}
  \item \texttt{cyg\_ISR\_t \textit{isr}}
  \begin{itemize}
    \item When an interrupt occurs the hardware will transfer control to the appropriate vector service routine or VSR, which is usually provided by eCos. This performs any appropriate processing, for example to work out exactly which interrupt occurred, and then as quickly as possible transfers control the installed ISR. An ISR is a C function which takes the following form:
  \begin{verbatim}
  cyg\_uint32
  isr\_function(cyg\_vector\_t vector, cyg\_addrword\_t data)
  {
    cyg\_bool\_t dsr\_required = 0;
    ...
    return dsr\_required ?
      (CYG\_ISR\_CALL\_DSR | CYG\_ISR\_HANDLED) :
      CYG\_ISR\_HANDLED;
  }
  \end{verbatim}

    \item The first argument identifies the particular interrupt source, especially useful if there multiple instances of a given device and a single ISR can be used for several different interrupt vectors. The second argument is the \textit{data} field passed to \texttt{cyg\_interrupt\_create}, usually a pointer to some data structure. The exact conditions under which an ISR runs will depend partly on the hardware and partly on configuration options. Interrupts may currently be disabled globally, especially if the hardware does not support interrupt priorities. Alternatively interrupts may be enabled such that higher priority interrupts are allowed through. The ISR may be running on a separate interrupt stack, or on the stack of whichever thread was running at the time the interrupt happened.
\end{itemize}
Interrupt Handling

A typical ISR will do as little work as possible, just enough to meet the needs of the hardware and then acknowledge the interrupt by calling `cyg_interrupt_acknowledge`. This ensures that interrupts will be quickly reenabled, so higher priority devices can be serviced. For some applications there may be one device which is especially important and whose ISR can take much longer than normal. However eCos device drivers usually will not assume that they are especially important, so their ISRs will be as short as possible.

The return value of an ISR is normally a bit mask containing zero, one or both of the following bits: `CYG_ISR_CALL_DSR` or `CYG_ISR_HANDLED`. The former indicates that further processing is required at DSR level, and the interrupt handler’s DSR will be run as soon as possible. The latter indicates that the interrupt was handled by this ISR so there is no need to call other interrupt handlers which might be chained on this interrupt vector. If this ISR did not handle the interrupt it should not set the `CYG_ISR_HANDLED` bit so that other chained interrupt handlers may handle the interrupt.

An ISR is allowed to make very few kernel calls. It can manipulate the interrupt mask, and on SMP systems it can use spinlocks. However an ISR must not make higher-level kernel calls such as posting to a semaphore, instead any such calls must be made from the DSR. This avoids having to disable interrupts throughout the kernel and thus improves interrupt latency.

cyg_DSR_t dsr

If an interrupt has occurred and the ISR has returned a value with `CYG_ISR_CALL_DSR` bit being set, the system will call the DSR associated with this interrupt handler. If the scheduler is not currently locked then the DSR will run immediately. However if the interrupted thread was in the middle of a kernel call and had locked the scheduler, then the DSR will be deferred until the scheduler is again unlocked. This allows the DSR to make certain kernel calls safely, for example posting to a semaphore or signalling a condition variable. A DSR is a C function which takes the following form:

```c
void
dsr_function(cyg_vector_t vector,
             cyg_ucount32 count,
             cyg_addrword_t data)
{
}
```

The first argument identifies the specific interrupt that has caused the DSR to run. The second argument indicates the number of these interrupts that have occurred and for which the ISR requested a DSR. Usually this will be 1, unless the system is suffering from a very heavy load. The third argument is the `data` field passed to `cyg_interrupt_create`.

cyg_handle_t* handle

The kernel will return a handle to the newly created interrupt handler via this argument. Subsequent operations on the interrupt handler such as attaching it to the interrupt source will use this handle.

cyg_interrupt* intr

This provides the kernel with an area of memory for holding this interrupt handler and associated data. The call to `cyg_interrupt_create` simply fills in a kernel data structure. A typical next step is to call `cyg_interrupt_attach` using the handle returned by the create operation. This makes it possible to have several different interrupt handlers for a given vector, attaching whichever one is currently appropriate. Replacing an interrupt handler requires a call to `cyg_interrupt_detach`, followed by another call to `cyg_interrupt_attach` for the replacement handler. `cyg_interrupt_delete` can be used if an interrupt handler is no longer required.

Some hardware may allow for further control over specific interrupts, for example whether an interrupt is level or edge triggered. Any such hardware functionality can be accessed using `cyg_interrupt_configure`.
Interrupt Handling

The `level` argument selects between level versus edge triggered; the `up` argument selects between high and low level, or between rising and falling edges.

Usually interrupt handlers are created, attached and configured during system initialization, while global interrupts are still disabled. On most hardware it will also be necessary to call `cyg_interrupt_unmask`, since the sensible default for interrupt masking is to ignore any interrupts for which no handler is installed.

Controlling Interrupts

eCos provides two ways of controlling whether or not interrupts happen. It is possible to disable and reenable all interrupts globally, using `cyg_interrupt_disable` and `cyg_interrupt_enable`. Typically this works by manipulating state inside the cpu itself, for example setting a flag in a status register or executing special instructions. Alternatively it may be possible to mask a specific interrupt source by writing to one or to several interrupt mask registers. Hardware-specific documentation should be consulted for the exact details of how interrupt masking works, because a full implementation is not possible on all hardware.

The primary use for these functions is to allow data to be shared between ISRs and other code such as DSRs or threads. If both a thread and an ISR need to manipulate either a data structure or the hardware itself, there is a possible conflict if an interrupt happens just when the thread is doing such manipulation. Problems can be avoided by the thread either disabling or masking interrupts during the critical region. If this critical region requires only a few instructions then usually it is more efficient to disable interrupts. For larger critical regions it may be more appropriate to use interrupt masking, allowing other interrupts to occur. There are other uses for interrupt masking. For example if a device is not currently being used by the application then it may be desirable to mask all interrupts generated by that device.

There are two functions for masking a specific interrupt source, `cyg_interrupt_mask` and `cyg_interrupt_mask_intunsafe`. On typical hardware masking an interrupt is not an atomic operation, so if two threads were to perform interrupt masking operations at the same time there could be problems. `cyg_interrupt_mask` disables all interrupts while it manipulates the interrupt mask. In situations where interrupts are already known to be disabled, `cyg_interrupt_mask_intunsafe` can be used instead. There are matching functions `cyg_interrupt_unmask` and `cyg_interrupt_unmask_intsafe`.

SMP Support

On SMP systems the kernel provides an additional two functions related to interrupt handling. `cyg_interrupt_set_cpu` specifies that a particular hardware interrupt should always be handled on one specific processor in the system. In other words when the interrupt triggers it is only that processor which detects it, and it is only on that processor that the VSR and ISR will run. If a DSR is requested then it will also run on the same CPU. The function `cyg_interrupt_get_cpu` can be used to find out which interrupts are handled on which processor.

VSR Support

When an interrupt occurs the hardware will transfer control to a piece of code known as the VSR, or Vector Service Routine. By default this code is provided by eCos. Usually it is written in assembler, but on some architectures it may be possible to implement VSRs in C by specifying an interrupt attribute. Compiler documentation should be consulted for more information on this. The default eCos VSR will work out which ISR function should process the interrupt, and set up a C environment suitable for this ISR.

For some applications it may be desirable to replace the default eCos VSR and handle some interrupts directly. This minimizes interrupt latency, but it requires application developers to program at a lower level. Usually
the best way to write a custom VSR is to copy the existing one supplied by eCos and then make appropriate modifications. The function `cyg_interrupt_get_vsr` can be used to get hold of the current VSR for a given interrupt vector, allowing it to be restored if the custom VSR is no longer required. `cyg_interrupt_set_vsr` can be used to install a replacement VSR. Usually the `vsr` argument will correspond to an exported label in an assembler source file.

**Valid contexts**

In a typical configuration interrupt handlers are created and attached during system initialization, and never detached or deleted. However it is possible to perform these operations at thread level, if desired. Similarly `cyg_interrupt_configure`, `cyg_interrupt_set_vsr`, and `cyg_interrupt_set_cpu` are usually called only during system initialization, but on typical hardware may be called at any time. `cyg_interrupt_get_vsr` and `cyg_interrupt_get_cpu` may be called at any time.

The functions for enabling, disabling, masking and unmasking interrupts can be called in any context, when appropriate. It is the responsibility of application developers to determine when the use of these functions is appropriate.
Interrupt Handling
Kernel Real-time Characterization

Name

tm_basic — Measure the performance of the eCos kernel

Description

When building a real-time system, care must be taken to ensure that the system will be able to perform properly within the constraints of that system. One of these constraints may be how fast certain operations can be performed. Another might be how deterministic the overall behavior of the system is. Lastly the memory footprint (size) and unit cost may be important.

One of the major problems encountered while evaluating a system will be how to compare it with possible alternatives. Most manufacturers of real-time systems publish performance numbers, ostensibly so that users can compare the different offerings. However, what these numbers mean and how they were gathered is often not clear. The values are typically measured on a particular piece of hardware, so in order to truly compare, one must obtain measurements for exactly the same set of hardware that were gathered in a similar fashion.

Two major items need to be present in any given set of measurements. First, the raw values for the various operations; these are typically quite easy to measure and will be available for most systems. Second, the determinacy of the numbers; in other words how much the value might change depending on other factors within the system. This value is affected by a number of factors: how long interrupts might be masked, whether or not the function can be interrupted, even very hardware-specific effects such as cache locality and pipeline usage. It is very difficult to measure the determinacy of any given operation, but that determinacy is fundamentally important to proper overall characterization of a system.

In the discussion and numbers that follow, three key measurements are provided. The first measurement is an estimate of the interrupt latency: this is the length of time from when a hardware interrupt occurs until its Interrupt Service Routine (ISR) is called. The second measurement is an estimate of overall interrupt overhead: this is the length of time average interrupt processing takes, as measured by the real-time clock interrupt (other interrupt sources will certainly take a different amount of time, but this data cannot be easily gathered). The third measurement consists of the timings for the various kernel primitives.

Methodology

Key operations in the kernel were measured by using a simple test program which exercises the various kernel primitive operations. A hardware timer, normally the one used to drive the real-time clock, was used for these measurements. In most cases this timer can be read with quite high resolution, typically in the range of a few microseconds. For each measurement, the operation was repeated a number of times. Time stamps were obtained directly before and after the operation was performed. The data gathered for the entire set of operations was then analyzed, generating average (mean), maximum and minimum values. The sample variance (a measure of how close most samples are to the mean) was also calculated. The cost of obtaining the real-time clock timer values was also measured, and was subtracted from all other times.

Most kernel functions can be measured separately. In each case, a reasonable number of iterations are performed. Where the test case involves a kernel object, for example creating a task, each iteration is performed on a different object. There is also a set of tests which measures the interactions between multiple tasks and certain kernel primitives. Most functions are tested in such a way as to determine the variations introduced by varying numbers of objects in the system. For example, the mailbox tests measure the cost of a ‘peek’ operation when the mailbox is empty, has a single item, and has multiple items present. In this way, any effects of the state of the object or how many items it contains can be determined.
There are a few things to consider about these measurements. Firstly, they are quite micro in scale and only measure the operation in question. These measurements do not adequately describe how the timings would be perturbed in a real system with multiple interrupting sources. Secondly, the possible aberration incurred by the real-time clock (system heartbeat tick) is explicitly avoided. Virtually all kernel functions have been designed to be interruptible. Thus the times presented are typical, but best case, since any particular function may be interrupted by the clock tick processing. This number is explicitly calculated so that the value may be included in any deadline calculations required by the end user. Lastly, the reported measurements were obtained from a system built with all options at their default values. Kernel instrumentation and asserts are also disabled for these measurements. Any number of configuration options can change the measured results, sometimes quite dramatically. For example, mutexes are using priority inheritance in these measurements. The numbers will change if the system is built with priority inheritance on mutex variables turned off.

The final value that is measured is an estimate of interrupt latency. This particular value is not explicitly calculated in the test program used, but rather by instrumenting the kernel itself. The raw number of timer ticks that elapse between the time the timer generates an interrupt and the start of the timer ISR is kept in the kernel. These values are printed by the test program after all other operations have been tested. Thus this should be a reasonable estimate of the interrupt latency over time.

**Using these Measurements**

These measurements can be used in a number of ways. The most typical use will be to compare different real-time kernel offerings on similar hardware, another will be to estimate the cost of implementing a task using eCos (applications can be examined to see what effect the kernel operations will have on the total execution time). Another use would be to observe how the tuning of the kernel affects overall operation.

**Influences on Performance**

A number of factors can affect real-time performance in a system. One of the most common factors, yet most difficult to characterize, is the effect of device drivers and interrupts on system timings. Different device drivers will have differing requirements as to how long interrupts are suppressed, for example. The eCos system has been designed with this in mind, by separating the management of interrupts (ISR handlers) and the processing required by the interrupt (DSR—Deferred Service Routine—handlers). However, since there is so much variability here, and indeed most device drivers will come from the end users themselves, these effects cannot be reliably measured. Attempts have been made to measure the overhead of the single interrupt that eCos relies on, the real-time clock timer. This should give you a reasonable idea of the cost of executing interrupt handling for devices.

**Measured Items**

This section describes the various tests and the numbers presented. All tests use the C kernel API (available by way of `cyg/kernel/kapi.h`). There is a single main thread in the system that performs the various tests. Additional threads may be created as part of the testing, but these are short lived and are destroyed between tests unless otherwise noted. The terminology “lower priority” means a priority that is less important, not necessarily lower in numerical value. A higher priority thread will run in preference to a lower priority thread even though the priority value of the higher priority thread may be numerically less than that of the lower priority thread.
Thread Primitives

Create thread

This test measures the cyg_thread_create() call. Each call creates a totally new thread. The set of threads created by this test will be reused in the subsequent thread primitive tests.

Yield thread

This test measures the cyg_thread_yield() call. For this test, there are no other runnable threads, thus the test should just measure the overhead of trying to give up the CPU.

Suspend [suspended] thread

This test measures the cyg_thread_suspend() call. A thread may be suspended multiple times; each thread is already suspended from its initial creation, and is suspended again.

Resume thread

This test measures the cyg_thread_resume() call. All of the threads have a suspend count of 2, thus this call does not make them runnable. This test just measures the overhead of resuming a thread.

Set priority

This test measures the cyg_thread_set_priority() call. Each thread, currently suspended, has its priority set to a new value.

Get priority

This test measures the cyg_thread_get_priority() call.

Kill [suspended] thread

This test measures the cyg_thread_kill() call. Each thread in the set is killed. All threads are known to be suspended before being killed.

Yield [no other] thread

This test measures the cyg_thread_yield() call again. This is to demonstrate that the cyg_thread_yield() call has a fixed overhead, regardless of whether there are other threads in the system.

Resume [suspended low priority] thread

This test measures the cyg_thread_resume() call again. In this case, the thread being resumed is lower priority than the main thread, thus it will simply become ready to run but not be granted the CPU. This test measures the cost of making a thread ready to run.

Resume [runnable low priority] thread

This test measures the cyg_thread_resume() call again. In this case, the thread being resumed is lower priority than the main thread and has already been made runnable, so in fact the resume call has no effect.

Suspend [runnable] thread

This test measures the cyg_thread_suspend() call again. In this case, each thread has already been made runnable (by previous tests).
Yield [only low priority] thread

This test measures the `cyg_thread_yield()` call. In this case, there are many other runnable threads, but they are all lower priority than the main thread, thus no thread switches will take place.

Suspend [runnable->not runnable] thread

This test measures the `cyg_thread_suspend()` call again. The thread being suspended will become non-runnable by this action.

Kill [runnable] thread

This test measures the `cyg_thread_kill()` call again. In this case, the thread being killed is currently runnable, but lower priority than the main thread.

Resume [high priority] thread

This test measures the `cyg_thread_resume()` call. The thread being resumed is higher priority than the main thread, thus a thread switch will take place on each call. In fact there will be two thread switches; one to the new higher priority thread and a second back to the test thread. The test thread exits immediately.

Thread switch

This test attempts to measure the cost of switching from one thread to another. Two equal priority threads are started and they will each yield to the other for a number of iterations. A time stamp is gathered in one thread before the `cyg_thread_yield()` call and after the call in the other thread.

Scheduler Primitives

Scheduler lock

This test measures the `cyg_scheduler_lock()` call.

Scheduler unlock [0 threads]

This test measures the `cyg_scheduler_unlock()` call. There are no other threads in the system and the unlock happens immediately after a lock so there will be no pending DSR’s to run.

Scheduler unlock [1 suspended thread]

This test measures the `cyg_scheduler_unlock()` call. There is one other thread in the system which is currently suspended.

Scheduler unlock [many suspended threads]

This test measures the `cyg_scheduler_unlock()` call. There are many other threads in the system which are currently suspended. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of `cyg_scheduler_unlock()`.

Scheduler unlock [many low priority threads]

This test measures the `cyg_scheduler_unlock()` call. There are many other threads in the system which are runnable but are lower priority than the main thread. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of `cyg_scheduler_unlock()`.
Mutex Primitives

Init mutex

This test measures the \texttt{cyg_mutex_init()} call. A number of separate mutex variables are created. The purpose of this test is to measure the cost of creating a new mutex and introducing it to the system.

Lock [unlocked] mutex

This test measures the \texttt{cyg_mutex_lock()} call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Unlock [locked] mutex

This test measures the \texttt{cyg_mutex_unlock()} call. The purpose of this test is to measure the cost of unlocking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Trylock [unlocked] mutex

This test measures the \texttt{cyg_mutex_trylock()} call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Trylock [locked] mutex

This test measures the \texttt{cyg_mutex_trylock()} call. The purpose of this test is to measure the cost of locking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Destroy mutex

This test measures the \texttt{cyg_mutex_destroy()} call. The purpose of this test is to measure the cost of deleting a mutex from the system. There are no other threads executing in the system while this test runs.

Unlock/Lock mutex

This test attempts to measure the cost of unlocking a mutex for which there is another higher priority thread waiting. When the mutex is unlocked, the higher priority waiting thread will immediately take the lock. The time from when the unlock is issued until after the lock succeeds in the second thread is measured, thus giving the round-trip or circuit time for this type of synchronizer.

Mailbox Primitives

Create mbox

This test measures the \texttt{cyg_mbox_create()} call. A number of separate mailboxes is created. The purpose of this test is to measure the cost of creating a new mailbox and introducing it to the system.

Peek [empty] mbox

This test measures the \texttt{cyg_mbox.peek()} call. An attempt is made to peek the value in each mailbox, which is currently empty. The purpose of this test is to measure the cost of checking a mailbox for a value without blocking.
Kernel Real-time Characterization

Put [first] mbox

This test measures the `cyg_mbox_put()` call. One item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [1 msg] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains a single item. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Put [second] mbox

This test measures the `cyg_mbox_put()` call. A second item is added to a mailbox. The purpose of this test is to measure the cost of adding an additional item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [2 msgs] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains two items. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Get [first] mbox

This test measures the `cyg_mbox_get()` call. The first item is removed from a mailbox that currently contains two items. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Get [second] mbox

This test measures the `cyg_mbox_get()` call. The last item is removed from a mailbox that currently contains one item. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Tryput [first] mbox

This test measures the `cyg_mbox_tryput()` call. A single item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox.

Peek item [non-empty] mbox

This test measures the `cyg_mbox_peek_item()` call. A single item is fetched from a mailbox that contains a single item. The purpose of this test is to measure the cost of obtaining an item without disturbing the mailbox.

Tryget [non-empty] mbox

This test measures the `cyg_mbox_tryget()` call. A single item is removed from a mailbox that contains exactly one item. The purpose of this test is to measure the cost of obtaining one item from a non-empty mailbox.

Peek item [empty] mbox

This test measures the `cyg_mbox_peek_item()` call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.
Tryget [empty] mbox

This test measures the \texttt{cyg\_mbox\_tryget()} call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.

Waiting to get mbox

This test measures the \texttt{cyg\_mbox\_waiting\_to\_get()} call. The purpose of this test is to measure the cost of determining how many threads are waiting to obtain a message from this mailbox.

Waiting to put mbox

This test measures the \texttt{cyg\_mbox\_waiting\_to\_put()} call. The purpose of this test is to measure the cost of determining how many threads are waiting to put a message into this mailbox.

Delete mbox

This test measures the \texttt{cyg\_mbox\_delete()} call. The purpose of this test is to measure the cost of destroying a mailbox and removing it from the system.

Put/Get mbox

In this round-trip test, one thread is sending data to a mailbox that is being consumed by another thread. The time from when the data is put into the mailbox until it has been delivered to the waiting thread is measured. Note that this time will contain a thread switch.

\section*{Semaphore Primitives}

Init semaphore

This test measures the \texttt{cyg\_semaphore\_init()} call. A number of separate semaphore objects are created and introduced to the system. The purpose of this test is to measure the cost of creating a new semaphore.

Post [0] semaphore

This test measures the \texttt{cyg\_semaphore\_post()} call. Each semaphore currently has a value of 0 and there are no other threads in the system. The purpose of this test is to measure the overhead cost of posting to a semaphore. This cost will differ if there is a thread waiting for the semaphore.

Wait [1] semaphore

This test measures the \texttt{cyg\_semaphore\_wait()} call. The semaphore has a current value of 1 so the call is non-blocking. The purpose of the test is to measure the overhead of “taking” a semaphore.

Trywait [0] semaphore

This test measures the \texttt{cyg\_semaphore\_trywait()} call. The semaphore has a value of 0 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be “taken” without blocking. In this case, the answer would be no.

Trywait [1] semaphore

This test measures the \texttt{cyg\_semaphore\_trywait()} call. The semaphore has a value of 1 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be “taken” without blocking. In this case, the answer would be yes.
Kernel Real-time Characterization

Peek semaphore

This test measures the `cyg_semaphore Peek()` call. The purpose of this test is to measure the cost of obtaining the current semaphore count value.

Destroy semaphore

This test measures the `cyg_semaphore_destroy()` call. The purpose of this test is to measure the cost of deleting a semaphore from the system.

Post/Wait semaphore

In this round-trip test, two threads are passing control back and forth by using a semaphore. The time from when one thread calls `cyg_semaphore_post()` until the other thread completes its `cyg_semaphore_wait()` is measured. Note that each iteration of this test will involve a thread switch.

Counters

Create counter

This test measures the `cyg_counter_create()` call. A number of separate counters are created. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Get counter value

This test measures the `cyg_counter_current_value()` call. The current value of each counter is obtained.

Set counter value

This test measures the `cyg_counter_set_value()` call. Each counter is set to a new value.

Tick counter

This test measures the `cyg_counter_tick()` call. Each counter is “ticked” once.

Delete counter

This test measures the `cyg_counter_delete()` call. Each counter is deleted from the system. The purpose of this test is to measure the cost of deleting a counter object.

Alarms

Create alarm

This test measures the `cyg_alarm_create()` call. A number of separate alarms are created, all attached to the same counter object. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Initialize alarm

This test measures the `cyg_alarm_initialize()` call. Each alarm is initialized to a small value.

Disable alarm

This test measures the `cyg_alarm_disable()` call. Each alarm is explicitly disabled.
Enable alarm

This test measures the cyg_alarm_enable() call. Each alarm is explicitly enabled.

Delete alarm

This test measures the cyg_alarm_delete() call. Each alarm is destroyed. The purpose of this test is to measure the cost of deleting an alarm and removing it from the system.

Tick counter [1 alarm]

This test measures the cyg_counter_tick() call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of “ticking” a counter when it has a single attached alarm. In this test, the alarm is not activated (fired).

Tick counter [many alarms]

This test measures the cyg_counter_tick() call. A counter is created that has multiple alarms attached to it. The purpose of this test is to measure the cost of “ticking” a counter when it has many attached alarms. In this test, the alarms are not activated (fired).

Tick & fire counter [1 alarm]

This test measures the cyg_counter_tick() call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of “ticking” a counter when it has a single attached alarm. In this test, the alarm is activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Tick & fire counter [many alarms]

This test measures the cyg_counter_tick() call. A counter is created that has multiple alarms attached to it. The purpose of this test is to measure the cost of “ticking” a counter when it has many attached alarms. In this test, the alarms are activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Alarm latency [0 threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are no threads that can be run, other than the system idle thread, when the clock interrupt occurs (all threads are suspended).

Alarm latency [2 threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are exactly two threads which are running when the clock interrupt occurs. They are simply passing back and forth by way of the cyg_thread_yield() call. The purpose of this test is to measure the variations in the latency when there are executing threads.

Alarm latency [many threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are a number of threads which are running when the clock interrupt occurs. They are simply passing back and forth by way of the cyg_thread_yield() call. The purpose of this test is to measure the variations in the latency when there are many executing threads.
II. The eCos Hardware Abstraction Layer (HAL)
Chapter 1. Introduction

This is an initial specification of the eCos Hardware Abstraction Layer (HAL). The HAL abstracts the underlying hardware of a processor architecture and/or the platform to a level sufficient for the eCos kernel to be ported onto that platform.

Caveat: This document is an informal description of the HAL capabilities and is not intended to be full documentation, although it may be used as a source for such. It also describes the HAL as it is currently implemented for the architectures targeted in this release. It most closely describes the HALs for the MIPS, I386 and PowerPC HALs. Other architectures are similar but may not be organized precisely as described here.
Chapter 1. Introduction
Chapter 2. Architecture, Variant and Platform

We have identified three levels at which the HAL must operate.

- The architecture HAL abstracts the basic CPU architecture and includes things like interrupt delivery, context switching, CPU startup etc.
- The variant HAL encapsulates features of the CPU variant such as caches, MMU and FPU features. It also deals with any on-chip peripherals such as memory and interrupt controllers. For architectural variations, the actual implementation of the variation is often in the architectural HAL, and the variant HAL simply provides the correct configuration definitions.
- The platform HAL abstracts the properties of the current platform and includes things like platform startup, timer devices, I/O register access and interrupt controllers.

The boundaries between these three HAL levels are necessarily blurred since functionality shifts between levels on a target-by-target basis. For example caches and MMU may be either an architecture feature or a variant feature. Similarly, memory and interrupt controllers may be on-chip and in the variant HAL, or off-chip and in the platform HAL.

Generally there is a separate package for each of the architecture, variant and package HALs for a target. For some of the older targets, or where it would be essentially empty, the variant HAL is omitted.
Chapter 2. Architecture, Variant and Platform
Chapter 3. General principles

The HAL has been implemented according to the following general principles:

1. The HAL is implemented in C and assembler, although the eCos kernel is largely implemented in C++. This is to permit the HAL the widest possible applicability.

2. All interfaces to the HAL are implemented by CPP macros. This allows them to be implemented as inline C code, inline assembler or function calls to external C or assembler code. This allows the most efficient implementation to be selected without affecting the interface. It also allows them to be redefined if the platform or variant HAL needs to replace or enhance a definition from the architecture HAL.

3. The HAL provides simple, portable mechanisms for dealing with the hardware of a wide range of architectures and platforms. It is always possible to bypass the HAL and program the hardware directly, but this may lead to a loss of portability.
Chapter 3. General principles
Chapter 4. HAL Interfaces

This section describes the main HAL interfaces.

Base Definitions

These are definitions that characterize the properties of the base architecture that are used to compile the portable parts of the kernel. They are concerned with such things as portable type definitions, endianness, and labeling.

These definitions are supplied by the cyg/hal/basetype.h header file which is supplied by the architecture HAL. It is included automatically by cyg/infra/cyg_type.h.

Byte order

CYG_BYTEORDER

This defines the byte order of the target and must be set to either CYG_LSBFIRST or CYG_MSBFIRST.

Label Translation

CYG_LABEL_NAME(name)

This is a wrapper used in some C and C++ files which use labels defined in assembly code or the linker script. It need only be defined if the default implementation in cyg/infra/cyg_type.h, which passes the name argument unaltered, is inadequate. It should be paired with CYG_LABEL_DEFN().

CYG_LABEL_DEFN(name)

This is a wrapper used in assembler sources and linker scripts which define labels. It need only be defined if the default implementation in cyg/infra/cyg_type.h, which passes the name argument unaltered, is inadequate. The most usual alternative definition of this macro prepends an underscore to the label name.

Base types

cyg_halint8
cyg_halint16
cyg_halint32
cyg_halint64
cyg_halcount8
cyg_halcount16
cyg_halcount32
cyg_halcount64
cyg_halbool
These macros define the C base types that should be used to define variables of the given size. They only need to be defined if the default types specified in cyg/infra/cyg_type.h cannot be used. Note that these are only the base types, they will be composed with signed and unsigned to form full type specifications.

### Atomic types

```c
#define cyg_halatomic CYG_ATOMIC
```

These types are guaranteed to be read or written in a single uninterruptible operation. It is architecture defined what size this type is, but it will be at least a byte.

### Architecture Characterization

These are definition that are related to the basic architecture of the CPU. These include the CPU context save format, context switching, bit twiddling, breakpoints, stack sizes and address translation.

Most of these definition are found in cyg/hal/hal_arch.h. This file is supplied by the architecture HAL. If there are variant or platform specific definitions then these will be found in cyg/hal/var_arch.h or cyg/hal/plf_arch.h. These files are include automatically by this header, so need not be included explicitly.

### Register Save Format

```c
typedef struct HAL_SavedRegisters
{
    /* architecture-dependent list of registers to be saved */
} HAL_SavedRegisters;
```

This structure describes the layout of a saved machine state on the stack. Such states are saved during thread context switches, interrupts and exceptions. Different quantities of state may be saved during each of these, but usually a thread context state is a subset of the interrupt state which is itself a subset of an exception state. For debugging purposes, the same structure is used for all three purposes, but where these states are significantly different, this structure may contain a union of the three states.

### Thread Context Initialization

```c
HAL_THREAD_INIT_CONTEXT( sp, arg, entry, id )
```

This macro initializes a thread’s context so that it may be switched to by HAL_THREAD_SWITCH_CONTEXT(). The arguments are:

- `sp`
  A location containing the current value of the thread’s stack pointer. This should be a variable or a structure field. The SP value will be read out of here and an adjusted value written back.

- `arg`
  A value that is passed as the first argument to the entry point function.
entry

The address of an entry point function. This will be called according the C calling conventions, and the
value of `arg` will be passed as the first argument. This function should have the following type signature
```c
void entry(CYG_ADDRWORD arg);
```

id

A thread id value. This is only used for debugging purposes, it is ORed into the initialization pattern for
unused registers and may be used to help identify the thread from its register dump. The least significant
16 bits of this value should be zero to allow space for a register identifier.

---

**Thread Context Switching**

```c
HAL_THREAD_LOAD_CONTEXT( to )
HAL_THREAD_SWITCH_CONTEXT( from, to )
```

These macros implement the thread switch code. The arguments are:

from

A pointer to a location where the stack pointer of the current thread will be stored.

to

A pointer to a location from where the stack pointer of the next thread will be read.

For `HAL_THREAD_LOAD_CONTEXT()` the current CPU state is discarded and the state of the destination thread is
loaded. This is only used once, to load the first thread when the scheduler is started.

For `HAL_THREAD_SWITCH_CONTEXT()` the state of the current thread is saved onto its stack, using the current
value of the stack pointer, and the address of the saved state placed in `*from`. The value in `*to` is then read and
the state of the new thread is loaded from it.

While these two operations may be implemented with inline assembler, they are normally implemented as calls
to assembly code functions in the HAL. There are two advantages to doing it this way. First, the return link of
the call provides a convenient PC value to be used in the saved context. Second, the calling conventions mean
that the compiler will have already saved the caller-saved registers before the call, so the HAL need only save
the callee-saved registers.

The implementation of `HAL_THREAD_SWITCH_CONTEXT()` saves the current CPU state on the stack, including
the current interrupt state (or at least the register that contains it). For debugging purposes it is useful to save
the entire register set, but for performance only the ABI-defined callee-saved registers need be saved. If it
is implemented, the option `CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM` controls how many registers are
saved.

The implementation of `HAL_THREAD_LOAD_CONTEXT()` loads a thread context, destroying the current context.
With a little care this can be implemented by sharing code with `HAL_THREAD_SWITCH_CONTEXT()`. To load a
thread context simply requires the saved registers to be restored from the stack and a jump or return made back
to the saved PC.

Note that interrupts are not disabled during this process, any interrupts that occur will be delivered onto the
stack to which the current CPU stack pointer points. Hence the stack pointer should never be invalid, or loaded
with a value that might cause the saved state to become corrupted by an interrupt. However, the current interrupt
state is saved and restored as part of the thread context. If a thread disables interrupts and does something to
cause a context switch, interrupts may be re-enabled on switching to another thread. Interrupts will be disabled
again when the original thread regains control.
Chapter 4. HAL Interfaces

Bit indexing

HAL_LSBIT_INDEX(index, mask)
HAL_MSBIT_INDEX(index, mask)

These macros place in index the bit index of the least significant bit in mask. Some architectures have instruction level support for one or other of these operations. If no architectural support is available, then these macros may call C functions to do the job.

Idle thread activity

HAL_IDLE_THREAD_ACTION(count)

It may be necessary under some circumstances for the HAL to execute code in the kernel idle thread’s loop. An example might be to execute a processor halt instruction. This macro provides a portable way of doing this. The argument is a copy of the idle thread’s loop counter, and may be used to trigger actions at longer intervals than every loop.

Reorder barrier

HAL_REORDER_BARRIER()

When optimizing the compiler can reorder code. In some parts of multi-threaded systems, where the order of actions is vital, this can sometimes cause problems. This macro may be inserted into places where reordering should not happen and prevents code being migrated across it by the compiler optimizer. It should be placed between statements that must be executed in the order written in the code.

Breakpoint support

HAL_BREAKPOINT(label)
HAL_BREAKINST
HAL_BREAKINST_SIZE

These macros provide support for breakpoints.

HAL_BREAKPOINT() executes a breakpoint instruction. The label is defined at the breakpoint instruction so that exception code can detect which breakpoint was executed.

HAL_BREAKINST contains the breakpoint instruction code as an integer value. HAL_BREAKINST_SIZE is the size of that breakpoint instruction in bytes. Together these may be used to place a breakpoint in any code.

GDB support

HAL_THREAD_GET_SAVED_REGISTERS(sp, regs)
HAL_GET_GDB_REGISTERS(regval, regs)
HAL_SET_GDB_REGISTERS(regs, regval)

These macros provide support for interfacing GDB to the HAL.

HAL_THREAD_GET_SAVED_REGISTERS() extracts a pointer to a HAL_SavedRegisters structure from a stack pointer value. The stack pointer passed in should be the value saved by the thread context macros. The macro will assign a pointer to the HAL_SavedRegisters structure to the variable passed as the second argument.
HAL\_GET\_GDB\_REGISTERS() translates a register state as saved by the HAL and into a register dump in the format expected by GDB. It takes a pointer to a HAL\_SavedRegisters structure in the \textit{regs} argument and a pointer to the memory to contain the GDB register dump in the \textit{regval} argument.

HAL\_SET\_GDB\_REGISTERS() translates a GDB format register dump into a format expected by the HAL. It takes a pointer to the memory containing the GDB register dump in the \textit{regval} argument and a pointer to a HAL\_SavedRegisters structure in the \textit{regs} argument.

### Setjmp and longjmp support

\texttt{CYGARC\_JMP\_BUF\_SIZE}
\texttt{hal\_jmp\_buf[CYGARC\_JMP\_BUF\_SIZE]}
\texttt{hal\_setjmp( hal\_jmp\_buf env )}
\texttt{hal\_longjmp( hal\_jmp\_buf env, int val )}

These functions provide support for the C \texttt{setjmp()} and \texttt{longjmp()} functions. Refer to the C library for further information.

### Stack Sizes

\texttt{CYGNUM\_HAL\_STACK\_SIZE\_MINIMUM}
\texttt{CYGNUM\_HAL\_STACK\_SIZE\_TYPICAL}

The values of these macros define the minimum and typical sizes of thread stacks.

\texttt{CYGNUM\_HAL\_STACK\_SIZE\_MINIMUM} defines the minimum size of a thread stack. This is enough for the thread to function correctly within eCos and allows it to take interrupts and context switches. There should also be enough space for a simple thread entry function to execute and call basic kernel operations on objects like mutexes and semaphores. However there will not be enough room for much more than this. When creating stacks for their own threads, applications should determine the stack usage needed for application purposes and then add \texttt{CYGNUM\_HAL\_STACK\_SIZE\_MINIMUM}.

\texttt{CYGNUM\_HAL\_STACK\_SIZE\_TYPICAL} is a reasonable increment over \texttt{CYGNUM\_HAL\_STACK\_SIZE\_MINIMUM}, usually about 1kB. This should be adequate for most modest thread needs. Only threads that need to define significant amounts of local data, or have very deep call trees should need to use a larger stack size.

### Address Translation

\texttt{CYGARC\_CACHED\_ADDRESS(addr)}
\texttt{CYGARC\_UNCACHED\_ADDRESS(addr)}
\texttt{CYGARC\_PHYSICAL\_ADDRESS(addr)}

These macros provide address translation between different views of memory. In many architectures a given memory location may be visible at different addresses in both cached and uncached forms. It is also possible that the MMU or some other address translation unit in the CPU presents memory to the program at a different virtual address to its physical address on the bus.

\texttt{CYGARC\_CACHED\_ADDRESS()} translates the given address to its location in cached memory. This is typically where the application will access the memory.

\texttt{CYGARC\_UNCACHED\_ADDRESS()} translates the given address to its location in uncached memory. This is typically where device drivers will access the memory to avoid cache problems. It may additionally be necessary for the cache to be flushed before the contents of this location is fully valid.
Chapter 4. HAL Interfaces

CYGARC_PHYSICAL_ADDRESS() translates the given address to its location in the physical address space. This is typically the address that needs to be passed to device hardware such as a DMA engine, ethernet device or PCI bus bridge. The physical address may not be directly accessible to the program, it may be re-mapped by address translation.

Global Pointer

CYGARC_HAL_SAVE_GP()
CYGARC_HAL_RESTORE_GP()

These macros insert code to save and restore any global data pointer that the ABI uses. These are necessary when switching context between two eCos instances - for example between an eCos application and RedBoot.

Interrupt Handling

These interfaces contain definitions related to interrupt handling. They include definitions of exception and interrupt numbers, interrupt enabling and masking.

These definitions are normally found in cyg/hal/hal_intr.h. This file is supplied by the architecture HAL. Any variant or platform specific definitions will be found in cyg/hal/var_intr.h, cyg/hal/plf_intr.h or cyg/hal/hal_platform_ints.h in the variant or platform HAL, depending on the exact target. These files are include automatically by this header, so need not be included explicitly.

Vector numbers

CYGNUM_HAL_VECTOR_XXXX
CYGNUM_HAL_VSR_MIN
CYGNUM_HAL_VSR_MAX
CYGNUM_HAL_VSR_COUNT

CYGNUM_HAL_INTERRUPT_XXXX
CYGNUM_HAL_ISR_MIN
CYGNUM_HAL_ISR_MAX
CYGNUM_HAL_ISR_COUNT

CYGNUM_HAL_EXCEPTION_XXXX
CYGNUM_HAL_EXCEPTION_MIN
CYGNUM_HAL_EXCEPTION_MAX
CYGNUM_HAL_EXCEPTION_COUNT

All possible VSR, interrupt and exception vectors are specified here, together with maximum and minimum values for range checking. While the VSR and exception numbers will be defined in this file, the interrupt numbers will normally be defined in the variant or platform HAL file that is included by this header.

There are two ranges of numbers, those for the vector service routines and those for the interrupt service routines. The relationship between these two ranges is undefined, and no equivalence should be assumed if vectors from the two ranges coincide.

The VSR vectors correspond to the set of exception vectors that can be delivered by the CPU architecture, many of these will be internal exception traps. The ISR vectors correspond to the set of external interrupts that can be delivered and are usually determined by extra decoding of the interrupt controller by the interrupt VSR.
Where a CPU supports synchronous exceptions, the range of such exceptions allowed are defined by CYGNUM_HAL_EXCEPTION_MIN and CYGNUM_HAL_EXCEPTION_MAX. The CYGNUM_HAL_EXCEPTION_XXXX definitions are standard names used by target independent code to test for the presence of particular exceptions in the architecture. The actual exception numbers will normally correspond to the VSR exception range. In future other exceptions generated by the system software (such as stack overflow) may be added.

CYGNUM_HAL_ISR_COUNT, CYGNUM_HAL_VSR_COUNT and CYGNUM_HAL_EXCEPTION_COUNT define the number of ISRs, VSRs and EXCEPTIONs respectively for the purposes of defining arrays etc. There might be a translation from the supplied vector numbers into array offsets. Hence CYGNUM_HAL_XXX_COUNT may not simply be CYGNUM_HAL_XXX_MAX - CYGNUM_HAL_XXX_MIN or CYGNUM_HAL_XXX_MAX+1.

Interrupt state control

CYG_INTERRUPT_STATE
HAL_DISABLE_INTERRUPTS( old )
HAL_RESTORE_INTERRUPTS( old )
HAL_ENABLE_INTERRUPTS()
HAL_QUERY_INTERRUPTS( state )

These macros provide control over the state of the CPUs interrupt mask mechanism. They should normally manipulate a CPU status register to enable and disable interrupt delivery. They should not access an interrupt controller.

CYG_INTERRUPT_STATE is a data type that should be used to store the interrupt state returned by HAL_DISABLE_INTERRUPTS() and HAL_QUERY_INTERRUPTS() and passed to HAL_RESTORE_INTERRUPTS().

HAL_DISABLE_INTERRUPTS() disables the delivery of interrupts and stores the original state of the interrupt mask in the variable passed in the old argument.

HAL_RESTORE_INTERRUPTS() restores the state of the interrupt mask to that recorded in old.

HAL_ENABLE_INTERRUPTS() simply enables interrupts regardless of the current state of the mask.

HAL_QUERY_INTERRUPTS() stores the state of the interrupt mask in the variable passed in the state argument. The state stored here should also be capable of being passed to HAL_RESTORE_INTERRUPTS() at a later point.

It is at the HAL implementer’s discretion exactly which interrupts are masked by this mechanism. Where a CPU has more than one interrupt type that may be masked separately (e.g. the ARM’s IRQ and FIQ) only those that can raise DSRs need to be masked here. A separate architecture specific mechanism may then be used to control the other interrupt types.

ISR and VSR management

HAL_INTERRUPT_IN_USE( vector, state )
HAL_INTERRUPT_ATTACH( vector, isr, data, object )
HAL_INTERRUPT_DETACH( vector, isr )
HAL_VSR_SET( vector, vsr, poldvsr )
HAL_VSR_GET( vector, pvsr )
HAL_VSR_SET_TO_ECOS_HANDLER( vector, poldvsr )

These macros manage the attachment of interrupt and vector service routines to interrupt and exception vectors respectively.

HAL_INTERRUPT_IN_USE() tests the state of the supplied interrupt vector and sets the value of the state parameter to either 1 or 0 depending on whether there is already an ISR attached to the vector. The HAL
will only allow one ISR to be attached to each vector, so it is a good idea to use this function before using
HAL_INTERRUPT_ATTACH().

HAL_INTERRUPT_ATTACH() attaches the ISR, data pointer and object pointer to the given vector. When an
interrupt occurs on this vector the ISR is called using the C calling convention and the vector number and data
pointer are passed to it as the first and second arguments respectively.

HAL_INTERRUPT_DETACH() detaches the ISR from the vector.

HAL_VSR_SET() replaces the VSR attached to the vector with the replacement supplied in vsr. The old VSR
is returned in the location pointed to by pvsr.

HAL_VSR_GET() assigns a copy of the VSR to the location pointed to by pvsr.

HAL_VSR_SET_TO_ECOS_HANDLER() ensures that the VSR for a specific exception is pointing at the eCos ex-
ception VSR and not one for RedBoot or some other ROM monitor. The default when running under RedBoot
is for exceptions to be handled by RedBoot and passed to GDB. This macro diverts the exception to eCos so
that it may be handled by application code. The arguments are the VSR vector to be replaces, and a location in
which to store the old VSR pointer, so that it may be replaced at a later point.

**Interrupt controller management**

HAL_INTERRUPT_MASK( vector )
HAL_INTERRUPT_UNMASK( vector )
HAL_INTERRUPT_ACKNOWLEDGE( vector )
HAL_INTERRUPT_CONFIGURE( vector, level, up )
HAL_INTERRUPT_SET_LEVEL( vector, level )

These macros exert control over any prioritized interrupt controller that is present. If no priority controller
exists, then these macros should be empty.

**Note:** These macros may not be reentrant, so care should be taken to prevent them being called while
interrupts are enabled. This means that they can be safely used in initialization code before interrupts are
enabled, and in ISRs. In DSRs, ASRs and thread code, however, interrupts must be disabled before these
macros are called. Here is an example for use in a DSR where the interrupt source is unmasked after data
processing:

...  
HAL_DISABLE_INTERRUPTS(old);
HAL_INTERRUPT_UNMASK(CYGNUM_HAL_INTERRUPT_ETH);
HAL_RESTORE_INTERRUPTS(old);
...

HAL_INTERRUPT_MASK() causes the interrupt associated with the given vector to be blocked.

HAL_INTERRUPT_UNMASK() causes the interrupt associated with the given vector to be unblocked.

HAL_INTERRUPT_ACKNOWLEDGE() acknowledges the current interrupt from the given vector. This is usually
executed from the ISR for this vector when it is prepared to allow further interrupts. Most interrupt controllers
need some form of acknowledge action before the next interrupt is allowed through. Executing this macro may
cause another interrupt to be delivered. Whether this interrupts the current code depends on the state of the
CPU interrupt mask.

HAL_INTERRUPT_CONFIGURE() provides control over how an interrupt signal is detected. The arguments are:
vector
The interrupt vector to be configured.

level
Set to true if the interrupt is detected by level, and false if it is edge triggered.

up
If the interrupt is set to level detect, then if this is true it is detected by a high signal level, and if false by a low signal level. If the interrupt is set to edge triggered, then if this is true it is triggered by a rising edge and if false by a falling edge.

HAL_INTERRUPT_SET_LEVEL() provides control over the hardware priority of the interrupt. The arguments are:

vector
The interrupt whose level is to be set.

level
The priority level to which the interrupt is to set. In some architectures the masking of an interrupt is achieved by changing its priority level. Hence this function, HAL_INTERRUPT_MASK() and HAL_INTERRUPT_UNMASK() may interfere with each other.

Clocks and Timers
These interfaces contain definitions related to clock and timer handling. They include interfaces to initialize and read a clock for generating regular interrupts, definitions for setting the frequency of the clock, and support for short timed delays.

Clock Control
HAL_CLOCK_INITIALIZE( period )
HAL_CLOCK_RESET( vector, period )
HAL_CLOCK_READ( pvalue )

These macros provide control over a clock or timer device that may be used by the kernel to provide time-out, delay and scheduling services. The clock is assumed to be implemented by some form of counter that is incremented or decremented by some external source and which raises an interrupt when it reaches a predetermined value.

HAL_CLOCK_INITIALIZE() initializes the timer device to interrupt at the given period. The period is essentially the value used to initialize the timer counter and must be calculated from the timer frequency and the desired interrupt rate. The timer device should generate an interrupt every period cycles.

HAL_CLOCK_RESET() re-initializes the timer to provoke the next interrupt. This macro is only really necessary when the timer device needs to be reset in some way after each interrupt.

HAL_CLOCK_READ() reads the current value of the timer counter and puts the value in the location pointed to by pvalue. The value stored will always be the number of timer cycles since the last interrupt, and hence ranges between zero and the initial period value. If this is a count-down cyclic timer, some arithmetic may be necessary to generate this value.
Chapter 4. HAL Interfaces

Microsecond Delay

HAL_DELAY_US(us)

This macro provides a busy loop delay for the given number of microseconds. It is intended mainly for controlling hardware that needs short delays between operations. Code which needs longer delays, of the order of milliseconds, should instead use higher-level functions such as cyg_thread_delay. The macro implementation should be thread-safe. It can also be used in ISRs or DSRs, although such usage is undesirable because of the impact on interrupt and dispatch latency.

The macro should never delay for less than the specified amount of time. It may delay for somewhat longer, although since the macro uses a busy loop this is a waste of cpu cycles. Of course the code invoking HAL_DELAY_US may get interrupted or timesliced, in which case the delay may be much longer than intended. If this is unacceptable then the calling code must take preventative action such as disabling interrupts or locking the scheduler.

There are three main ways of implementating the macro:

1. a counting loop, typically written in inline assembler, using an outer loop for the microseconds and an inner loop that consumes approximately 1us. This implementation is automatically thread-safe and does not impose any dependencies on the rest of the system, for example it does not depend on the system clock having been started. However it assumes that the cpu clock speed is known at compile-time or can be easily determined at run-time.

2. monitor one of the hardware clocks, usually the system clock. Usually this clock ticks at a rate independent of the cpu so calibration is easier. However the implementation relies on the system clock having been started, and assumes that no other code is manipulating the clock hardware. There can also be complications when the system clock wraps around.

3. a combination of the previous two. The system clock is used during system initialization to determine the cpu clock speed, and the result is then used to calibrate a counting loop. This has the disadvantage of significantly increasing the system startup time, which may be unacceptable to some applications. There are also complications if the system startup code normally runs with the cache disabled because the instruction cache will greatly affect any calibration loop.

Clock Frequency Definition

CYGNUM_HAL_RTC_NUMERATOR
CYGNUM_HAL_RTC_DENOMINATOR
CYGNUM_HAL_RTC_PERIOD

These macros are defined in the CDL for each platform and supply the necessary parameters to specify the frequency at which the clock interrupts. These parameters are usually found in the CDL definitions for the target platform, or in some cases the CPU variant.

CYGNUM_HAL_RTC_NUMERATOR and CYGNUM_HAL_RTC_DENOMINATOR specify the resolution of the clock interrupt. This resolution involves two separate values, the numerator and the denominator. The result of dividing the numerator by the denominator should correspond to the number of nanoseconds between clock interrupts. For example a numerator of 1000000000 and a denominator of 100 means that there are 10000000 nanoseconds (or 10 milliseconds) between clock interrupts. Expressing the resolution as a fraction minimizes clock drift even for frequencies that cannot be expressed as a simple integer. For example a frequency of 60Hz corresponds to a clock resolution of 16666666.66... nanoseconds. This can be expressed accurately as 1000000000 over 60.
CYGNUM_HAL_RTC_PERIOD specifies the exact value used to initialize the clock hardware, it is the value passed as a parameter to HAL_CLOCK_INITIALIZE() and HAL_CLOCK_RESET(). The exact meaning of the value and the range of legal values therefore depends on the target hardware, and the hardware documentation should be consulted for further details.

The default values for these macros in all HALs are calculated to give a clock interrupt frequency of 100Hz, or 10ms between interrupts. To change the clock frequency, the period needs to be changed, and the resolution needs to be adjusted accordingly. As an example consider the i386 PC target. The default values for these macros are:

CYGNUM_HAL_RTC_NUMERATOR 1000000000
CYGNUM_HAL_RTC_DENOMINATOR 100
CYGNUM_HAL_RTC_PERIOD 11932

To change to, say, a 200Hz clock the period needs to be halved to 5966, and to compensate the denominator needs to be doubled to 200. To change to a 1KHz interrupt rate change the period to 1193 and the denominator to 1000.

Some HALs make this process a little easier by deriving the period arithmetically from the denominator. This calculation may also involve the CPU clock frequency and possibly other factors. For example in the ARM AT91 variant HAL the period is defined by the following expression:

\(\frac{(\text{CYGNUM_HAL_ARM_AT91_CLOCK_SPEED/32})}{\text{CYGNUM_HAL_RTC_DENOMINATOR}}\)

In this case it is not necessary to change the period at all, just change the denominator to select the desired clock frequency. However, note that for certain choices of frequency, rounding errors in this calculation may result in a small clock drift over time. This is usually negligible, but if perfect accuracy is required, it may be necessary to adjust the frequency or period by hand.

HAL I/O

This section contains definitions for supporting access to device control registers in an architecture neutral fashion.

These definitions are normally found in the header file cyg/hal/hal_io.h. This file itself contains macros that are generic to the architecture. If there are variant or platform specific IO access macros then these will be found in cyg/hal/var_io.h and cyg/hal/plf_io.h in the variant or platform HALs respectively. These files are include automatically by this header, so need not be included explicitly.

This header (or more likely cyg/hal/plf_io.h) also defines the PCI access macros. For more information on these see the Section called PCI Library reference in Chapter 25.

Register address

HAL_IO_REGISTER

This type is used to store the address of an I/O register. It will normally be a memory address, an integer port address or an offset into an I/O space. More complex architectures may need to code an address space plus offset pair into a single word, or may represent it as a structure.

Values of variables and constants of this type will usually be supplied by configuration mechanisms or in target specific headers.
Register read

HAL_READ_XXX( register, value )
HAL_READ_XXX_VECTOR( register, buffer, count, stride )

These macros support the reading of I/O registers in various sizes. The XXX component of the name may be UINT8, UINT16, UINT32.

HAL_READ_XXX() reads the appropriately sized value from the register and stores it in the variable passed as the second argument.

HAL_READ_XXX_VECTOR() reads count values of the appropriate size into buffer. The stride controls how the pointer advances through the register space. A stride of zero will read the same register repeatedly, and a stride of one will read adjacent registers of the given size. Greater strides will step by larger amounts, to allow for sparsely mapped registers for example.

Register write

HAL_WRITE_XXX( register, value )
HAL_WRITE_XXX_VECTOR( register, buffer, count, stride )

These macros support the writing of I/O registers in various sizes. The XXX component of the name may be UINT8, UINT16, UINT32.

HAL_WRITE_XXX() writes the appropriately sized value from the variable passed as the second argument stored in the register.

HAL_WRITE_XXX_VECTOR() writes count values of the appropriate size from buffer. The stride controls how the pointer advances through the register space. A stride of zero will write the same register repeatedly, and a stride of one will write adjacent registers of the given size. Greater strides will step by larger amounts, to allow for sparsely mapped registers for example.

Cache Control

This section contains definitions for supporting control of the caches on the CPU.

These definitions are usually found in the header file cyg/hal/hal_cache.h. This file may be defined in the architecture, variant or platform HAL, depending on where the caches are implemented for the target. Often there will be a generic implementation of the cache control macros in the architecture HAL with the ability to override or undefine them in the variant or platform HAL. Even when the implementation of the cache macros is in the architecture HAL, the cache dimensions will be defined in the variant or platform HAL. As with other files, the variant or platform specific definitions are usually found in cyg/hal/var_cache.h and cyg/hal/plf_cache.h respectively. These files are included automatically by this header, so need not be included explicitly.

There are versions of the macros defined here for both the Data and Instruction caches. these are distinguished by the use of either DCACHE or ICACHE in the macro names. Some architectures have a unified cache, where both data and instruction share the same cache. In these cases the control macros use UCACHE and the DCACHE and ICACHE macros will just be calls to the UCACHE version. In the following descriptions, XCACHE is used to stand for any of these. Where there are issues specific to a particular cache, this will be explained in the text.

There might be target specific restrictions on the use of some of the macros which it is the user’s responsibility to comply with. Such restrictions are documented in the header file with the macro definition.
Note that destructive cache macros should be used with caution. Preceding a cache invalidation with a cache synchronization is not safe in itself since an interrupt may happen after the synchronization but before the invalidation. This might cause the state of dirty data lines created during the interrupt to be lost.

Depending on the architecture’s capabilities, it may be possible to temporarily disable the cache while doing the synchronization and invalidation which solves the problem (no new data would be cached during an interrupt). Otherwise it is necessary to disable interrupts while manipulating the cache which may take a long time.

Some platform HALs now support a pair of cache state query macros: `HAL_ICACHE_IS_ENABLED(x)` and `HAL_DCACHE_IS_ENABLED(x)` which set the argument to true if the instruction or data cache is enabled, respectively. Like most cache control macros, these are optional, because the capabilities of different targets and boards can vary considerably. Code which uses them, if it is to be considered portable, should test for their existence first by means of `ifdef`. Be sure to include `<cyg/hal/hal_cache.h>` in order to do this test and (maybe) use the macros.

**Cache Dimensions**

- **HAL_XCACHE_SIZE**
  - Defines the total size of the cache in bytes.

- **HAL_XCACHE_LINE_SIZE**
  - Defines the cache line size in bytes.

- **HAL_XCACHE_WAYS**
  - Defines the number of ways in each set and defines its level of associativity. This would be 1 for a direct mapped cache, 2 for a 2-way cache, 4 for 4-way and so on.

- **HAL_XCACHE_SETS**
  - Defines the number of sets in the cache, and is calculated from the previous values.

**Global Cache Control**

- **HAL_XCACHE_ENABLE()**
- **HAL_XCACHE_DISABLE()**
- **HAL_XCACHE_INVALIDATE_ALL()**
- **HAL_XCACHE_SYNC()**
- **HAL_XCACHE_BURST_SIZE(size)**
- **HAL_DCACHE_WRITE_MODE(mode)**
- **HAL_XCACHE_LOCK(base, size)**
- **HAL_XCACHE_UNLOCK(base, size)**
- **HAL_XCACHE_UNLOCK_ALL()**

These macros affect the state of the entire cache, or a large part of it.
Chapter 4. HAL Interfaces

HAL_XCACHE_ENABLE() and HAL_XCACHE_DISABLE()
Enable and disable the cache.

HAL_XCACHE_INVALIDATE_ALL()
Causes the entire contents of the cache to be invalidated. Depending on the hardware, this may require the cache to be disabled during the invalidation process. If so, the implementation must use HAL_XCACHE_IS_ENABLED() to save and restore the previous state.

**Note:** If this macro is called after HAL_XCACHE_SYNC() with the intention of clearing the cache (invalidating the cache after writing dirty data back to memory), you must prevent interrupts from happening between the two calls:

```c
...  
HAL_DISABLE_INTERRUPTS(old);  
HAL_XCACHE_SYNC();  
HAL_XCACHE_INVALIDATE_ALL();  
HAL_RESTORE_INTERRUPTS(old);  
...  
```

Since the operation may take a very long time, real-time responsiveness could be affected, so only do this when it is absolutely required and you know the delay will not interfere with the operation of drivers or the application.

HAL_XCACHE_SYNC()
Causes the contents of the cache to be brought into synchronization with the contents of memory. In some implementations this may be equivalent to HAL_XCACHE_INVALIDATE_ALL().

HAL_XCACHE_BURST_SIZE()
Allows the size of cache to/from memory bursts to be controlled. This macro will only be defined if this functionality is available.

HAL_DCACHE_WRITE_MODE()
Controls the way in which data cache lines are written back to memory. There will be definitions for the possible modes. Typical definitions are HAL_DCACHE_WRITEBACK_MODE and HAL_DCACHE_WRITETHRU_MODE. This macro will only be defined if this functionality is available.

HAL_XCACHE_LOCK()
Causes data to be locked into the cache. The base and size arguments define the memory region that will be locked into the cache. It is architecture dependent whether more than one locked region is allowed at any one time, and whether this operation causes the cache to cease acting as a cache for addresses outside the region during the duration of the lock. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK()
Cancels the locking of the memory region given. This should normally correspond to a region supplied in a matching lock call. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK_ALL()
Cancels all existing locked memory regions. This may be required as part of the cache initialization on some architectures. This macro will only be defined if this functionality is available.
Cache Line Control

HAL_DCACHE_ALLOCATE( base, size )
HAL_DCACHE_FLUSH( base, size )
HAL_XCACHE_INVALIDATE( base, size )
HAL_DCACHE_STORE( base, size )
HAL_DCACHE_READ_HINT( base, size )
HAL_DCACHE_WRITE_HINT( base, size )
HAL_DCACHE_ZERO( base, size )

All of these macros apply a cache operation to all cache lines that match the memory address region defined by the base and size arguments. These macros will only be defined if the described functionality is available. Also, it is not guaranteed that the cache function will only be applied to just the described regions, in some architectures it may be applied to the whole cache.

HAL_DCACHE_ALLOCATE()
   Allocates lines in the cache for the given region without reading their contents from memory, hence the contents of the lines is undefined. This is useful for preallocating lines which are to be completely overwritten, for example in a block copy operation.

HAL_DCACHE_FLUSH()
   Invalidates all cache lines in the region after writing any dirty lines to memory.

HAL_XCACHE_INVALIDATE()
   Invalidates all cache lines in the region. Any dirty lines are invalidated without being written to memory.

HAL_DCACHE_STORE()
   Writes all dirty lines in the region to memory, but does not invalidate any lines.

HAL_DCACHE_READ_HINT()
   Hints to the cache that the region is going to be read from in the near future. This may cause the region to be speculatively read into the cache.

HAL_DCACHE_WRITE_HINT()
   Hints to the cache that the region is going to be written to in the near future. This may have the identical behavior to HAL_DCACHE_READ_HINT().

HAL_DCACHE_ZERO()
   Allocates and zeroes lines in the cache for the given region without reading memory. This is useful if a large area of memory is to be cleared.

Linker Scripts

When an eCos application is linked it must be done under the control of a linker script. This script defines the memory areas, addresses and sized, into which the code and data are to be put, and allocates the various sections generated by the compiler to these.

The linker script actually used is in lib/target.ld in the install directory. This is actually manufactured out of two other files: a base linker script and an .ldi file that was generated by the memory layout tool.
Chapter 4. HAL Interfaces

The base linker script is usually supplied either by the architecture HAL or the variant HAL. It consists of a set of linker script fragments, in the form of C preprocessor macros, that define the major output sections to be generated by the link operation. The .ldi file, which is #include’ed by the base linker script, uses these macro definitions to assign the output sections to the required memory areas and link addresses.

The .ldi file is supplied by the platform HAL, and contains knowledge of the memory layout of the target platform. These files generally conform to a standard naming convention, each file being of the form:

```
pkgconf/mlt_<architecture>_<variant>_<platform>_<startup>.ldi
```

where <architecture>, <variant> and <platform> are the respective HAL package names and <startup> is the startup type which is usually one of ROM, RAM or ROMRAM.

In addition to the .ldi file, there is also a congruously name .h file. This may be used by the application to access information defined in the .ldi file. Specifically it contains the memory layout defined there, together with any additional section names defined by the user. Examples of the latter are heap areas or PCI bus memory access windows.

The .ldi is manufactured by the Memory Layout Tool (MLT). The MLT saves the memory configuration into a file named

```
include/pkgconf/mlt_<architecture>_<variant>_<platform>_<startup>.mlt
```

in the platform HAL. This file is used by the MLT to manufacture both the .ldi and .h files. Users should beware that direct edits the either of these files may be overwritten if the MLT is run and regenerates them from the .mlt file.

The names of the .ldi and .h files are defined by macro definitions in pkgconf/system.h. These are CYGHWR_MEMORY_LAYOUT_LDI and CYGHWR_MEMORY_LAYOUT_H respectively. While there will be little need for the application to refer to the .ldi file directly, it may include the .h file as follows:

```c
#include CYGHWR_MEMORY_LAYOUT_H
```

Diagnoctic Support

The HAL provides support for low level diagnostic IO. This is particularly useful during early development as an aid to bringing up a new platform. Usually this diagnostic channel is a UART or some other serial IO device, but it may equally be a a memory buffer, a simulator supported output channel, a ROM emulator virtual UART, and LCD panel, a memory mapped video buffer or any other output device.

HAL_DIAG_INIT() performs any initialization required on the device being used to generate diagnostic output. This may include, for a UART, setting baud rate, and stop, parity and character bits. For other devices it may include initializing a controller or establishing contact with a remote device.

HAL_DIAG_WRITE_CHAR(c) writes the character supplied to the diagnostic output device.

HAL_DIAG_READ_CHAR(c) reads a character from the diagnostic device into the supplied variable. This is not supported for all diagnostic devices.

These macros are defined in the header file cyg/hal/hal_diag.h. This file is usually supplied by the variant or platform HAL, depending on where the IO device being used is located. For example for on-chip UARTs it would be in the variant HAL, but for a board-level LCD panel it would be in the platform HAL.
SMP Support

eCos contains support for limited Symmetric Multi-Processing (SMP). This is only available on selected architectures and platforms.

Target Hardware Limitations

To allow a reasonable implementation of SMP, and to reduce the disruption to the existing source base, a number of assumptions have been made about the features of the target hardware.

- Modest multiprocessing. The typical number of CPUs supported is two to four, with an upper limit around eight. While there are no inherent limits in the code, hardware and algorithmic limitations will probably become significant beyond this point.

- SMP synchronization support. The hardware must supply a mechanism to allow software on two CPUs to synchronize. This is normally provided as part of the instruction set in the form of test-and-set, compare-and-swap or load-link/store-conditional instructions. An alternative approach is the provision of hardware semaphore registers which can be used to serialize implementations of these operations. Whatever hardware facilities are available, they are used in eCos to implement spinlocks.

- Coherent caches. It is assumed that no extra effort will be required to access shared memory from any processor. This means that either there are no caches, they are shared by all processors, or are maintained in a coherent state by the hardware. It would be too disruptive to the eCos sources if every memory access had to be bracketed by cache load/flush operations. Any hardware that requires this is not supported.

- Uniform addressing. It is assumed that all memory that is shared between CPUs is addressed at the same location from all CPUs. Like non-coherent caches, dealing with CPU-specific address translation is considered too disruptive to the eCos source base. This does not, however, preclude systems with non-uniform access costs for different CPUs.

- Uniform device addressing. As with access to memory, it is assumed that all devices are equally accessible to all CPUs. Since device access is often made from thread contexts, it is not possible to restrict access to device control registers to certain CPUs, since there is currently no support for binding or migrating threads to CPUs.

- Interrupt routing. The target hardware must have an interrupt controller that can route interrupts to specific CPUs. It is acceptable for all interrupts to be delivered to just one CPU, or for some interrupts to be bound to specific CPUs, or for some interrupts to be local to each CPU. At present dynamic routing, where a different CPU may be chosen each time an interrupt is delivered, is not supported. ECos cannot support hardware where all interrupts are delivered to all CPUs simultaneously with the expectation that software will resolve any conflicts.

- Inter-CPU interrupts. A mechanism to allow one CPU to interrupt another is needed. This is necessary so that events on one CPU can cause rescheduling on other CPUs.

- CPU Identifiers. Code running on a CPU must be able to determine which CPU it is running on. The CPU Id is usually provided either in a CPU status register, or in a register associated with the inter-CPU interrupt delivery subsystem. ECos expects CPU Ids to be small positive integers, although alternative representations, such as bitmaps, can be converted relatively easily. Complex mechanisms for getting the CPU Id cannot be supported. Getting the CPU Id must be a cheap operation, since it is done often, and in performance critical places such as interrupt handlers and the scheduler.
Chapter 4. HAL Interfaces

HAL Support

SMP support in any platform depends on the HAL supplying the appropriate operations. All HAL SMP support is defined in the cyg/hal/hal_smp.h header. Variant and platform specific definitions will be in cyg/hal/var_smp.h and cyg/hal/plf_smp.h respectively. These files are include automatically by this header, so need not be included explicitly.

SMP support falls into a number of functional groups.

CPU Control

This group consists of descriptive and control macros for managing the CPUs in an SMP system.

HAL_SMP_CPU_TYPE

A type that can contain a CPU id. A CPU id is usually a small integer that is used to index arrays of variables that are managed on an per-CPU basis.

HAL_SMP_CPU_MAX

The maximum number of CPUs that can be supported. This is used to provide the size of any arrays that have an element per CPU.

HAL_SMP_CPU_COUNT()

Returns the number of CPUs currently operational. This may differ from HAL_SMP_CPU_MAX depending on the runtime environment.

HAL_SMP_CPU_THIS()

Returns the CPU id of the current CPU.

HAL_SMP_CPU_NONE

A value that does not match any real CPU id. This is uses where a CPU type variable must be set to a null value.

HAL_SMP_CPU_START( cpu )

 Starts the given CPU executing at a defined HAL entry point. After performing any HAL level initialization, the CPU calls up into the kernel at cyg_kernel_cpu_startup().

HAL_SMP_CPU_RESCHEDULE_INTERRUPT( cpu, wait )

Sends the CPU a reschedule interrupt, and if wait is non-zero, waits for an acknowledgment. The interrupted CPU should call cyg_scheduler_set_need_reschedule() in its DSR to cause the reschedule to occur.

HAL_SMP_CPU_TIMESLICE_INTERRUPT( cpu, wait )

Sends the CPU a timeslice interrupt, and if wait is non-zero, waits for an acknowledgment. The interrupted CPU should call cyg_scheduler_timeslice_cpu() to cause the timeslice event to be processed.

Test-and-set Support

Test-and-set is the foundation of the SMP synchronization mechanisms.
HAL_TAS_TYPE

The type for all test-and-set variables. The test-and-set macros only support operations on a single bit (usually the least significant bit) of this location. This allows for maximum flexibility in the implementation.

HAL_TAS_SET( tas, oldb )

Performs a test and set operation on the location tas. oldb will contain true if the location was already set, and false if it was clear.

HAL_TAS_CLEAR( tas, oldb )

Performs a test and clear operation on the location tas. oldb will contain true if the location was already set, and false if it was clear.

Spinlocks

Spinlocks provide inter-CPU locking. Normally they will be implemented on top of the test-and-set mechanism above, but may also be implemented by other means if, for example, the hardware has more direct support for spinlocks.

HAL_SPINLOCK_TYPE

The type for all spinlock variables.

HAL_SPINLOCK_INIT_CLEAR

A value that may be assigned to a spinlock variable to initialize it to clear.

HAL_SPINLOCK_INIT_SET

A value that may be assigned to a spinlock variable to initialize it to set.

HAL_SPINLOCK_SPIN( lock )

The caller spins in a busy loop waiting for the lock to become clear. It then sets it and continues. This is all handled atomically, so that there are no race conditions between CPUs.

HAL_SPINLOCK_CLEAR( lock )

The caller clears the lock. One of any waiting spinners will then be able to proceed.

HAL_SPINLOCK_TRY( lock, val )

Attempts to set the lock. The value put in val will be true if the lock was claimed successfully, and false if it was not.

HAL_SPINLOCK_TEST( lock, val )

Tests the current value of the lock. The value put in val will be true if the lock is claimed and false if it is clear.

Scheduler Lock

The scheduler lock is the main protection for all kernel data structures. By default the kernel implements the scheduler lock itself using a spinlock. However, if spinlocks cannot be supported by the hardware, or there is a more efficient implementation available, the HAL may provide macros to implement the scheduler lock.
Chapter 4. HAL Interfaces

HAL_SMP_SCHEDLOCK_DATA_TYPE

A data type, possibly a structure, that contains any data items needed by the scheduler lock implementation. A variable of this type will be instantiated as a static member of the Cyg_Scheduler_SchedLock class and passed to all the following macros.

HAL_SMP_SCHEDLOCK_INIT( lock, data )

Initialize the scheduler lock. The lock argument is the scheduler lock counter and the data argument is a variable of HAL_SMP_SCHEDLOCK_DATA_TYPE type.

HAL_SMP_SCHEDLOCK_INC( lock, data )

Increment the scheduler lock. The first increment of the lock from zero to one for any CPU may cause it to wait until the lock is zeroed by another CPU. Subsequent increments should be less expensive since this CPU already holds the lock.

HAL_SMP_SCHEDLOCK_ZERO( lock, data )

Zero the scheduler lock. This operation will also clear the lock so that other CPUs may claim it.

HAL_SMP_SCHEDLOCK_SET( lock, data, new )

Set the lock to a different value, in new. This is only called when the lock is already known to be owned by the current CPU. It is never called to zero the lock, or to increment it from zero.

Interrupt Routing

The routing of interrupts to different CPUs is supported by two new interfaces in hal_intr.h.

Once an interrupt has been routed to a new CPU, the existing vector masking and configuration operations should take account of the CPU routing. For example, if the operation is not invoked on the destination CPU itself, then the HAL may need to arrange to transfer the operation to the destination CPU for correct application.

HAL_INTERRUPT_SET_CPU( vector, cpu )

Route the interrupt for the given vector to the given cpu.

HAL_INTERRUPT_GET_CPU( vector, cpu )

Set cpu to the id of the CPU to which this vector is routed.
Chapter 5. Exception Handling

Most of the HAL consists of simple macros or functions that are called via the interfaces described in the previous section. These just perform whatever operation is required by accessing the hardware and then return. The exception to this is the handling of exceptions: either synchronous hardware traps or asynchronous device interrupts. Here control is passed first to the HAL, which then passed it on to eCos or the application. After eCos has finished with it, control is then passed back to the HAL for it to tidy up the CPU state and resume processing from the point at which the exception occurred.

The HAL exceptions handling code is usually found in the file `vectors.S` in the architecture HAL. Since the reset entry point is usually implemented as one of these it also deals with system startup.

The exact implementation of this code is under the control of the HAL implementer. So long as it interacts correctly with the interfaces defined previously it may take any form. However, all current implementation follow the same pattern, and there should be a very good reason to break with this. The rest of this section describes these operate.

Exception handling normally deals with the following broad areas of functionality:

- Startup and initialization.
- Hardware exception delivery.
- Default handling of synchronous exceptions.
- Default handling of asynchronous interrupts.

HAL Startup

Execution normally begins at the reset vector with the machine in a minimal startup state. From here the HAL needs to get the machine running, set up the execution environment for the application, and finally invoke its entry point.

The following is a list of the jobs that need to be done in approximately the order in which they should be accomplished. Many of these will not be needed in some configurations.

- Initialize the hardware. This may involve initializing several subsystems in both the architecture, variant and platform HALs. These include:
  - Initialize various CPU status registers. Most importantly, the CPU interrupt mask should be set to disable interrupts.
  - Initialize the MMU, if it is used. On many platforms it is only possible to control the cacheability of address ranges via the MMU. Also, it may be necessary to remap RAM and device registers to locations other than their defaults. However, for simplicity, the mapping should be kept as close to one-to-one physical-to-virtual as possible.
  - Set up the memory controller to access RAM, ROM and I/O devices correctly. Until this is done it may not be possible to access RAM. If this is a ROM/RAM startup then the program code can now be copied to its RAM address and control transferred to it.
  - Set up any bus bridges and support chips. Often access to device registers needs to go through various bus bridges and other intermediary devices. In many systems these are combined with the memory controller,
Chapter 5. Exception Handling

so it makes sense to set these up together. This is particularly important if early diagnostic output needs to go through one of these devices.

• Set up diagnostic mechanisms. If the platform includes an LED or LCD output device, it often makes sense to output progress indications on this during startup. This helps with diagnosing hardware and software errors.

• Initialize floating point and other extensions such as SIMD and multimedia engines. It is usually necessary to enable these and maybe initialize control and exception registers for these extensions.

• Initialize interrupt controller. At the very least, it should be configured to mask all interrupts. It may also be necessary to set up the mapping from the interrupt controller’s vector number space to the CPU’s exception number space. Similar mappings may need to be set up between primary and secondary interrupt controllers.

• Disable and initialize the caches. The caches should not normally be enabled at this point, but it may be necessary to clear or initialize them so that they can be enabled later. Some architectures require that the caches be explicitly reinitialized after a power-on reset.

• Initialize the timer, clock etc. While the timer used for RTC interrupts will be initialized later, it may be necessary to set up the clocks that drive it here.

The exact order in which these initializations is done is architecture or variant specific. It is also often not necessary to do anything at all for some of these options. These fragments of code should concentrate on getting the target up and running so that C function calls can be made and code can be run. More complex initializations that cannot be done in assembly code may be postponed until calls to hal_variant_init() or hal_platform_init() are made.

Not all of these initializations need to be done for all startup types. In particular, RAM startups can reasonably assume that the ROM monitor or loader has already done most of this work.

• Set up the stack pointer, this allows subsequent initialization code to make proper procedure calls. Usually the interrupt stack is used for this purpose since it is available, large enough, and will be reused for other purposes later.

• Initialize any global pointer register needed for access to globally defined variables. This allows subsequent initialization code to access global variables.

• If the system is starting from ROM, copy the ROM template of the .data section out to its correct position in RAM. (the Section called Linker Scripts in Chapter 4).

• Zero the .bss section.

• Create a suitable C call stack frame. This may involve making stack space for call frames, and arguments, and initializing the back pointers to halt a GDB backtrace operation.

• Call hal_variant_init() and hal_platform_init(). These will perform any additional initialization needed by the variant and platform. This typically includes further initialization of the interrupt controller, PCI bus bridges, basic IO devices and enabling the caches.

• Call cyg_hal_invoke_constructors() to run any static constructors.

• Call cyg_start(). If cyg_start() returns, drop into an infinite loop.

Vectors and VSRs

The CPU delivers all exceptions, whether synchronous faults or asynchronous interrupts, to a set of hardware defined vectors. Depending on the architecture, these may be implemented in a number of different ways. Examples of existing mechanisms are:
PowerPC

Exceptions are vectored to locations 256 bytes apart starting at either zero or \texttt{0xFFF00000}. There are 16 such vectors defined by the basic architecture and extra vectors may be defined by specific variants. One of the base vectors is for all external interrupts, and another is for the architecture defined timer.

MIPS

Most exceptions and all interrupts are vectored to a single address at either \texttt{0x80000000} or \texttt{0xBFC00180}. Software is responsible for reading the exception code from the CPU \texttt{cause} register to discover its true source. Some TLB and debug exceptions are delivered to different vector addresses, but these are not used currently by eCos. One of the exception codes in the \texttt{cause} register indicates an external interrupt. Additional bits in the \texttt{cause} register provide a first-level decode for the interrupt source, one of which represents an architecture defined timer.

IA32

Exceptions are delivered via an Interrupt Descriptor Table (IDT) which is essentially an indirection table indexed by exception number. The IDT may be placed anywhere in memory. In PC hardware the standard interrupt controller can be programmed to deliver the external interrupts to a block of 16 vectors at any offset in the IDT. There is no hardware supplied mechanism for determining the vector taken, other than from the address jumped to.

ARM

All exceptions, including the FIQ and IRQ interrupts, are vectored to locations four bytes apart starting at zero. There is only room for one instruction here, which must immediately jump out to handling code higher in memory. Interrupt sources have to be decoded entirely from the interrupt controller.

With such a wide variety of hardware approaches, it is not possible to provide a generic mechanism for the substitution of exception vectors directly. Therefore, eCos translates all of these mechanisms into a common approach that can be used by portable code on all platforms.

The mechanism implemented is to attach to each hardware vector a short piece of trampoline code that makes an indirect jump via a table to the actual handler for the exception. This handler is called the Vector Service Routine (VSR) and the table is called the VSR table.

The trampoline code performs the absolute minimum processing necessary to identify the exception source, and jump to the VSR. The VSR is then responsible for saving the CPU state and taking the necessary actions to handle the exception or interrupt. The entry conditions for the VSR are as close to the raw hardware exception entry state as possible - although on some platforms the trampoline will have had to move or reorganize some registers to do its job.

To make this more concrete, consider how the trampoline code operates in each of the architectures described above:

PowerPC

A separate trampoline is contained in each of the vector locations. This code saves a few work registers away to the special purposes registers available, loads the exception number into a register and then uses that to index the VSR table and jump to the VSR. The VSR is entered with some registers move to the SPRs, and one of the data register containing the number of the vector taken.

MIPS

A single trampoline routine attached to the common vector reads the exception code out of the \texttt{cause} register and uses that value to index the VSR table and jump to the VSR. The trampoline uses the two
Chapter 5. Exception Handling

registers defined in the ABI for kernel use to do this, one of these will contain the exception vector number for the VSR.

IA32

There is a separate 3 or 4 instruction trampoline pointed to by each active IDT table entry. The trampoline for exceptions that also have an error code pop it from the stack and put it into a memory location. Trampolines for non-error-code exceptions just zero the memory location. Then all trampolines push an interrupt/exception number onto the stack, and take an indirect jump through a precalculated offset in the VSR table. This is all done without saving any registers, using memory-only operations. The VSR is entered with the vector number pushed onto the stack on top of the standard hardware saved state.

ARM

The trampoline consists solely of the single instruction at the exception entry point. This is an indirect jump via a location 32 bytes higher in memory. These locations, from 0x20 up, form the VSR table. Since each VSR is entered in a different CPU mode (SVC, UNDEF, ABORT, IRQ or FIQ) there has to be a different VSR for each exception that knows how to save the CPU state correctly.

Default Synchronous Exception Handling

Most synchronous exception VSR table entries will point to a default exception VSR which is responsible for handling all exceptions in a generic manner. The default VSR simply saves the CPU state, makes any adjustments to the CPU state that is necessary, and calls cyg_hal_exception_handler().

cyg_hal_exception_handler() needs to pass the exception on to some handling code. There are two basic destinations: enter GDB or pass the exception up to eCos. Exactly which destination is taken depends on the configuration. When the GDB stubs are included then the exception is passed to them, otherwise it is passed to eCos.

If an eCos application has been loaded by RedBoot then the VSR table entries will all point into RedBoot’s exception VSR, and will therefore enter GDB if an exception occurs. If the eCos application wants to handle an exception itself, it needs to replace the the VSR table entry with one pointing to its own VSR. It can do this with the HAL_VSR_SET_TO_ECOS_HANDLER() macro.

Default Interrupt Handling

Most asynchronous external interrupt vectors will point to a default interrupt VSR which decodes the actual interrupt being delivered from the interrupt controller and invokes the appropriate ISR.

The default interrupt VSR has a number of responsibilities if it is going to interact with the Kernel cleanly and allow interrupts to cause thread preemption.

To support this VSR an ISR vector table is needed. For each valid vector three pointers need to be stored: the ISR, its data pointer and an opaque (to the HAL) interrupt object pointer needed by the kernel. It is implementation defined whether these are stored in a single table of triples, or in three separate tables.

The VSR follows the following approximate plan:

1. Save the CPU state. In non-debug configurations, it may be possible to get away with saving less than the entire machine state. The option CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT is supported in some targets to do this.
2. Increment the kernel scheduler lock. This is a static member of the Cyg_Scheduler class, however it has also been aliased to cyg_scheduler_sched_lock so that it can be accessed from assembly code.

3. (Optional) Switch to an interrupt stack if not already running on it. This allows nested interrupts to be delivered without needing every thread to have a stack large enough to take the maximum possible nesting. It is implementation defined how to detect whether this is a nested interrupt but there are two basic techniques. The first is to inspect the stack pointer and switch only if it is not currently within the interrupt stack range; the second is to maintain a counter of the interrupt nesting level and switch only if it is zero. The option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK controls whether this happens.

4. Decode the actual external interrupt being delivered from the interrupt controller. This will yield the ISR vector number. The code to do this usually needs to come from the variant or platform HAL, so is usually present in the form of a macro or procedure callout.

5. (Optional) Re-enable interrupts to permit nesting. At this point we can potentially allow higher priority interrupts to occur. It depends on the interrupt architecture of the CPU and platform whether more interrupts will occur at this point, or whether they will only be delivered after the current interrupt has been acknowledged (by a call to HAL_INTERRUPT_ACKNOWLEDGE() in the ISR).

6. Using the ISR vector number as an index, retrieve the ISR pointer and its data pointer from the ISR vector table.

7. Construct a C call stack frame. This may involve making stack space for call frames, and arguments, and initializing the back pointers to halt a GDB backtrace operation.

8. Call the ISR, passing the vector number and data pointer. The vector number and a pointer to the saved state should be preserved across this call, preferably by storing them in registers that are defined to be callee-saved by the calling conventions.

9. If this is an un-nested interrupt and a separate interrupt stack is being used, switch back to the interrupted thread’s own stack.

10. Use the saved ISR vector number to get the interrupt object pointer from the ISR vector table.

11. Call interrupt_end() passing it the return value from the ISR, the interrupt object pointer and a pointer to the saved CPU state. This function is implemented by the Kernel and is responsible for finishing off the interrupt handling. Specifically, it may post a DSR depending on the ISR return value, and will decrement the scheduler lock. If the lock is zeroed by this operation then any posted DSRs may be called and may in turn result in a thread context switch.

12. The return from interrupt_end() may occur some time after the call. Many other threads may have executed in the meantime. So here all we may do is restore the machine state and resume execution of the interrupted thread. Depending on the architecture, it may be necessary to disable interrupts again for part of this.

The detailed order of these steps may vary slightly depending on the architecture, in particular where interrupts are enabled and disabled.
Chapter 5. Exception Handling
Chapter 6. Porting Guide

Introduction

eCos has been designed to be fairly easy to port to new targets. A target is a specific platform (board) using a given architecture (CPU type). The porting is facilitated by the hierarchical layering of the eCos sources - all architecture and platform specific code is implemented in a HAL (hardware abstraction layer).

By porting the eCos HAL to a new target the core functionality of eCos (infra, kernel, uITRON, etc) will be able to run on the target. It may be necessary to add further platform specific code such as serial drivers, display drivers, ethernet drivers, etc. to get a fully capable system.

This document is intended as a help to the HAL porting process. Due to the nature of a porting job, it is impossible to give a complete description of what has to be done for each and every potential target. This should not be considered a clear-cut recipe - you will probably need to make some implementation decisions, tweak a few things, and just plain have to rely on common sense.

However, what is covered here should be a large part of the process. If you get stuck, you are advised to read the ecos-discuss archive (http://ecos.sourceware.org/ml/ecos-discuss/) where you may find discussions which apply to the problem at hand. You are also invited to ask questions on the ecos-discuss mailing list (http://ecos.sourceware.org/intouch.html) to help you resolve problems - but as is always the case with community lists, do not consider it an oracle for any and all questions. Use common sense - if you ask too many questions which could have been answered by reading the documentation (http://ecos.sourceware.org/ecos/docs-latest/), FAQ (http://ecos.sourceware.org/fom/ecos) or source code (http://ecos.sourceware.org/cgi-bin/cvsweb.cgi/ecos/packages/?cvsroot=ecos), you are likely to be ignored.

This document will be continually improved by Red Hat engineers as time allows. Feedback and help with improving the document is sought, so if you have any comments at all, please do not hesitate to post them on ecos-discuss (mailto:ecos-discuss@ecos.sourceware.org?subject=[porting]<subject>) (please prefix the subject with [porting]).

At the moment this document is mostly an outline. There are many details to fill in before it becomes complete. Many places you'll just find a list of keywords / concepts that should be described (please post on ecos-discuss if there are areas you think are not covered).

All pages or sections where the caption ends in [TBD] contain little more than key words and/or random thoughts - there has been no work done as such on the content. The word FIXME may appear in the text to highlight places where information is missing.

HAL Structure

In order to write an eCos HAL it’s a good idea to have at least a passing understanding of how the HAL interacts with the rest of the system.

HAL Classes

The eCos HAL consists of four HAL sub-classes. This table gives a brief description of each class and partly reiterates the description in Chapter 2. The links refer to the on-line CVS tree (specifically to the sub-HALs
### HAL type | Description | Functionality Overview
--- | --- | ---
Common HAL (hal/common) ([http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/common/current?cvsroot=ecos](http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/common/current?cvsroot=ecos)) | Configuration options and functionality shared by all HALs. Driver API, eCos/ROM monitor calling interface, and tests. |
Architecture HAL (hal/<<architecture>>/arch) ([http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/arch/current?cvsroot=ecos](http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/arch/current?cvsroot=ecos)) | Functionality specific to the given architecture. Also default implementations of some functionality which can be overridden by variant or platform HALs. Architecture specific debugger functionality (handles single stepping, exception-to-signal conversion, etc.), exception/interrupt vector definitions and handlers, cache definition and control macros, context switching code, assembler functions for early system initialization, configuration options, and possibly tests. |
Variant HAL (hal/<<architecture>>/<<variant>>) ([http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/mpc8xx/current?cvsroot=ecos](http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/mpc8xx/current?cvsroot=ecos)) | Some CPU architectures consist of a number of variants, for example MIPS CPUs come in both 32 and 64 bit versions, and some variants have embedded features additional to the CPU core. Variant extensions to the architecture code (cache, exception/interrupt), configuration options, possibly drivers for variant on-core devices, and possibly tests. |
Platform HAL (hal/<<architecture>>/<<platform>>) ([http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/mbx/current?cvsroot=ecos](http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/mbx/current?cvsroot=ecos)) | Early platform initialization code, platform memory layout specification, configuration options (processor speed, compiler options), diagnostic IO functions, debugger IO functions, platform specific extensions to architecture or variant code (off-core interrupt controller), and possibly tests. |
Auxiliary HAL (hal/<<architecture>>/<<module>>) ([http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/quicc/current?cvsroot=ecos](http://ecos.sourceforge.org/cgi-bin/cvsweb.cgi/ecos/packages/hal/powerpc/quicc/current?cvsroot=ecos)) | Some variants share common modules on the core. Motorola’s PowerPC QUICC is an example of such a module. Module specific functionality (interrupt controller, simple device drivers), possibly tests. |

### File Descriptions

Listed below are the files found in various HALs, with a short description of what each file contains. When looking in existing HALs beware that they do not necessarily follow this naming scheme. If you are writing a new HAL, please try to follow it as closely as possible. Still, no two targets are the same, so sometimes it makes sense to use additional files.

**Common HAL**
### Chapter 6. Porting Guide

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/dbg-thread-syscall.h</td>
<td>Defines the thread debugging syscall function. This is used by the ROM monitor to access the thread debugging API in the RAM application.</td>
</tr>
<tr>
<td>include/dbg-threads-api.h</td>
<td>Defines the thread debugging API.</td>
</tr>
<tr>
<td>include/drv_api.h</td>
<td>Defines the driver API.</td>
</tr>
<tr>
<td>include/generic-stub.h</td>
<td>Defines the generic stub features.</td>
</tr>
<tr>
<td>include/hal_if.h</td>
<td>Defines the ROM/RAM calling interface API.</td>
</tr>
<tr>
<td>include/hal_misc.h</td>
<td>Defines miscellaneous helper functions shared by all HALs.</td>
</tr>
<tr>
<td>include/hal_stub.h</td>
<td>Defines eCos mappings of GDB stub features.</td>
</tr>
<tr>
<td>src/dbg-threads-syscall.c</td>
<td>Thread debugging implementation.</td>
</tr>
<tr>
<td>src/drv_api.c</td>
<td>Driver API implementation. Depending on configuration this provides either wrappers for the kernel API, or a minimal implementation of these features. This allows drivers to be written relying only on HAL features.</td>
</tr>
<tr>
<td>src/dummy.c</td>
<td>Empty dummy file ensuring creation of libtarget.a.</td>
</tr>
<tr>
<td>src/generic-stub.c</td>
<td>Generic GDB stub implementation. This provides the communication protocol used to communicate with GDB over a serial device or via the network.</td>
</tr>
<tr>
<td>src/hal_if.c</td>
<td>ROM/RAM calling interface implementation. Provides wrappers from the calling interface API to the eCos features used for the implementation.</td>
</tr>
<tr>
<td>src/hal_misc.c</td>
<td>Various helper functions shared by all platforms and architectures.</td>
</tr>
<tr>
<td>src/hal_stub.c</td>
<td>Wrappers from eCos HAL features to the features required by the generic GDB stub.</td>
</tr>
<tr>
<td>src/stubrom/stubrom.c</td>
<td>The file used to build eCos GDB stub images. Basically a cyg_start function with a hard coded breakpoint.</td>
</tr>
<tr>
<td>src/thread-packets.c</td>
<td>More thread debugging related functions.</td>
</tr>
<tr>
<td>src/thread-pkts.h</td>
<td>Defines more thread debugging related function.</td>
</tr>
</tbody>
</table>

### Architecture HAL

Some architecture HALs may add extra files for architecture specific serial drivers, or for handling interrupts and exceptions if it makes sense.

Note that many of the definitions in these files are only conditionally defined - if the equivalent variant or platform headers provide the definitions, those override the generic architecture definitions.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/arch.inc</td>
<td>Various assembly macros used during system initialization.</td>
</tr>
</tbody>
</table>
### File Description

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/basetype.h</td>
<td>Endian, label, alignment, and type size definitions. These override common defaults in CYGPKG_INFRA.</td>
</tr>
<tr>
<td>include/hal_arch.h</td>
<td>Saved register frame format, various thread, register and stack related macros.</td>
</tr>
<tr>
<td>include/hal_cache.h</td>
<td>Cache definitions and cache control macros.</td>
</tr>
<tr>
<td>include/hal_intr.h</td>
<td>Exception and interrupt definitions. Macros for configuring and controlling interrupts. eCos real-time clock control macros.</td>
</tr>
<tr>
<td>include/hal_io.h</td>
<td>Macros for accessing IO devices.</td>
</tr>
<tr>
<td>include/&lt;arch&gt;_regs.h</td>
<td>Architecture register definitions.</td>
</tr>
<tr>
<td>include/&lt;arch&gt;_stub.h</td>
<td>Architecture stub definitions. In particular the register frame layout used by GDB. This may differ from the one used by eCos.</td>
</tr>
<tr>
<td>include/&lt;arch&gt;.inc</td>
<td>Architecture convenience assembly macros.</td>
</tr>
<tr>
<td>src/&lt;arch&gt;.ld</td>
<td>Linker macros.</td>
</tr>
<tr>
<td>src/context.S</td>
<td>Functions handling context switching and setjmp/longjmp.</td>
</tr>
<tr>
<td>src/hal_misc.c</td>
<td>Exception and interrupt handlers in C. Various other utility functions.</td>
</tr>
<tr>
<td>src/hal_mk_defs.c</td>
<td>Used to export definitions from C header files to assembler header files.</td>
</tr>
<tr>
<td>src/hal_intr.c</td>
<td>Any necessary interrupt handling functions.</td>
</tr>
<tr>
<td>src/&lt;arch&gt;stub.c</td>
<td>Architecture stub code. Contains functions for translating eCos exceptions to UNIX signals and functions for single-stepping.</td>
</tr>
<tr>
<td>src/vectors.S</td>
<td>Exception, interrupt and early initialization code.</td>
</tr>
</tbody>
</table>

### Variant HAL

Some variant HALs may add extra files for variant specific serial drivers, or for handling interrupts/exceptions if it makes sense.

Note that these files may be mostly empty if the CPU variant can be controlled by the generic architecture macros. The definitions present are only conditionally defined - if the equivalent platform headers provide the definitions, those override the variant definitions.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/var_arch.h</td>
<td>Saved register frame format, various thread, register and stack related macros.</td>
</tr>
<tr>
<td>include/var_cache.h</td>
<td>Cache related macros.</td>
</tr>
<tr>
<td>include/var_intr.h</td>
<td>Interrupt related macros.</td>
</tr>
<tr>
<td>include/var_regs.h</td>
<td>Extra register definitions for the CPU variant.</td>
</tr>
<tr>
<td>include/variant.inc</td>
<td>Various assembly macros used during system initialization.</td>
</tr>
<tr>
<td>src/var_intr.c</td>
<td>Interrupt functions if necessary.</td>
</tr>
</tbody>
</table>
## Platform HAL

Extras files may be added for platform specific serial drivers. Extra files for handling interrupts and exceptions will be present if it makes sense.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/hal_diag.h</td>
<td>Defines functions used for HAL diagnostics output. This would normally be the ROM calling interface wrappers, but may also be the low-level IO functions themselves, saving a little overhead.</td>
</tr>
<tr>
<td>include/platform.inc</td>
<td>Platform initialization code. This includes memory controller, vectors, and monitor initialization. Depending on the architecture, other things may need defining here as well: interrupt decoding, status register initialization value, etc.</td>
</tr>
<tr>
<td>include/plf_cache.h</td>
<td>Platform specific cache handling.</td>
</tr>
<tr>
<td>include/plf_intr.h</td>
<td>Platform specific interrupt handling.</td>
</tr>
<tr>
<td>include/plf_io.h</td>
<td>PCI IO definitions and macros. May also be used to override generic HAL IO macros if the platform endianness differs from that of the CPU.</td>
</tr>
<tr>
<td>include/plf_stub.h</td>
<td>Defines stub initializer and board reset details.</td>
</tr>
<tr>
<td>src/hal_diag.c</td>
<td>May contain the low-level device drivers. But these may also reside in plf_stub.c</td>
</tr>
<tr>
<td>src/platform.S</td>
<td>Memory controller setup macro, and if necessary interrupt springboard code.</td>
</tr>
<tr>
<td>src/var_misc.c</td>
<td>hal_variant_init function and any necessary extra functions.</td>
</tr>
<tr>
<td>src/variant.S</td>
<td>Interrupt handler table definition.</td>
</tr>
<tr>
<td>src/&lt;arch&gt;_&lt;_variant&gt;.ld</td>
<td>Linker macros.</td>
</tr>
</tbody>
</table>

The platform HAL also contains files specifying the platform’s memory layout. These files are located in include/pkgconf.

## Auxiliary HAL

Auxiliary HALs contain whatever files are necessary to provide the required functionality. There are no pre-defined set of files required in an auxiliary HAL.
Chapter 6. Porting Guide

Virtual Vectors (eCos/ROM Monitor Calling Interface)

Virtually all eCos platforms provide full debugging capabilities via RedBoot. This environment contains not only debug stubs based on GDB, but also rich I/O support which can be exported to loaded programs. Such programs can take advantage of the I/O capabilities using a special ROM/RAM calling interface (also referred to as virtual vector table). eCos programs make use of the virtual vector mechanism implicitly. Non-eCos programs can access these functions using the support from the newlib library.

Virtual Vectors

What are virtual vectors, what do they do, and why are they needed?

"Virtual vectors" is the name of a table located at a static location in the target address space. This table contains 64 vectors that point to service functions or data.

The fact that the vectors are always placed at the same location in the address space means that both ROM and RAM startup configurations can access these and thus the services pointed to.

The primary goal is to allow services to be provided by ROM configurations (ROM monitors such as RedBoot in particular) with clients in RAM configurations being able to use these services.

Without the table of pointers this would be impossible since the ROM and RAM applications would be linked separately - in effect having separate name spaces - preventing direct references from one to the other.

This decoupling of service from client is needed by RedBoot, allowing among other things debugging of applications which do not contain debugging client code (stubs).

Initialization (or Mechanism vs. Policy)

Virtual vectors are a mechanism for decoupling services from clients in the address space.

The mechanism allows services to be implemented by a ROM monitor, a RAM application, to be switched out at run-time, to be disabled by installing pointers to dummy functions, etc.

The appropriate use of the mechanism is specified loosely by a policy. The general policy dictates that the vectors are initialized in whole by ROM monitors (built for ROM or RAM), or by stand-alone applications.

For configurations relying on a ROM monitor environment, the policy is to allow initialization on a service by service basis. The default is to initialize all services, except COMMS services since these are presumed to already be carrying a communication session to the debugger / console which was used for launching the application. This means that the bulk of the code gets tested in normal builds, and not just once in a blue moon when building new stubs or a ROM configuration.

The configuration options are written to comply with this policy by default, but can be overridden by the user if desired. Defaults are:

- For application development: the ROM monitor provides debugging and diagnostic IO services, the RAM application relies on these by default.
- For production systems: the application contains all the necessary services.

Pros and Cons of Virtual Vectors

There are pros and cons associated with the use of virtual vectors. We do believe that the pros generally outweigh the cons by a great margin, but there may be situations where the opposite is true.
The use of the services are implemented by way of macros, meaning that it is possible to circumvent the virtual vectors if desired. There is (as yet) no implementation for doing this, but it is possible.

Here is a list of pros and cons:

Pro: Allows debugging without including stubs

This is the primary reason for using virtual vectors. It allows the ROM monitor to provide most of the debugging infrastructure, requiring only the application to provide hooks for asynchronous debugger interrupts and for accessing kernel thread information.

Pro: Allows debugging to be initiated from arbitrary channel

While this is only true where the application does not actively override the debugging channel setup, it is a very nice feature during development. In particular it makes it possible to launch (and/or debug) applications via Ethernet even though the application configuration does not contain networking support.

Pro: Image smaller due to services being provided by ROM monitor

All service functions except HAL IO are included in the default configuration. But if these are all disabled the image for download will be a little smaller. Probably doesn’t matter much for regular development, but it is a worthwhile saving for the 20000 daily tests run in the Red Hat eCos test farm.

Con: The vectors add a layer of indirection, increasing application size and reducing performance.

The size increase is a fraction of what is required to implement the services. So for RAM configurations there is a net saving, while for ROM configurations there is a small overhead.

The performance loss means little for most of the services (of which the most commonly used is diagnostic IO which happens via polled routines anyway).

Con: The layer of indirection is another point of failure.

The concern primarily being that of vectors being trashed by rogue writes from bad code, causing a complete loss of the service and possibly a crash. But this does not differ much from a rogue write to anywhere else in the address space which could cause the same amount of mayhem. But it is arguably an additional point of failure for the service in question.

Con: All the indirection stuff makes it harder to bring a HAL up

This is a valid concern. However, seeing as most of the code in question is shared between all HALs and should remain unchanged over time, the risk of it being broken when a new HAL is being worked on should be minimal.

When starting a new port, be sure to implement the HAL IO drivers according to the scheme used in other drivers, and there should be no problem.

However, it is still possible to circumvent the vectors if they are suspect of causing problems: simply change the HAL_DIAG_INIT and HAL_DIAG_WRITE_CHAR macros to use the raw IO functions.

**Available services**

The `hal_if.h` file in the common HAL defines the complete list of available services. A few worth mentioning in particular:

- COMMS services. All HAL IO happens via the communication channels.
- uS delay. Fine granularity (busy wait) delay function.
Reset. Allows a software initiated reset of the board.

### The COMMS channels

As all HAL IO happens via the COMMS channels these deserve to be described in a little more detail. In particular the controls of where diagnostic output is routed and how it is treated to allow for display in debuggers.

#### Console and Debugging Channels

There are two COMMS channels - one for console IO and one for debugging IO. They can be individually configured to use any of the actual IO ports (serial or Ethernet) available on the platform.

The console channel is used for any IO initiated by calling the `diag_*()` functions. Note that these should only be used during development for debugging, assertion and possibly tracing messages. All proper IO should happen via proper devices. This means it should be possible to remove the HAL device drivers from production configurations where assertions are disabled.

The debugging channel is used for communication between the debugger and the stub which remotely controls the target for the debugger (the stub runs on the target). This usually happens via some protocol, encoding commands and replies in some suitable form.

Having two separate channels allows, e.g., for simple logging without conflicts with the debugger or interactive IO which some debuggers do not allow.

#### Mangling

As debuggers usually have a protocol using specialized commands when communicating with the stub on the target, sending out text as raw ASCII from the target on the same channel will either result in protocol errors (with loss of control over the target) or the text may just be ignored as junk by the debugger.

To get around this, some debuggers have a special command for text output. Mangling is the process of encoding diagnostic ASCII text output in the form specified by the debugger protocol.

When it is necessary to use mangling, i.e. when writing console output to the same port used for debugging, a mangler function is installed on the console channel which mangles the text and passes it on to the debugger channel.

#### Controlling the Console Channel

Console output configuration is either inherited from the ROM monitor launching the application, or it is specified by the application. This is controlled by the new option `CYGSEM_HAL_VIRTUAL_VECTOR_INHERIT_CONS` which defaults to enabled when the configuration is set to use a ROM monitor.

If the user wants to specify the console configuration in the application image, there are two new options that are used for this.

Defaults are to direct diagnostic output via a mangler to the debugging channel (`CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` enabled). The mangler type is controlled by the option `CYGSEM_HAL_DIAG_MANGLER`. At present there are only two mangler types:
Chapter 6. Porting Guide

GDB

This causes a mangler appropriate for debugging with GDB to be installed on the console channel.

None

This causes a NULL mangler to be installed on the console channel. It will redirect the IO to/from the debug channel without mangling of the data. This option differs from setting the console channel to the same IO port as the debugging channel in that it will keep redirecting data to the debugging channel even if that is changed to some other port.

Finally, by disabling CYGDBG_HAL_DIAG_TO_DEBUG_CHAN, the diagnostic output is directed in raw form to the specified console IO port.

In summary this results in the following common configuration scenarios for RAM startup configurations:

• For regular debugging with diagnostic output appearing in the debugger, mangling is enabled and stubs disabled.
  Diagnostic output appears via the debugging channel as initiated by the ROM monitor, allowing for correct behavior whether the application was launched via serial or Ethernet, from the RedBoot command line or from a debugger.

• For debugging with raw diagnostic output, mangling is disabled.
  Debugging session continues as initiated by the ROM monitor, whether the application was launched via serial or Ethernet. Diagnostic output is directed at the IO port configured in the application configuration.

  Note: There is one caveat to be aware of. If the application uses proper devices (be it serial or Ethernet) on the same ports as those used by the ROM monitor, the connections initiated by the ROM monitor will be terminated.

And for ROM startup configurations:

• Production configuration with raw output and no debugging features (configured for RAM or ROM), mangling is disabled, no stubs are included.
  Diagnostic output appears (in unmangled form) on the specified IO port.

• RedBoot configuration, includes debugging features and necessary mangling.
  Diagnostic and debugging output port is auto-selected by the first connection to any of the supported IO ports. Can change from interactive mode to debugging mode when a debugger is detected - when this happens a mangler will be installed as required.

• GDB stubs configuration (obsoleted by RedBoot configuration), includes debugging features, mangling is hardwired to GDB protocol.
  Diagnostic and debugging output is hardwired to configured IO ports, mangling is hardwired.

Footnote: Design Reasoning for Control of Console Channel

The current code for controlling the console channel is a replacement for an older implementation which had some shortcomings which addressed by the new implementation.

This is what the old implementation did: on initialization it would check if the CDL configured console channel differed from the active debug channel - and if so, set the console channel, thereby disabling mangling.
The idea was that whatever channel was configured to be used for console (i.e., diagnostic output) in the application was what should be used. Also, it meant that if debug and console channels were normally the same, a changed console channel would imply a request for unmangled output.

But this prevented at least two things:

- It was impossible to inherit the existing connection by which the application was launched (either by Red-Boot commands via telnet, or by via a debugger).
  This was mostly a problem on targets supporting Ethernet access since the diagnostic output would not be returned via the Ethernet connection, but on the configured serial port.
  The problem also occurred on any targets with multiple serial ports where the ROM monitor was configured to use a different port than the CDL defaults.
- Proper control of when to mangle or just write out raw ASCII text.
  Sometimes it’s desirable to disable mangling, even if the channel specified is the same as that used for debugging. This usually happens if GDB is used to download the application, but direct interaction with the application on the same channel is desired (GDB protocol only allows output from the target, no input).

### The calling Interface API

The calling interface API is defined by hal_if.h and hal_if.c in hal/common.

The API provides a set of services. Different platforms, or different versions of the ROM monitor for a single platform, may implement fewer or extra service. The table has room for growth, and any entries which are not supported map to a NOP-service (when called it returns 0 (false)).

A client of a service should either be selected by configuration, or have suitable fall back alternatives in case the feature is not implemented by the ROM monitor.

**Note:** Checking for unimplemented service when this may be a data field/pointer instead of a function: suggest reserving the last entry in the table as the NOP-service pointer. Then clients can compare a service entry with this pointer to determine whether it’s initialized or not.

The header file cyg/hal/hal_if.h defines the table layout and accessor macros (allowing primitive type checking and alternative implementations should it become necessary).

The source file hal_if.c defines the table initialization function. All HALs should call this during platform initialization - the table will get initialized according to configuration. Also defined here are wrapper functions which map between the calling interface API and the API of the used eCos functions.

### Implemented Services

This is a brief description of the services, some of which are described in further detail below.

**VERSION**

Version of table. Serves as a way to check for how many features are available in the table. This is the index of the last service in the table.
KILL VECTOR

[Presently unused by the stub code, but initialized] This vector defines a function to execute when the system receives a kill signal from the debugger. It is initialized with the reset function (see below), but the application (or eCos) can override it if necessary.

CONSOLE_PROCS

The communication procedure table used for console IO (see the Section called IO channels).

DEBUG_PROCS

The communication procedure table used for debugger IO (see the Section called IO channels).

FLUSH_DCACHE

Flushes the data cache for the specified region. Some implementations may flush the entire data cache.

FLUSH_ICACHE

Flushes (invalidates) the instruction cache for the specified region. Some implementations may flush the entire instruction cache.

SET_DEBUG_COMM

Change debugging communication channel.

SET_CONSOLE_COMM

Change console communication channel.

DBG_SYSCALL

Vector used to communication between debugger functions in ROM and in RAM. RAM eCos configurations may install a function pointer here which the ROM monitor uses to get thread information from the kernel running in RAM.

RESET

Resets the board on call. If it is not possible to reset the board from software, it will jump to the ROM entry point which will perform a "software" reset of the board.

CONSOLE_INTERRUPT_FLAG

Set if a debugger interrupt request was detected while processing console IO. Allows the actual breakpoint action to be handled after return to RAM, ensuring proper backtraces etc.

DELAY_US

Will delay the specified number of microseconds. The precision is platform dependent to some extend - a small value (<100us) is likely to cause bigger delays than requested.

FLASH_CFG_OP

For accessing configuration settings kept in flash memory.

INSTALL_BPT_FN

Installs a breakpoint at the specified address. This is used by the asynchronous breakpoint support (see ).
Chapter 6. Porting Guide

Compatibility

When a platform is changed to support the calling interface, applications will use it if so configured. That means that if an application is run on a platform with an older ROM monitor, the service is almost guaranteed to fail.

For this reason, applications should only use Console Comm for HAL diagnostics output if explicitly configured to do so (CYGSEM_HAL_VIRTUAL_VECTOR_DIAG).

As for asynchronous GDB interrupts, the service will always be used. This is likely to cause a crash under older ROM monitors, but this crash may be caught by the debugger. The old workaround still applies: if you need asynchronous breakpoints or thread debugging under older ROM monitors, you may have to include the debugging support when configuring eCos.

Implementation details

During the startup of a ROM monitor, the calling table will be initialized. This also happens if eCos is configured not to rely on a ROM monitor.

**Note**: There is reserved space (256 bytes) for the vector table whether it gets used or not. This may be something that we want to change if we ever have to shave off every last byte for a given target.

If thread debugging features are enabled, the function for accessing the thread information gets registered in the table during startup of a RAM startup configuration.

Further implementation details are described where the service itself is described.

New Platform Ports

The `hal_platform_init()` function must call `hal_if_init()`.

The HAL serial driver must, when called via `cyg_hal_plf_comms_init()` must initialize the communication channels.

The `reset()` function defined in `hal_if.c` will attempt to do a hardware reset, but if this fails it will fall back to simply jumping to the reset entry-point. On most platforms the startup initialization will go a long way to reset the target to a sane state (there will be exceptions, of course). For this reason, make sure to define `HAL_STUB_PLATFORM_RESET_ENTRY` in `plf_stub.h`.

All debugging features must be in place in order for the debugging services to be functional. See general platform porting notes.

New architecture ports

There are no specific requirements for a new architecture port in order to support the calling interface, but the basic debugging features must be in place. See general architecture porting notes.

IO channels

The calling interface provides procedure tables for all IO channels on the platform. These are used for console (diagnostic) and debugger IO, allowing a ROM monitor to provided all the needed IO routines. At the same...
time, this makes it easy to switch console/debugger channels at run-time (the old implementation had hardwired
drivers for console and debugger IO, preventing these to change at run-time).
The hal_if provides wrappers which interface these services to the eCos infrastructure diagnostics routines.
This is done in a way which ensures proper string mangling of the diagnostics output when required (e.g.
O-packetization when using a GDB compatible ROM monitor).

Available Procedures
This is a brief description of the procedures

CH_DATA
   Pointer to the controller IO base (or a pointer to a per-device structure if more data than the IO base is
   required). All the procedures below are called with this data item as the first argument.

WRITE
   Writes the buffer to the device.

READ
   Fills a buffer from the device.

PUTC
   Write a character to the device.

GETC
   Read a character from the device.

CONTROL
   Device feature control. Second argument specifies function:

   SETBAUD
   Changes baud rate.

   GETBAUD
   Returns the current baud rate.

   INSTALL_DBG_ISR
   [Unused]

   REMOVE_DBG_ISR
   [Unused]

   IRQ_DISABLE
   Disable debugging receive interrupts on the device.

   IRQ_ENABLE
   Enable debugging receive interrupts on the device.
Chapter 6. Porting Guide

DBG_ISR_VECTOR

Returns the ISR vector used by the device for debugging receive interrupts.

SET_TIMEOUT

Set GETC timeout in milliseconds.

FLUSH_OUTPUT

Forces driver to flush data in its buffers. Note that this may not affect hardware buffers (e.g. FIFOs).

DBG_ISR

ISR used to handle receive interrupts from the device (see ).

GETC_TIMEOUT

Read a character from the device with timeout.

Usage

The standard eCos diagnostics IO functions use the channel procedure table when CYGSEM_HAL_VIRTUAL_VECTOR_DIAG is enabled. That means that when you use diag_printf (or the libc printf function) the stream goes through the selected console procedure table. If you use the virtual vector function SET_CONSOLE_COMM you can change the device which the diagnostics output goes to at run-time.

You can also use the table functions directly if desired (regardless of the CYGSEM_HAL_VIRTUAL_VECTOR_DIAG setting - assuming the ROM monitor provides the services). Here is a small example which changes the console to use channel 2, fetches the comm procs pointer and calls the write function from that table, then restores the console to the original channel:

```c
#define T "Hello World!\n"

int
main(void)
{
    hal_virtual_comm_table_t* comm;
    int cur = CYGACC_CALL_IF_SET_CONSOLE_COMM(CYGNUM_CALL_IF_SET_COMM_ID_QUERY_CURRENT);
    CYGACC_CALL_IF_SET_CONSOLE_COMM(2);
    comm = CYGACC_CALL_IF_CONSOLE_PROCS();
    CYGACC_COMM_IF_WRITE(*comm, T, strlen(T));
    CYGACC_CALL_IF_SET_CONSOLE_COMM(cur);
}
```

Beware that if doing something like the above, you should only do it to a channel which does not have GDB at the other end: GDB ignores raw data, so you would not see the output.

Compatibility

The use of this service is controlled by the option CYGSEM_HAL_VIRTUAL_VECTOR_DIAG which is disabled per default on most older platforms (thus preserving backwards compatibility with older stubs). On newer ports, this option should always be set.
Implementation Details

There is an array of procedure tables (raw comm channels) for each IO device of the platform which get initialized by the ROM monitor, or optionally by a RAM startup configuration (allowing the RAM configuration to take full control of the target). In addition to this, there’s a special table which is used to hold mangler procedures.

The vector table defines which of these channels are selected for console and debugging IO respectively: console entry can be empty, point to mangler channel, or point to a raw channel. The debugger entry should always point to a raw channel.

During normal console output (i.e., diagnostic output) the console table will be used to handle IO if defined. If not defined, the debug table will be used.

This means that debuggers (such as GDB) which require text streams to be mangled (O-packetized in the case of GDB), can rely on the ROM monitor install mangling IO routines in the special mangler table and select this for console output. The mangler will pass the mangled data on to the selected debugging channel.

If the eCos configuration specifies a different console channel from that used by the debugger, the console entry will point to the selected raw channel, thus overriding any mangler provided by the ROM monitor.

See hal_if_diag_* routines in hal_if.c for more details of the stream path of diagnostic output. See cyg_hal_gdb_diag_*() routines in hal_stub.c for the mangler used for GDB communication.

New Platform Ports

Define CDL options CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS, CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL, and CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL.

If CYGSEM_HAL_VIRTUAL_VECTOR_DIAG is set, make sure the infra diag code uses the hal_if diag functions:

```c
#define HAL_DIAG_INIT() hal_if_diag_init()
#define HAL_DIAG_WRITE_CHAR(_c_) hal_if_diag_write_char(_c_)
#define HAL_DIAG_READ_CHAR(_c_) hal_if_diag_read_char(&_c_)
```

In addition to the above functions, the platform HAL must also provide a function cyg_hal_plf_comms_init which initializes the drivers and the channel procedure tables.

Most of the other functionality in the table is more or less possible to copy unchanged from existing ports. Some care is necessary though to ensure the proper handling of interrupt vectors and timeouts for various devices handled by the same driver. See PowerPC/Cogent platform HAL for an example implementation.

**Note::** When vector table console code is not used, the platform HAL must map the HAL_DIAG_INIT, HAL_DIAG_WRITE_CHAR and HAL_DIAG_READ_CHAR macros directly to the low-level IO functions, hardwired to use a compile-time configured channel.

**Note::** On old ports the hardwired HAL_DIAG_INIT, HAL_DIAG_WRITE_CHAR and HAL_DIAG_READ_CHAR implementations will also contain code to O-packetize the output for GDB. This should not be adopted for new ports! On new ports the ROM monitor is guaranteed to provide the necessary mangling via the vector table. The hardwired configuration should be reserved for ROM startups where achieving minimal image size is crucial.
HAL Coding Conventions

To get changes and larger submissions included into the eCos source repository, we ask that you adhere to a set of coding conventions. The conventions are defined as an attempt to make a consistent tree. Consistency makes it easier for people to read, understand and maintain the code, which is important when many people work on the same project.

The below is only a brief, and probably incomplete, summary of the rules. Please look through files in the area where you are making changes to get a feel for any additional conventions. Also feel free to ask on the list if you have specific questions.

Implementation issues

There are a few implementation issues that should be kept in mind:

HALs

HALs must be written in C and assembly only. C++ must not be used. This is in part to keep the HALs simple since this is usually the first part of eCos a newcomer will see, and in part to maintain the existing de facto standard.

IO access

Use HAL IO access macros for code that might be reused on different platforms than the one you are writing it for.

MMU

If it is necessary to use the MMU (e.g., to prevent caching of IO areas), use a simple 1-1 mapping of memory if possible. On most platforms where using the MMU is necessary, it will be possible to achieve the 1-1 mapping using the MMU’s provision for mapping large continuous areas (hardwired TLBs or BATs). This reduces the footprint (no MMU table) and avoids execution overhead (no MMU-related exceptions).

Assertions

The code should contain assertions to validate argument values, state information and any assumptions the code may be making. Assertions are not enabled in production builds, so liberally sprinkling assertions throughout the code is good.

Testing

The ability to test your code is very important. In general, do not add new code to the eCos runtime unless you also add a new test to exercise that code. The test also serves as an example of how to use the new code.

Source code details

Line length

Keep line length below 78 columns whenever possible.

Comments

Whenever possible, use // comments instead of /***/.
Indentation

Use spaces instead of TABs. Indentation level is 4. Braces start on the same line as the expression. See below for emcs mode details.

```
;;------------------------------------------------------------------------
;; eCos C/C++ mode Setup.
;;
;; bsd mode: indent = 4
;; tail comments are at col 40.
;; uses spaces not tabs in C

(defun ecos-c-mode ()
  "C mode with adjusted defaults for use with the eCos sources."
  (interactive)
  (c++-mode)
  (c-set-style "bsd")
  (setq comment-column 40)
  (setq indent-tabs-mode nil)
  (show-paren-mode 1)
  (setq c-basic-offset 4)

  (set-variable 'add-log-full-name "Your Name")
  (set-variable 'add-log-mailing-address "Your email address")
)

(defun ecos-asm-mode ()
  "ASM mode with adjusted defaults for use with the eCos sources."
  (interactive)
  (setq comment-column 40)
  (setq indent-tabs-mode nil)
  (asm-mode)
  (setq c-basic-offset 4)

  (set-variable 'add-log-full-name "Your Name")
  (set-variable 'add-log-mailing-address "Your email address")
)

(setq auto-mode-alist
  (append '('/local/ecc/.*\.C$" . ecos-c-mode)
           ('/local/ecc/.*\.cc$" . ecos-c-mode)
           ('/local/ecc/.*\.cpp$" . ecos-c-mode)
           ('/local/ecc/.*\.inl$" . ecos-c-mode)
           ('/local/ecc/.*\.c$" . ecos-c-mode)
           ('/local/ecc/.*\.h$" . ecos-c-mode)
           ('/local/ecc/.*\.S$" . ecos-asm-mode)
           ('/local/ecc/.*\.inc$" . ecos-asm-mode)
           ('/local/ecc/.*\.cdl$" . tcl-mode)
         )
  auto-mode-alist))
```

Nested Headers

In order to allow platforms to define all necessary details, while still maintaining the ability to share code between common platforms, all HAL headers are included in a nested fashion.

The architecture header (usually `hal_XXX.h`) includes the variant equivalent of the header (`var_XXX.h`) which in turn includes the platform equivalent of the header (`plf_XXX.h`).

All definitions that may need to be overridden by a platform are then only conditionally defined, depending on whether a lower layer has already made the definition:

```
hal_intr.h: #include <var_intr.h>
```
#ifndef MACRO_DEFINED
# define MACRO ...
# define MACRO_DEFINED
#endif

var_intr.h: #include <plf_intr.h>

#ifndef MACRO_DEFINED
#define MACRO ...
#define MACRO_DEFINED
#endif

plf_intr.h:

#define MACRO ...
#define MACRO_DEFINED

This means a platform can opt to rely on the variant or architecture implementation of a feature, or implement it itself.

**Platform HAL Porting**

This is the type of port that takes the least effort. It basically consists of describing the platform (board) for the HAL: memory layout, early platform initialization, interrupt controllers, and a simple serial device driver.

Doing a platform port requires a preexisting architecture and possibly a variant HAL port.

**HAL Platform Porting Process**

**Brief overview**

The easiest way to make a new platform HAL is simply to copy an existing platform HAL of the same architecture/variant and change all the files to match the new one. In case this is the first platform for the architecture/variant, a platform HAL from another architecture should be used as a template.

The best way to start a platform port is to concentrate on getting RedBoot to run. RedBoot is a simpler environment than full eCos, it does not use interrupts or threads, but covers most of the basic startup requirements.

RedBoot normally runs out of FLASH or ROM and provides program loading and debugging facilities. This allows further HAL development to happen using RAM startup configurations, which is desirable for the simple reason that downloading an image which you need to test is often many times faster than either updating a flash part, or indeed, erasing and reprogramming an EPROM.

There are two approaches to getting to this first goal:

1. The board is equipped with a ROM monitor which allows "load and go" of ELF, binary, S-record or some other image type which can be created using objcopy. This allows you to develop RedBoot by downloading and running the code (saving time).
When the stub is running it is a good idea to examine the various hardware registers to help you write the platform initialization code.

Then you may have to fiddle a bit going through step two (getting it to run from ROM startup). If at all possible, preserve the original ROM monitor so you can revert to it if necessary.

2. The board has no ROM monitor. You need to get the platform initialization and stub working by repeatedly making changes, updating flash or EPROM and testing the changes. If you are lucky, you have a JTAG or similar CPU debugger to help you. If not, you will probably learn to appreciate LEDs. This approach may also be needed during the initial phase of moving RedBoot from RAM startup to ROM, since it is very unlikely to work first time.

Step-by-step

Given that no two platforms are exactly the same, you may have to deviate from the below. Also, you should expect a fair amount of fiddling - things almost never go right the first time. See the hints section below for some suggestions that might help debugging.

The description below is based on the HAL layout used in the MIPS, PC and MN10300 HALs. Eventually all HALs should be converted to look like these - but in a transition period there will be other HALs which look substantially different. Please try to adhere to the following as much is possible without causing yourself too much grief integrating with a HAL which does not follow this layout.

Minimal requirements

These are the changes you must make before you attempt to build RedBoot. You are advised to read all the sources though.

1. Copy an existing platform HAL from the same or another architecture. Rename the files as necessary to follow the standard: CDL and MLT related files should contain the `<arch>_<variant>_<platform>` triplet.

2. Adjust CDL options. Primarily option naming, real-time clock/counter, and CYGHWR_MEMORY_LAYOUT variables, but also other options may need editing. Look through the architecture/variant CDL files to see if there are any requirements/features which where not used on the platform you copied. If so, add appropriate ones. See the Section called HAL Platform CDL for more details.

3. Add the necessary packages and target descriptions to the top-level `ecos.db` file. See the Section called eCos Database. Initially, the target entry should only contain the HAL packages. Other hardware support packages will be added later.

4. Adjust the MLT files in `include/pkgconf` to match the memory layout on the platform. For initial testing it should be enough to just hand edit .h and .ldi files, but eventually you should generate all files using the memory layout editor in the configuration tool. See the Section called Platform Memory Layout for more details.

5. Edit the `misc/redboot_<STARTUP>.ecm` for the startup type you have chosen to begin with. Rename any platform specific options and remove any that do not apply. In the `cdl_configuration` section, comment out any extra packages that are added, particularly packages such as CYGPKG_IO_FLASH and CYGPKG_IO_ETH_DRIVERS. These are not needed for initial porting and will be added back later.

6. If the default IO macros are not correct, override them in `plf_io.h`. This may be necessary if the platform uses a different endianness from the default for the CPU.

7. Leave out/comment out code that enables caches and/or MMU if possible. Execution speed will not be a concern until the port is feature complete.
Chapter 6. Porting Guide

8. Implement a simple serial driver (polled mode only). Make sure the initialization function properly hooks the procedures up in the virtual vector IO channel tables. RedBoot will call the serial driver via these tables.

By copying an existing platform HAL most of this code will be already done, and will only need the platform specific hardware access code to be written.

9. Adjust/implement necessary platform initialization. This can be found in platform.inc and platform.S files (ARM: hal_platform_setup.h and <platform>_misc.c, PowerPC: <platform>.S). This step can be postponed if you are doing a RAM startup RedBoot first and the existing ROM monitor handles board initialization.

10. Define HAL_STUB_PLATFORM_RESET (optionally empty) and HAL_STUB_PLATFORM_RESET_ENTRY so that RedBoot can reset-on-detach - this is very handy, often removing the need for physically resetting the board between downloads.

You should now be able to build RedBoot. For ROM startup:

```
% ecosconfig new <target_name> redboot
% ecosconfig import $(ECOS_REPOSITORY)/hal/<architecture>/<platform>/<version>/misc/redboot_ROM.ecm
% ecosconfig tree
% make
```

You may have to make further changes than suggested above to get the make command to succeed. But when it does, you should find a RedBoot image in install/bin. To program this image into flash or EPROM, you may need to convert to some other file type, and possibly adjust the start address. When you have the correct objcopy command to do this, add it to the CYGBLD_BUILD_GDB_STUBS custom build rule in the platform CDL file.

Having updated the flash/EPROM on the board, you should see output on the serial port looking like this when powering on the board:

```
RedBoot (tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:42:24, Mar 14 2002

Platform: <PLATFORM> (<ARCHITECTURE> <VARIANT>)

RAM: 0x00000000-0x01000000, 0x000293e8-0x00ed1000 available
FLASH: 0x24000000 - 0x26000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

If you do not see this output, you need to go through all your changes and figure out what’s wrong. If there’s a user programmable LED or LCD on the board it may help you figure out how far RedBoot gets before it hangs. Unfortunately there’s no good way to describe what to do in this situation - other than that you have to play with the code and the board.

Adding features

Now you should have a basic RedBoot running on the board. This means you have a the correct board initialization and a working serial driver. It’s time to flesh out the remaining HAL features.

1. Reset. As mentioned above it is desirable to get the board to reset when GDB disconnects. When GDB disconnects it sends RedBoot a kill-packet, and RedBoot first calls HAL_STUB_PLATFORM_RESET(), attempting to perform a software-invoked reset. Most embedded CPUs/boards have a watchdog which is capable of triggering a reset. If your target does not have a watchdog, leave HAL_STUB_PLATFORM_RESET() empty and rely on the fallback approach.
If `HAL_STUB_PLATFORM_RESET()` did not cause a reset, RedBoot will jump to `HAL_STUB_PLATFORM_RESET_ENTRY` - this should be the address where the CPU will start execution after a reset. Re-initializing the board and drivers will usually be good enough to make a hardware reset unnecessary.

After the reset caused by the kill-packet, the target will be ready for GDB to connect again. During a days work, this will save you from pressing the reset button many times.

Note that it is possible to disconnect from the board without causing it to reset by using the GDB command "detach".

2. Single-stepping is necessary for both instruction-level debugging and for breakpoint support. Single-stepping support should already be in place as part of the architecture/variant HAL, but you want to give it a quick test since you will come to rely on it.

3. Real-time clock interrupts drive the eCos scheduler clock. Many embedded CPUs have an on-core timer (e.g. SH) or decrementer (e.g. MIPS, PPC) that can be used, and in this case it will already be supported by the architecture/variant HAL. You only have to calculate and enter the proper `CYGNUM_HAL_RTC_CONSTANTS` definitions in the platform CDL file.

On some targets it may be necessary to use a platform-specific timer source for driving the real-time clock. In this case you also have to enter the proper CDL definitions, but must also define suitable versions of the `HAL_CLOCK_XXXX` macros.

4. Interrupt decoding usually differs between platforms because the number and type of devices on the board differ. In `plf_intr.h` (ARM: `hal_platform_ints.h`) you must either extend or replace the default vector definitions provided by the architecture or variant interrupt headers. You may also have to define `HAL_INTERRUPT_XXXX` control macros.

5. Caching may also differ from architecture/variant definitions. This may be just the cache sizes, but there can also be bigger differences for example if the platform supports 2nd level caches.

When cache definitions are in place, enable the caches on startup. First verify that the system is stable for RAM startups, then build a new RedBoot and install it. This will test if caching, and in particular the cache sync/flush operations, also work for ROM startup.

6. Asynchronous breakpoints allow you to stop application execution and enter the debugger. Asynchronous breakpoint details are described in .

You should now have a completed platform HAL port. Verify its stability and completeness by running all the eCos tests and fix any problems that show up (you have a working RedBoot now, remember! That means you can debug the code to see why it fails).

Given the many configuration options in eCos, there may be hidden bugs or missing features that do not show up even if you run all the tests successfully with a default configuration. A comprehensive test of the entire system will take many configuration permutations and many many thousands of tests executed.

**Hints**

- JTAG or similar CPU debugging hardware can greatly reduce the time it takes to write a HAL port since you always have full visibility of what the CPU is doing.

- LEDs can be your friends if you don’t have a JTAG device. Especially in the start of the porting effort if you don’t already have a working ROM monitor on the target. Then you have to get a basic RedBoot working while basically being blindfolded. The LED can make it little easier, as you’ll be able to do limited tracking of program flow and behavior by switching the LED on and off. If the board has multiple LEDs you can show
Chapter 6. Porting Guide

A number (using binary notation with the LEDs) and sprinkle code which sets different numbers throughout the code.

- Debugging the interrupt processing is possible if you are careful with the way you program the very early interrupt entry handling. Write it so that as soon as possible in the interrupt path, taking a trap (exception) does not harm execution. See the SH vectors.S code for an example. Look for `cyg_hal_default_interrupt_vsr` and the label `cyg_hal_default_interrupt_vsr_bp_safe`, which marks the point after which traps/single-stepping is safe.

Being able to display memory content, CPU registers, interrupt controller details at the time of an interrupt can save a lot of time.

- Using assertions is a good idea. They can sometimes reveal subtle bugs or missing features long before you would otherwise have found them, let alone notice them.

The default eCos configuration does not use assertions, so you have to enable them by switching on the option `CYGPKG_INFRA_DEBUG` in the `infra` package.

- The idle loop can be used to help debug the system.

  Triggering clock from the idle loop is a neat trick for examining system behavior either before interrupts are fully working, or to speed up "the clock".

  Use the idle loop to monitor and/or print out variables or hardware registers.

- `hal_mk_defs` is used in some of the HALs (ARM, SH) as a way to generate assembler symbol definitions from C header files without imposing an assembler/C syntax separation in the C header files.

HAL Platform CDL

The platform CDL both contains details necessary for the building of eCos, and platform-specific configuration options. For this reason the options differ between platforms, and the below is just a brief description of the most common options.

See Components Writers Guide for more details on CDL. Also have a quick look around in existing platform CDL files to get an idea of what is possible and how various configuration issues can be represented with CDL.

eCos Database

The eCos configuration system is made aware of a package by adding a package description in `ecos.db`. As an example we use the `TX39/JMR3904` platform:

```plaintext
package CYGPKG_HAL_MIPS_TX39_JMR3904 {
  alias { "Toshiba JMR-TX3904 board" hal_tx39_jmr3904 tx39_jmr3904_hal }
  directory hal/mips/jmr3904
  script hal_mips_tx39_jmr3904.cdl
  hardware
  description "The JMR3904 HAL package should be used when targeting the actual hardware. The same package can also be used when running on the full simulator, since this provides an accurate simulation of the hardware including I/O devices. To use the simulator in this mode the command 'target sim --board=jmr3904' should be used from inside gdb."}
```
Chapter 6. Porting Guide

This contains the title and description presented in the Configuration Tool when the package is selected. It also specifies where in the tree the package files can be found (directory) and the name of the CDL file which contains the package details (script).

To be able to build and test a configuration for the new target, there also needs to be a target entry in the ecos.db file.

target jmr3904 {
    alias { "Toshiba JMR-TX3904 board" jmr tx39 }
    packages { CYGPKG_HAL_MIPS
                    CYGPKG_HAL_MIPS_TX39
                    CYGPKG_HAL_MIPS_TX39_JMR3904
                }
    description "The jmr3904 target provides the packages needed to run
eCos on a Toshiba JMR-TX3904 board. This target can also
be used when running in the full simulator, since the simulator provides an
accurate simulation of the hardware including I/O devices.
To use the simulator in this mode the command
'target sim --board=jmr3904' should be used from inside gdb."
}

The important part here is the packages section which defines the various hardware specific packages that contribute to support for this target. In this case the MIPS architecture package, the TX39 variant package, and the JMR-TX3904 platform packages are selected. Other packages, for serial drivers, ethernet drivers and FLASH memory drivers may also appear here.

CDL File Layout

All the platform options are contained in a CDL package named CYGPKG_HAL_<architecture>_<variant>_<platform>. They all share more or less the same cdl_package details:

cdl_package CYGPKG_HAL_MIPS_TX39_JMR3904 {
    display  "JMR3904 evaluation board"
    parent CYGPKG_HAL_MIPS
    requires CYGPKG_HAL_MIPS_TX39
    define_header hal_mips_tx39_jmr3904.h
    include_dir cyg/hal
    description "The JMR3904 HAL package should be used when targeting the
actual hardware. The same package can also be used when
running on the full simulator, since this provides an
accurate simulation of the hardware including I/O devices.
To use the simulator in this mode the command
'target sim --board=jmr3904' should be used from inside gdb."
    compile platform.S plf_misc.c plf_stub.c
    define_proc {
        puts $::cdl_system_header "#define CYGBLD_HAL_TARGET_H <pkgconf/hal_mips_tx39.h>"
        puts $::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H <pkgconf/hal_mips_tx39_jmr3904.h>"
    }
}

...
This specifies that the platform package should be parented under the MIPS packages, requires the TX39 variant HAL and all configuration settings should be saved in cyg/hal/hal_mips_tx39_jmt3904.h.

The compile line specifies which files should be built when this package is enabled, and the define_proc defines some macros that are used to access the variant or architecture (the _TARGET_ name is a bit of a misnomer) and platform configuration options.

**Startup Type**

eCos uses an option to select between a set of valid startup configurations. These are normally RAM, ROM and possibly ROMRAM. This setting is used to select which linker map to use (i.e., where to link eCos and the application in the memory space), and how the startup code should behave.

```c
cdl_component CYG_HAL_STARTUP {
    display "Startup type"
    flavor data
    legal_values {"RAM" "ROM"}
    default_value {"RAM"}
    no_define
    define -file system.h CYG_HAL_STARTUP
description "When targeting the JMR3904 board it is possible to build the system for either RAM bootstrap, ROM bootstrap, or STUB bootstrap. RAM bootstrap generally requires that the board is equipped with ROMs containing a suitable ROM monitor or equivalent software that allows GDB to download the eCos application on to the board. The ROM bootstrap typically requires that the eCos application be blown into EPROMs or equivalent technology."
}
```

The no_define and define pair is used to make the setting of this option appear in the file system.h instead of the default specified in the header.

**Build options**

A set of options under the components CYGBLD_GLOBAL_OPTIONS and CYGHWR_MEMORY_LAYOUT specify how eCos should be built: what tools and compiler options should be used, and which linker fragments should be used.

```c
cdl_component CYGBLD_GLOBAL_OPTIONS {
    display "Global build options"
    flavor none
    parent CYGPKG_NONE
    description "Global build options including control over compiler flags, linker flags and choice of toolchain."
}

cdl_option CYGBLD_GLOBAL_COMMAND_PREFIX {
    display "Global command prefix"
    flavor data
    no_define
    default_value {"mips-tx39-elf"}
    description "This option specifies the command prefix used when invoking the build tools."
```
Chapter 6. Porting Guide

```plaintext
cdl_option CYGBLD_GLOBAL_CFLAGS {
  display "Global compiler flags"
  flavor data
  no_define
default_value { 
  }
description "This option controls the global compiler flags which are used to compile all packages by default. Individual packages may define options which override these global flags."
}

cdl_option CYGBLD_GLOBAL_LDFLAGS {
  display "Global linker flags"
  flavor data
  no_define
default_value { 
    "-g -nostdlib -Wl,--gc-sections -Wl,-static"
  }
description "This option controls the global linker flags. Individual packages may define options which override these global flags."
}

cdl_component CYGHWR_MEMORY_LAYOUT {
  display "Memory layout"
  flavor data
  no_define
calculated { CYG_HAL_STARTUP == "RAM" ? "mips_tx39_jmr3904_ram" : "mips_tx39_jmr3904_rom" }

cdl_option CYGHWR_MEMORY_LAYOUT_LDI {
  display "Memory layout linker script fragment"
  flavor data
  no_define
define -file system.h CYGHWR_MEMORY_LAYOUT_LDI
calculated { CYG_HAL_STARTUP == "RAM" ? "<pkgconf/mlt_mips_tx39_jmr3904_ram.ldi>" : "<pkgconf/mlt_mips_tx39_jmr3904_rom.ldi>" }
}

cdl_option CYGHWR_MEMORY_LAYOUT_H {
  display "Memory layout header file"
  flavor data
  no_define
define -file system.h CYGHWR_MEMORY_LAYOUT_H
calculated { CYG_HAL_STARTUP == "RAM" ? "<pkgconf/mlt_mips_tx39_jmr3904_ram.h>" : "<pkgconf/mlt_mips_tx39_jmr3904_rom.h>" }
}
```

**Common Target Options**

All platforms also specify real-time clock details:

```plaintext
# Real-time clock/counter specifics
cdl_component CYGNUM_HAL_RTC_CONSTANTS {
  display "Real-time clock constants."
  flavor none
}
```
Chapter 6. Porting Guide

cdl_option CYGNUM_HAL_RTC_NUMERATOR {
    display "Real-time clock numerator"
    flavor data
    calculated 1000000000
}
cdl_option CYGNUM_HAL_RTC_DENOMINATOR {
    display "Real-time clock denominator"
    flavor data
    calculated 100
}
# Isn't a nice way to handle freq requirement!
cdl_option CYGNUM_HAL_RTC_PERIOD {
    display "Real-time clock period"
    flavor data
    legal_values { 15360 20736 }
    calculated { CYGHWR_HAL_MIPS_CPU_FREQ == 50 ? 15360 : \ 
                   CYGHWR_HAL_MIPS_CPU_FREQ == 66 ? 20736 : 0 }
}

The NUMERATOR divided by the DENOMINATOR gives the number of nanoseconds per tick. The PERIOD is the
divider to be programmed into a hardware timer that is driven from an appropriate hardware clock, such that
the timer overflows once per tick (normally generating a CPU interrupt to mark the end of a tick). The tick
default rate is typically 100Hz.

Platforms that make use of the virtual vector ROM calling interface (see the Section called Virtual Vectors
(eCos/ROM Monitor Calling Interface)) will also specify details necessary to define configuration channels
(these options are from the SH/EDK7707 HAL):

cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS {
    display "Number of communication channels on the board"
    flavor data
    calculated 1
}
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL {
    display "Debug serial port"
    flavor data
    legal_values 0 to CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS-1
    default_value 0
    description "The EDK/7708 board has only one serial port. This option
                 chooses which port will be used to connect to a host
                 running GDB."
}
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL {
    display "Diagnostic serial port"
    flavor data
    legal_values 0 to CYGNUM_HAL_VIRTUALVECTOR_COMM_CHANNELS-1
    default_value 0
    description "The EDK/7708 board has only one serial port. This option
               chooses which port will be used for diagnostic output."
}

The platform usually also specify an option controlling the ability to co-exist with a ROM monitor:

cdl_option CYGSEM_HAL_USE_ROM_MONITOR {
    display "Work with a ROM monitor"
Chapter 6. Porting Guide

Support can be enabled for three different varieties of ROM monitor. This support changes various eCos semantics such as the encoding of diagnostic output, or the overriding of hardware interrupt vectors.

Firstly there is "Generic" support which prevents the HAL from overriding the hardware vectors that it does not use, to instead allow an installed ROM monitor to handle them. This is the most basic support which is likely to be common to most implementations of ROM monitor.

"CygMon" provides support for the Cygnus ROM Monitor.

And finally, "GDB_stubs" provides support when GDB stubs are included in the ROM monitor or boot ROM.

Or the ability to be configured as a ROM monitor:

cdl_option CYGSEM_HAL_ROM_MONITOR {

display "Behave as a ROM monitor"

flavor bool

default_value 0

parent CYGPKG_HAL_ROM_MONITOR

requires { CYG_HAL_STARTUP == "ROM" }

description "Enable this option if this program is to be used as a ROM monitor, i.e. applications will be loaded into RAM on the board, and this ROM monitor may process exceptions or interrupts generated from the application. This enables features such as utilizing a separate interrupt stack when exceptions are generated."
}

The latter option is accompanied by a special build rule that extends the generic ROM monitor build rule in the common HAL:

cdl_option CYGBLD_BUILD_GDB_STUBS {

display "Build GDB stub ROM image"

default_value 0

requires { CYG_HAL_STARTUP == "ROM" }

requires CYGSEM_HAL_ROM_MONITOR

requires CYGBLD_BUILD_COMMON_GDB_STUBS

requires CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS

requires CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT

requires CYGDBG_HAL_DEBUG_GDB_THREAD_SUPPORT

requires CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT

requires CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM

no_define

description "This option enables the building of the GDB stubs for the board. The common HAL controls takes care of most of the build process, but the final conversion from ELF image to binary data is handled by the platform CDL, allowing relocation of the data if necessary."

make -priority 320 {

<PREFIX>/bin/gdb_module.bin : <PREFIX>/bin/gdb_module.img
$(OBJCOPY) -O binary $< $@
Chapter 6. Porting Guide

Most platforms support RedBoot, and some options are needed to configure for RedBoot.

```c
cdl_component CYGPKG_REDBOOT_HAL_OPTIONS {
    display "Redboot HAL options"
    flavor none
    no_define
    parent CYGPKG_REDBOOT
    active_if CYGPKG_REDBOOT
    description "This option lists the target’s requirements for a valid Redboot configuration."
}

cdl_option CYGBLD_BUILD_REDBOOT_BIN {
    display "Build Redboot ROM binary image"
    active_if CYGBLD_BUILD_REDBOOT
    default_value 1
    no_define
    description "This option enables the conversion of the Redboot ELF image to a binary image suitable for ROM programming."
}

make -priority 325 {
    <PREFIX>/bin/redboot.bin : <PREFIX>/bin/redboot.elf
    $(OBJCOPY) --strip-debug $< $(@:.bin=.img)
    $(OBJCOPY) -O srec $< $(@:.bin=.srec)
    $(OBJCOPY) -O binary $< $@
}
```

The important part here is the `make` command in the `CYGBLD_BUILD_REDBOOT_BIN` option which emits `makefile` commands to translate the `.elf` file generated by the link phase into both a binary file and an S-Record file. If a different format is required by a PROM programmer or ROM monitor, then different output formats would need to be generated here.

Platform Memory Layout

The platform memory layout is defined using the Memory Configuration Window in the Configuration Tool.

Note: If you do not have access to a Windows machine, you can hand edit the `.h` and `.ldi` files to match the properties of your platform. If you want to contribute your port back to the eCos community, ask someone on the list to make proper memory map files for you.

Layout Files

The memory configuration details are saved in three files:

```.mlt```

This is the Configuration Tool save-file. It is only used by the Configuration Tool.
This is the linker script fragment. It defines the memory and location of sections by way of macros defined in the architecture or variant linker script.

This file describes some of the memory region details as C macros, allowing eCos or the application adapt the memory layout of a specific configuration.

These three files are generated for each startup-type, since the memory details usually differ.

**Reserved Regions**

Some areas of the memory space are reserved for specific purposes, making room for exception vectors and various tables. RAM startup configurations also need to reserve some space at the bottom of the memory map for the ROM monitor.

These reserved areas are named with the prefix "reserved_" which is handled specially by the Configuration Tool: instead of referring to a linker macro, the start of the area is labeled and a gap left in the memory map.

**Platform Serial Device Support**

The first step is to set up the CDL definitions. The configuration options that need to be set are the following:

- `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS` - The number of channels, usually 0, 1 or 2.
- `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` - The channel to use for GDB.
- `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` - Initial baud rate for debug channel.
- `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` - The channel to use for the console.
- `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` - The initial baud rate for the console channel.
- `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_DEFAULT` - The default console channel.

The code in `hal_diag.c` need to be converted to support the new serial device. If this the same as a device already supported, copy that.

The following functions and types need to be rewritten to support a new serial device.
Chapter 6. Porting Guide

```
struct channel_data_t;

    Structure containing base address, timeout and ISR vector number for each serial device supported. Extra
    fields my be added if necessary for the device. For example some devices have write-only control registers,
    so keeping a shadow of the last value written here can be useful.

xxxxx_ser_channels[];

    Array of channel_data_t, initialized with parameters of each channel. The index into this array is the
    channel number used in the CDL options above and is used by the virtual vector mechanism to refer to
    each channel.

void cyg_hal_plf_serial_init_channel(void *ch_data)

    Initialize the serial device. The parameter is actually a pointer to a channel_data_t and should be cast
    back to this type before use. This function should use the CDL definition for the baud rate for the channel
    it is initializing.

void cyg_hal_plf_serial_putchar(void *ch_data, char *c)

    Send a character to the serial device. This function should poll for the device being ready to send and then
    write the character. Since this is intended to be a diagnostic/debug channel, it is often also a good idea to
    poll for end of transmission too. This ensures that as much data gets out of the system as possible.

bool cyg_hal_plf_serial_getchar_nonblock(void *ch_data, cyg_uint8* ch)

    This function tests the device and if a character is available, places it in *ch and returns TRUE. If no
    character is available, then the function returns FALSE immediately.

int cyg_hal_plf_serial_control(void *ch_data, __comm_control_cmd_t __func, ...)

    This is an IOCTL-like function for controlling various aspects of the serial device. The only part in which
    you may need to do some work initially is in the __COMMCTL_IRQ_ENABLE and __COMMCTL_IRQ_DISABLE
    cases to enable/disable interrupts.

int cyg_hal_plf_serial_isr(void *ch_data, int *ctrlc, CYG_ADDRWORD __vector,
CYG_ADDRWORD __data)

    This interrupt handler, called from the spurious interrupt vector, is specifically for dealing with Ctrl-C
    interrupts from GDB. When called this function should do the following:

    1. Check for an incoming character. The code here is very similar to that in
    cyg_hal_plf_serial_getchar_nonblock().

    2. Read the character and call cyg_hal_is_break().

    3. If result is true, set *ctrlc to 1.

    4. Return CYG_ISR_HANDLED.

void cyg_hal_plf_serial_init()

    Initialize each of the serial channels. First call cyg_hal_plf_serial_init_channel() for each channel. Then
    call the CYGACC_COMM_IF_* macros for each channel. This latter set of calls are identical for all
    channels, so the best way to do this is to copy and edit an existing example.
```
Variant HAL Porting

A variant port can be a fairly limited job, but can also require quite a lot of work. A variant HAL describes how a specific CPU variant differs from the generic CPU architecture. The variant HAL can re-define cache, MMU, interrupt, and other features which override the default implementation provided by the architecture HAL.

Doing a variant port requires a preexisting architecture HAL port. It is also likely that a platform port will have to be done at the same time if it is to be tested.

HAL Variant Porting Process

The easiest way to make a new variant HAL is simply to copy an existing variant HAL and change all the files to match the new variant. If this is the first variant for an architecture, it may be hard to decide which parts should be put in the variant - knowledge of other variants of the architecture is required.

Looking at existing variant HALs (e.g., MIPS tx39, tx49) may be a help - usually things such as caching, interrupt and exception handling differ between variants. Initialization code, and code for handling various core components (FPU, DSP, MMU, etc.) may also differ or be missing altogether on some variants. Linker scripts may also require specific variant versions.

Note: Some CPU variants may require specific compiler support. That support must be in place before you can undertake the eCos variant port.

HAL Variant CDL

The CDL in a variant HAL tends to depend on the exact functionality supported by the variant. If it implements some of the devices described in the platform HAL, then the CDL for those will be here rather than there (for example the real-time clock).

There may also be CDL to select options in the architecture HAL to configure it to a particular architectural variant.

Each variant needs an entry in the ecos.db file. This is the one for the SH3:

```plaintext
cdl_package CYGPKG_HAL_MIPS_VR4300 {  
display "VR4300 variant"  
parent CYGPKG_HAL_MIPS  
implements CYGINT_HAL_MIPS_VARIANT  
hardware
}
```

As you can see, it is very similar to the platform entry.

The variant CDL file will contain a package entry named for the architecture and variant, matching the package name in the ecos.db file. Here is the initial part of the MIPS VR4300 CDL file:

```plaintext
cdl_package CYGPKG_HAL_MIPS_VR4300 {  
display "VR4300 variant"  
parent CYGPKG_HAL_MIPS  
implements CYGINT_HAL_MIPS_VARIANT  
hardware
}
```
Chapter 6. Porting Guide

include_dir cyg/hal
define_header hal_mips_vr4300.h
description "The VR4300 variant HAL package provides generic support for this processor architecture. It is also necessary to select a specific target platform HAL package."

This defines the package, placing it under the MIPS architecture package in the hierarchy. The implements line indicates that this is a MIPS variant. The architecture package uses this to check that exactly one variant is configured in.

The variant defines some options that cause the architecture HAL to configure itself to support this variant.

cdl_option CYGHWR_HAL_MIPS_64BIT {
    display "Variant 64 bit architecture support"
    calculated 1
}

cdl_option CYGHWR_HAL_MIPS_FPU {
    display "Variant FPU support"
    calculated 1
}

cdl_option CYGHWR_HAL_MIPS_FPU_64BIT {
    display "Variant 64 bit FPU support"
    calculated 1
}

These tell the architecture that this is a 64 bit MIPS architecture, that it has a floating point unit, and that we are going to use it in 64 bit mode rather than 32 bit mode.

The CDL file finishes off with some build options.

define_proc {
    puts $::cdl_header "#include <pkgconf/hal_mips.h>"
}

compile var_misc.c

make {
    <PREFIX>/lib/target.ld: <PACKAGE>/src/mips_vr4300.ld
    $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS) -o $@ $<
    @echo $@ " : \"" > $(notdir $@).deps
    @tail +2 target.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm target.tmp
}

cdl_option CYGBLD_LINKER_SCRIPT {
    display "Linker script"
    flavor data
    no_define
    calculated { "src/mips_vr4300.ld" }
}

The define_proc causes the architecture configuration file to be included into the configuration file for the variant. The compile causes the single source file for this variant, var_misc.c to be compiled. The make
command emits makefile rules to combine the linker script with the .ldi file to generate target.ld. Finally, in the MIPS HALs, the main linker script is defined in the variant, rather than the architecture, so CYGBLD_LINKER_SCRIPT is defined here.

## Cache Support

The main area where the variant is likely to be involved is in cache support. Often the only thing that distinguishes one CPU variant from another is the size of its caches.

In architectures such as the MIPS and PowerPC where cache instructions are part of the ISA, most of the actual cache operations are implemented in the architecture HAL. In this case the variant HAL only needs to define the cache dimensions. The following are the cache dimensions defined in the MIPS VR4300 variant var_cache.h.

```c
// Data cache
#define HAL_DCACHE_SIZE (8*1024) // Size of data cache in bytes
#define HAL_DCACHE_LINE_SIZE 16 // Size of a data cache line
#define HAL_DCACHE_WAYS 1 // Associativity of the cache

// Instruction cache
#define HAL_ICACHE_SIZE (16*1024) // Size of cache in bytes
#define HAL_ICACHE_LINE_SIZE 32 // Size of a cache line
#define HAL_ICACHE_WAYS 1 // Associativity of the cache
#define HAL_DCACHE_SETS (HAL_DCACHE_SIZE/(HAL_DCACHE_LINE_SIZE*HAL_DCACHE_WAYS))
#define HAL_ICACHE_SETS (HAL_ICACHE_SIZE/(HAL_ICACHE_LINE_SIZE*HAL_ICACHE_WAYS))
```

Additional cache macros, or overrides for the defaults, may also appear in here. While some architectures have instructions for managing cache lines, overall enable/disable operations may be handled via variant specific registers. If so then var_cache.h should also define the HAL_XCACHE_ENABLE() and HAL_XCACHE_DISABLE() macros.

If there are any generic features that the variant does not support (cache locking is a typical example) then var_cache.h may need to disable definitions of certain operations. It is architecture dependent exactly how this is done.

## Architecture HAL Porting

A new architecture HAL is the most complex HAL to write, and it the least easily described. Hence this section is presently nothing more than a place holder for the future.

## HAL Architecture Porting Process

The easiest way to make a new architecture HAL is simply to copy an existing architecture HAL of an, if possible, closely matching architecture and change all the files to match the new architecture. The MIPS architecture HAL should be used if possible, as it has the appropriate layout and coding conventions. Other HALs may deviate from that norm in various ways.

**Note:** eCos is written for GCC. It requires C and C++ compiler support as well as a few compiler features introduced during eCos development - so compilers older than eCos may not provide these features. Note
that there is no C++ support for any 8 or 16 bit CPUs. Before you can undertake an eCos port, you need the required compiler support.

The following gives a rough outline of the steps needed to create a new architecture HAL. The exact order and set of steps needed will vary greatly from architecture to architecture, so a lot of flexibility is required. And of course, if the architecture HAL is to be tested, it is necessary to do variant and platform ports for the initial target simultaneously.

1. Make a new directory for the new architecture under the `hal` directory in the source repository. Make an `arch` directory under this and populate this with the standard set of package directories.

2. Copy the CDL file from an example HAL changing its name to match the new HAL. Edit the file, changing option names as appropriate. Delete any options that are specific to the original HAL, and any new options that are necessary for the new architecture. This is likely to be a continuing process during the development of the HAL. See the section called **CDL Requirements** for more details.

3. Copy the `hal_arch.h` file from an example HAL. Within this file you need to change or define the following:
   - Define the HAL_SavedRegisters structure. This may need to reflect the save order of any group register save/restore instructions, the interrupt and exception save and restore formats, and the procedure calling conventions. It may also need to cater for optional FPUs and other functional units. It can be quite difficult to develop a layout that copes with all requirements.
   - Define the bit manipulation routines, `HAL_LSBIT_INDEX()` and `HAL_MSBIT_INDEX()`. If the architecture contains instructions to perform these, or related, operations, then these should be defined as inline assembler fragments. Otherwise make them calls to functions.
   - Define `HAL_THREAD_INIT_CONTEXT()`. This initializes a restorable CPU context onto a stack pointer so that a later call to `HAL_THREAD_LOAD_CONTEXT()` or `HAL_THREAD_SWITCH_CONTEXT()` will execute it correctly. This macro needs to take account of the same optional features of the architecture as the definition of HAL_SavedRegisters.
   - Define `HAL_THREAD_LOAD_CONTEXT()` and `HAL_THREAD_SWITCH_CONTEXT()`. These should just be calls to functions in `context.S`.
   - Define `HAL_REORDER_BARRIER()`. This prevents code being moved by the compiler and is necessary in some order-sensitive code. This macro is actually defined identically in all architecture, so it can just be copied.
   - Define breakpoint support. The macro `HAL_BREAKPOINT(label)` needs to be an inline assembly fragment that invokes a breakpoint. The breakpoint instruction should be labeled with the `label` argument. `HAL_BREAKINST` and `HAL_BREAKINST_SIZE` define the breakpoint instruction for debugging purposes.
   - Define GDB support. GDB views the registers of the target as a linear array, with each register having a well defined offset. This array may differ from the ordering defined in HAL_SavedRegisters. The macros `HAL_GET_GDB_REGISTERS()` and `HAL_SET_GDB_REGISTERS()` translate between the GDB array and the HAL_SavedRegisters structure. The `HAL_THREAD_GET_SAVED_REGISTERS()` translates a stack pointer saved by the context switch macros into a pointer to a HAL_SavedRegisters structure. Usually this is a one-to-one translation, but this macro allows it to differ if necessary.
   - Define long jump support. The type `hal_jmp_buf` and the functions `hal_setjmp()` and `hal_longjmp()` provide the underlying implementation of the C library `setjmp()` and `longjmp()`.
   - Define idle thread action. Generally the macro `HAL_IDLE_THREAD_ACTION()` is defined to call a function in `hal_misc.c`.
• Define stack sizes. The macros `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HAL_STACK_SIZE_TYPICAL` should be defined to the minimum size for any thread stack and a reasonable default for most threads respectively. It is usually best to construct these out of component sizes for the CPU save state and procedure call stack usage. These definitions should not use anything other than numerical values since they can be used from assembly code in some HALs.

• Define memory access macros. These macros provide translation between cached and uncached and physical memory spaces. They usually consist of masking out bits of the supplied address and ORing in alternative address bits.

• Define global pointer save/restore macros. These really only need defining if the calling conventions of the architecture require a global pointer (as does the MIPS architecture), they may be empty otherwise. If it is necessary to define these, then take a look at the MIPS implementation for an example.

4. Copy `hal_intr.h` from an example HAL. Within this file you should change or define the following:

• Define the exception vectors. These should be detailed in the architecture specification. Essentially for each exception entry point defined by the architecture there should be an entry in the VSR table. The offsets of these VSR table entries should be defined here by `CYGNUM_HAL_VECTOR_*` definitions. The size of the VSR table also needs to be defined here.

• Map any hardware exceptions to standard names. There is a group of exception vector name of the form `CYGNUM_HAL_EXCEPTION_*` that define a wide variety of possible exceptions that many architectures raise. Generic code detects whether the architecture can raise a given exception by testing whether a given `CYGNUM_HAL_EXCEPTION_*` definition is present. If it is present then its value is the vector that raises that exception. This does not need to be a one-to-one correspondence, and several `CYGNUM_HAL_EXCEPTION_*` definitions may have the same value.

Interrupt vectors are usually defined in the variant or platform HALs. The interrupt number space may either be continuous with the VSR number space, where they share a vector table (as in the i386) or may be a separate space where a separate decode stage is used (as in MIPS or PowerPC).

• Declare any static data used by the HAL to handle interrupts and exceptions. This is usually three vectors for interrupts: `hal_interrupt_handlers[]`, `hal_interrupt_data[]` and `hal_interrupt_objects[]`, which are sized according to the interrupt vector definitions. In addition a definition for the VSR table, `hal_vsr_table[]` should be made. These vectors are normally defined in either `vectors.S` or `hal_misc.c`.

• Define interrupt enable/disable macros. These are normally inline assembly fragments to execute the instructions, or manipulate the CPU register, that contains the CPU interrupt enable bit.

• A feature that many HALs support is the ability to execute DSRs on the interrupt stack. This is not an essential feature, and is better left unimplemented in the initial porting effort. If this is required, then the macro `HAL_INTERRUPT_STACK_CALL_PENDING_DSRs()` should be defined to call a function in `vectors.S`.

• Define the interrupt and VSR attachment macros. If the same arrays as for other HALs have been used for VSR and interrupt vectors, then these macro can be copied across unchanged.

5. A number of other header files also need to be filled in:

• `basetype.h`. This file defines the basic types used by eCos, together with the endianness and some other characteristics. This file only really needs to contain definitions if the architecture differs significantly from the defaults defined in `cyg_type.h`

• `hal_io.h`. This file contains macros for accessing device IO registers. If the architecture uses memory mapped IO, then these can be copied unchanged from an existing HAL such as MIPS. If the architecture
Chapter 6. Porting Guide

uses special IO instructions, then these macros must be defined as inline assembler fragments. See the
I386 HAL for an example. PCI bus access macros are usually defined in the variant or platform HALs.

• hal_cache.h. This file contains cache access macros. If the architecture defines cache instructions, or
control registers, then the access macros should be defined here. Otherwise they must be defined in the
variant or platform HAL. Usually the cache dimensions (total size, line size, ways etc.) are defined in
the variant HAL.

• arch.inc and <architecture>.inc. These files are assembler headers used by vectors.S and
context.S.<architecture>.inc is a general purpose header that should contain things like register
aliases, ABI definitions and macros useful to general assembly code. If there are no such definitions,
then this file need not be provided. arch.inc contains macros for performing various eCos related
operations such as initializing the CPU, caches, FPU etc. The definitions here may often be configured
or overridden by definitions in the variant or platform HALs. See the MIPS HAL for an example of
this.

6. Write vectors.S. This is the most important file in the HAL. It contains the CPU initialization code,
exception and interrupt handlers. While other HALs should be consulted for structures and techniques,
there is very little here that can be copied over without major edits.

The main pieces of code that need to be defined here are:

• Reset vector. This usually need to be positioned at the start of the ROM or FLASH, so should be in a
linker section of its own. It can then be placed correctly by the linker script. Normally this code is little
more than a jump to the label _start.

• Exception vectors. These are the trampoline routines connected to the hardware exception entry points
that vector through the VSR table. In many architectures these are adjacent to the reset vector, and
should occupy the same linker section. If the architecture allow the vectors to be moved then it may be
necessary for these trampolines to be position independent so they can be relocated at runtime.

The trampolines should do the minimum necessary to transfer control from the hardware vector to the
VSR pointed to by the matching table entry. Exactly how this is done depends on the architecture. Usu-
ally the trampoline needs to get some working registers by either saving them to CPU special registers
(e.g. PowerPC SPRs), using reserved general registers (MIPS K0 and K1), using only memory based
operations (IA32), or just jumping directly (ARM). The VSR table index to be used is either implicit in
the entry point taken (PowerPC, IA32, ARM), or must be determined from a CPU register (MIPS).

• Write kernel startup code. This is the location the reset vector jumps to, and can be in the main text
section of the executable, rather than a special section. The code here should first initialize the CPU
and other hardware subsystems. The best approach is to use a set of macro calls that are defined either
in arch.inc or overridden in the variant or platform HALs. Other jobs that this code should do are:
initialize stack pointer; copy the data section from ROM to RAM if necessary; zero the BSS; call
variant and platform initializers; call cyg_hal_invoke_constructors(); call initialize_stub() if
necessary. Finally it should call cyg_start(). See the Section called HAL Startup in Chapter 5 for
details.

• Write the default exception VSR. This VSR is installed in the VSR table for all synchronous exception
vectors. See the Section called Default Synchronous Exception Handling in Chapter 5 for details of
what this VSR does.

• Write the default interrupt VSR. This is installed in all VSR table entries that correspond to external
interrupts. See the Section called Default Synchronous Exception Handling in Chapter 5 for details of
what this VSR does.

• Write hal_interrupt_stack_call_pending_drs(). If this function is defined in hal_arch.h then
it should appear here. The purpose of this function is to call DSRs on the interrupt stack rather than the
current thread’s stack. This is not an essential feature, and may be left until later. However it interacts
with the stack switching that goes on in the interrupt VSR, so it may make sense to write these pieces
of code at the same time to ensure consistency.

When this function is implemented it should do the following:

- Take a copy of the current SP and then switch to the interrupt stack.
- Save the old SP, together with the CPU status register (or whatever register contains the interrupt
  enable status) and any other registers that may be corrupted by a function call (such as any link
  register) to locations in the interrupt stack.
- Enable interrupts.
- Call cyg_interrupt_call_pending_DSRs(). This is a kernel function that actually calls any pend-
  ing DSRs.
- Retrieve saved registers from the interrupt stack and switch back to the current thread stack.
- Merge the interrupt enable state recorded in the save CPU status register with the current value of
  the status register to restore the previous enable state. If the status register does not contain any other
  persistent state then this can be a simple restore of the register. However if the register contains other
  state bits that might have been changed by a DSR, then care must be taken not to disturb these.

- Define any data items needed. Typically vectors.S may contain definitions for the VSR table, the inter-
  rupt tables and the interrupt stack. Sometimes these are only default definitions that may be overridden
  by the variant or platform HALs.

7. Write context.S. This file contains the context switch code. See the Section called Thread Context
   Switching in Chapter 4 for details of how these functions operate. This file may also contain the im-
   plementation of hal_setjmp() and hal_longjmp().

8. Write hal_misc.c. This file contains any C data and functions needed by the HAL. These might include:

   - hal_interrupt_*[]. In some HALs, if these arrays are not defined in vectors.S then they must be
     defined here.
   - cyg_hal_exception_handler(). This function is called from the exception VSR. It usually does
     extra decoding of the exception and invokes any special handlers for things like FPU traps, bus
     errors or memory exceptions. If there is nothing special to be done for an exception, then it either
     calls into the GDB stubs, by calling __handle_exception(), or invokes the kernel by calling
     cyg_hal_deliver_exception().
   - hal_arch_default_isr(). The hal_interrupt_handlers[] array is usually initialized with pointers
     to hal_default_isr(), which is defined in the common HAL. This function handles things like Ctrl-
     C processing, but if that is not relevant, then it will call hal_arch_default_isr(). Normally this
     function should just return zero.
   - cyg_hal_invoke_constructors(). This calls the constructors for all static objects before the program
     starts. eCos relies on these being called in the correct order for it to function correctly. The exact way in
     which constructors are handled may differ between architectures, although most use a simple table of
     function pointers between labels __CTOR_LIST__ and __CTOR_END__ which must called in order from
     the top down. Generally, this function can be copied directly from an existing architecture HAL.
   - Bit indexing functions. If the macros HAL_LSBIT_INDEX() and HAL_MSBIT_INDEX() are defined as
     function calls, then the functions should appear here. The main reason for doing this is that the ar-
     chitecture does not have support for bit indexing and these functions must provide the functionality
by conventional means. While the trivial implementation is a simple for loop, it is expensive and non-deterministic. Better, constant time, implementations can be found in several HALs (MIPS for example).

- **hal_delay_us().** If the macro `HAL_DELAY_US()` is defined in `hal_intr.h` then it should be defined to call this function. While most of the time this function is called with very small values, occasionally (particularly in some ethernet drivers) it is called with values of several seconds. Hence the function should take care to avoid overflow in any calculations.

- **hal_idle_thread_action().** This function is called from the idle thread via the `HAL_IDLE_THREAD_ACTION()` macro, if so defined. While normally this function does nothing, during development this is often a good place to report various important system parameters on LCDs, LED or other displays. This function can also monitor system state and report any anomalies. If the architecture supports a `halt` instruction then this is a good place to put an inline assembly fragment to execute it. It is also a good place to handle any power saving activity.

9. Create the `<architecture>.ld` file. While this file may need to be moved to the variant HAL in the future, it should initially be defined here, and only moved if necessary.

This file defines a set of macros that are used by the platform `.ld` files to generate linker scripts. Most GCC toolchains are very similar so the correct approach is to copy the file from an existing architecture and edit it. The main things that will need editing are the `OUTPUT_FORMAT()` directive and maybe the creation or allocation of extra sections to various macros. Running the target linker with just the `--verbose` argument will cause it to output its default linker script. This can be compared with the `.ld` file and appropriate edits made.

10. If GDB stubs are to be supported in RedBoot or eCos, then support must be included for these. The most important of these are `include/<architecture>-stub.h` and `src/<architecture>-stub.c`. In all existing architecture HALs these files, and any support files they need, have been derived from files supplied in `libgloss`, as part of the GDB toolchain package. If this is a totally new architecture, this may not have been done, and they must be created from scratch.

   `include/<architecture>-stub.h` contains definitions that are used by the GDB stubs to describe the size, type, number and names of CPU registers. This information is usually found in the GDB support files for the architecture. It also contains prototypes for the functions exported by `src/<architecture>-stub.c`; however, since this is common to all architectures, it can be copied from some other HAL.

   `src/<architecture>-stub.c` implements the functions exported by the header. Most of this is fairly straightforward: the implementation in existing HALs should show exactly what needs to be done. The only complex part is the support for single-stepping. This is used a lot by GDB, so it cannot be avoided. If the architecture has support for a trace or single-step trap then that can be used for this purpose. If it does not then this must be simulated by planting a breakpoint in the next instruction. This can be quite involved since it requires some analysis of the current instruction plus the state of the CPU to determine where execution is going to go next.

**CDL Requirements**

The CDL needed for any particular architecture HAL depends to a large extent on the needs of that architecture. This includes issues such as support for different variants, use of FPUs, MMUs and caches. The exact split between the architecture, variant and platform HALs for various features is also somewhat fluid.

To give a rough idea about how the CDL for an architecture is structured, we will take as an example the I386 CDL.
Chapter 6. Porting Guide

This first section introduces the CDL package and placed it under the main HAL package. Include files from this package will be put in the `include/cyg/hal` directory, and definitions from this file will be placed in `include/pkgconf/hal_i386.h`. The `compile` line specifies the files in the `src` directory that are to be compiled as part of this package.

```cdl
cdl_package CYGPKG_HAL_I386 {
  display "i386 architecture"
  parent CYGPKG_HAL
  hardware
  include_dir cyg/hal
  define_header hal_i386.h
  description "The i386 architecture HAL package provides generic support for this processor architecture. It is also necessary to select a specific target platform HAL package."
  compile hal_misc.c context.S i386_stub.c hal_syscall.c
}
```

Next we need to generate some files using non-standard make rules. The first is `vectors.S`, which is not put into the library, but linked explicitly with all applications. The second is the generation of the `target.ld` file from `i386.ld` and the startup-selected `.ldi` file. Both of these are essentially boilerplate code that can be copied and edited.

```make
make {
  @echo $@ ": \"
  $(notdir $@).deps
  @tail +2 vectors.tmp >> $(notdir $@).deps
  @echo $@ ": \"
  $(notdir $@).deps
  @rm vectors.tmp
}
make {
  @echo $@ ": \"
  $(notdir $@).deps
  @rm target.tmp
}
```

The i386 is currently the only architecture that supports SMP. The following CDL simply enabled the HAL SMP support if required. Generally this will get enabled as a result of a `requires` statement in the kernel. The `requires` statement here turns off lazy FPU switching in the FPU support code, since it is inconsistent with SMP operation.

```cdl
cdl_component CYGPKG_HAL_SMP_SUPPORT {
  display "SMP support"
  default_value 0
  requires { CYGHWR_HAL_I386_FPU_SWITCH_LAZY == 0 }
}

cdl_option CYGPKG_HAL_SMP_CPU_MAX {
  display "Max number of CPUs supported"
  flavor data
  default_value 2
}
```

71
Chapter 6. Porting Guide

The i386 HAL has optional FPU support, which is enabled by default. It can be disabled to improve system performance. There are two FPU support options: either to save and restore the FPU state on every context switch, or to only switch the FPU state when necessary.

```cdl

cdl_component CYGHWR_HAL_I386_FPU {
    display "Enable I386 FPU support"
    default_value 1
    description "This component enables support for the
    I386 floating point unit."

cdl_option CYGHWR_HAL_I386_FPU_SWITCH_LAZY {
    display "Use lazy FPU state switching"
    flavor bool
    default_value 1
    description "This option enables lazy FPU state switching.
    The default behaviour for eCos is to save and
    restore FPU state on every thread switch, interrupt
    and exception. While simple and deterministic, this
    approach can be expensive if the FPU is not used by
    all threads. The alternative, enabled by this option,
    is to use hardware features that allow the FPU state
    of a thread to be left in the FPU after it has been
descheduled, and to allow the state to be switched to
    a new thread only if it actually uses the FPU. Where
    only one or two threads use the FPU this can avoid a
    lot of unnecessary state switching."
}
```

The i386 HAL also has support for different classes of CPU. In particular, Pentium class CPUs have extra functional units, and some variants of GDB expect more registers to be reported. These options enable these features. Generally these are enabled by `requires` statements in variant or platform packages, or in `.ecm` files.

```cdl

cdl_component CYGHWR_HAL_I386_PENTIUM {
    display "Enable Pentium class CPU features"
    default_value 0
    description "This component enables support for various
    features of Pentium class CPUs."

cdl_option CYGHWR_HAL_I386_PENTIUM_SSE {
    display "Save/Restore SSE registers on context switch"
    flavor bool
    default_value 0
    description "This option enables SSE state switching. The default
    behaviour for eCos is to ignore the SSE registers. Enabling this option adds SSE state information to
    every thread context."
}

cdl_option CYGHWR_HAL_I386_PENTIUM_GDB_REGS {
    display "Support extra Pentium registers in GDB stub"
    flavor bool
    default_value 0
```
description "This option enables support for extra Pentium registers in the GDB stub. These are registers such as CR0-CR4, and all MSRs. Not all GDBs support these registers, so the default behaviour for eCos is to not include them in the GDB stub support code."
}

In the i386 HALs, the linker script is provided by the architecture HAL. In other HALs, for example MIPS, it is provided in the variant HAL. The following option provides the name of the linker script to other elements in the configuration system.

```c

cdl_option CYGBLD_LINKER_SCRIPT {
    display "Linker script"
    flavor data
    no_define
    calculated { "src/i386.ld" }
}
```

Finally, this interface indicates whether the platform supplied an implementation of the `hal_i386_mem_real_region_top()` function. If it does then it will contain a line of the form: `implements CYGINT_HAL_I386_MEM_REAL_REGION_TOP`. This allows packages such as RedBoot to detect the presence of this function so that they may call it.

```c

cdl_interface CYGINT_HAL_I386_MEM_REAL_REGION_TOP {
    display "Implementations of hal_i386_mem_real_region_top()"
}
```
Chapter 7. Future developments

The HAL is not complete, and will evolve and increase over time. Among the intended developments are:

• Common macros for interpreting the contents of a saved machine context. These would allow portable code, such as debug stubs, to extract such values as the program counter and stack pointer from a state without having to interpret a HAL_SavedRegisters structure directly.

• Debugging support. Macros to set and clear hardware and software breakpoints. Access to other areas of machine state may also be supported.

• Static initialization support. The current HAL provides a dynamic interface to things like thread context initialization and ISR attachment. We also need to be able to define the system entirely statically so that it is ready to go on restart, without needing to run code. This will require extra macros to define these initializations. Such support may have a consequential effect on the current HAL specification.

• CPU state control. Many CPUs have both kernel and user states. Although it is not intended to run any code in user state for the foreseeable future, it is possible that this may happen eventually. If this is the case, then some minor changes may be needed to the current HAL API to accommodate this. These should mostly be extensions, but minor changes in semantics may also be required.

• Physical memory management. Many embedded systems have multiple memory areas with varying properties such as base address, size, speed, bus width, cacheability and persistence. An API is needed to support the discovery of this information about the machine’s physical memory map.

• Memory management control. Some embedded processors have a memory management unit. In some cases this must be enabled to allow the cache to be controlled, particularly if different regions of memory must have different caching properties. For some purposes, in some systems, it will be useful to manipulate the MMU settings dynamically.

• Power management. Macros to access and control any power management mechanisms available on the CPU implementation. These would provide a substrate for a more general power management system that also involved device drivers and other hardware components.

• Generic serial line macros. Most serial line devices operate in the same way, the only real differences being exactly which bits in which registers perform the standard functions. It should be possible to develop a set of HAL macros that provide basic serial line services such as baud rate setting, enabling interrupts, polling for transmit or receive ready, transmitting and receiving data etc. Given these it should be possible to create a generic serial line device driver that will allow rapid bootstrapping on any new platform. It may be possible to extend this mechanism to other device types.
Chapter 7. Future developments
III. The ISO Standard C and Math Libraries
Chapter 8. C and math library overview

eCos provides compatibility with the ISO 9899:1990 specification for the standard C library, which is essentially the same as the better-known ANSI C3.159-1989 specification (C-89).

There are three aspects of this compatibility supplied by eCos. First there is a C library which implements the functions defined by the ISO standard, except for the mathematical functions. This is provided by the eCos C library packages.

Then eCos provides a math library, which implements the mathematical functions from the ISO C library. This distinction between C and math libraries is frequently drawn — most standard C library implementations provide separate linkable files for the two, and the math library contains all the functions from the math.h header file.

There is a third element to the ISO C library, which is the environment in which applications run when they use the standard C library. This environment is set up by the C library startup procedure (the Section called C library startup) and it provides (among other things) a main() entry point function, an exit() function that does the cleanup required by the standard (including handlers registered using the atexit() function), and an environment that can be read with getenv().

The description in this manual focuses on the eCos-specific aspects of the C library (mostly related to eCos’s configurability) as well as mentioning the omissions from the standard in this release. We do not attempt to define the semantics of each function, since that information can be found in the ISO, ANSI, POSIX and IEEE standards, and the many good books that have been written about the standard C library, that cover usage of these functions in a more general and useful way.

Included non-ISO functions

The following functions from the POSIX specification are included for convenience:

extern char **environ variable (for setting up the environment for use with getenv())
_exit()
strtok_r()
rand_r()
asctime_r()
ctime_r()
localtime_r()
gmtime_r()
eCos provides the following additional implementation-specific functions within the standard C library to adjust the date and time settings:

void cyg_libc_time_setdst{
  cyg_libc_time_dst state
};

This function sets the state of Daylight Savings Time. The values for state are:

CYG_LIBC_TIME_DSTNA unknown
CYG_LIBC_TIME_DSTOFF off
CYG_LIBC_TIME_DSTON on

void cyg_libc_time_setzoneoffsets{
Chapter 8. C and math library overview

This function sets the offsets from UTC used when Daylight Savings Time is enabled or disabled. The offsets are in time_t’s, which are seconds in the current implementation.

```c
Cyg_libc_time_dst cyg_libc_time_getzoneoffsets(
    time_t *stdoffset, time_t *dstoffset
);
```

This function retrieves the current setting for Daylight Savings Time along with the offsets used for both STD and DST. The offsets are both in time_t’s, which are seconds in the current implementation.

```c
cyg_bool cyg_libc_time_settime(
    time_t utctime
);
```

This function sets the current time for the system. The time is specified as a time_t in UTC. It returns non-zero on error.

Math library compatibility modes

This math library is capable of being operated in several different compatibility modes. These options deal solely with how errors are handled.

There are 4 compatibility modes: ANSI/POSIX 1003.1; IEEE-754; X/Open Portability Guide issue 3 (XPG3); and System V Interface Definition Edition 3.

In IEEE mode, the `matherr()` function (see below) is never called, no warning messages are printed on the stderr output stream, and errno is never set.

In ANSI/POSIX mode, errno is set correctly, but `matherr()` is never called and no warning messages are printed on the stderr output stream.

In X/Open mode, errno is set correctly, `matherr()` is called, but no warning messages are printed on the stderr output stream.

In SVID mode, functions which overflow return a value HUGE (defined in `math.h`), which is the maximum single precision floating point value (as opposed to HUGE_VAL which is meant to stand for infinity), errno is set correctly and `matherr()` is called. If `matherr()` returns 0, warning messages are printed on the stderr output stream for some errors.

The mode can be compiled-in as IEEE-only, or any one of the above methods settable at run-time.

**Note:** This math library assumes that the hardware (or software floating point emulation) supports IEEE-754 style arithmetic, 32-bit 2’s complement integer arithmetic, doubles are in 64-bit IEEE-754 format.

`matherr()`

As mentioned above, in X/Open or SVID modes, the user can supply a function `matherr()` of the form:

```c
int matherr( struct exception *e )
```

where struct exception is defined as:
struct exception {
    int type;
    char *name;
    double arg1, arg2, retval;
};

type is the exception type and is one of:

**DOMAIN**
- argument domain exception

**SING**
- argument singularity

**OVERFLOW**
- overflow range exception

**UNDERFLOW**
- underflow range exception

**TLOSS**
- total loss of significance

**PLOSS**
- partial loss of significance

*name* is a string containing the name of the function

*arg1* and *arg2* are the arguments passed to the function

*retval* is the default value that will be returned by the function, and can be changed by `matherr()`

**Note:** `matherr` must have “C” linkage, not “C++” linkage.

If `matherr` returns zero, or the user doesn’t supply their own `matherr`, then the following usually happens in SVID mode:

**Table 8-1. Behavior of math exception handling**

<table>
<thead>
<tr>
<th>Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMAIN</td>
<td>0.0 returned, errno=EDOM, and a message printed on stderr</td>
</tr>
<tr>
<td>SING</td>
<td>HUGE of appropriate sign is returned, errno=EDOM, and a message is printed on stderr</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>HUGE of appropriate sign is returned, and errno=ERANGE</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>0.0 is returned and errno=ERANGE</td>
</tr>
<tr>
<td>TLOSS</td>
<td>0.0 is returned, errno=ERANGE, and a message is printed on stderr</td>
</tr>
<tr>
<td>PLOSS</td>
<td>The current implementation doesn’t return this type</td>
</tr>
</tbody>
</table>
Chapter 8. C and math library overview

X/Open mode is similar except that the message is not printed on stderr and HUGE_VAL is used in place of HUGE.

Thread-safety and re-entrancy

With the appropriate configuration options set below, the math library is fully thread-safe if:

• Depending on the compatibility mode, the setting of the errno variable from the C library is thread-safe
• Depending on the compatibility mode, sending error messages to the stderr output stream using the C library puts() function is thread-safe
• Depending on the compatibility mode, the user-supplied matherr() function and anything it depends on are thread-safe

In addition, with the exception of the gamma*() and lgamma*() functions, the math library is reentrant (and thus safe to use from interrupt handlers) if the Math library is always in IEEE mode.

Some implementation details

Here are some details about the implementation which might be interesting, although they do not affect the ISO-defined semantics of the library.

• It is possible to configure eCos to have the standard C library without the kernel. You might want to do this to use less memory. But if you disable the kernel, you will be unable to use memory allocation, thread-safety and certain stdio functions such as input. Other C library functionality is unaffected.
• The opaque type returned by clock() is called clock_t, and is implemented as a 64 bit integer. The value returned by clock() is only correct if the kernel is configured with real-time clock support, as determined by the CYGVAR_KERNEL_COUNTERS_CLOCK configuration option in kernel.h.
• The FILE type is not implemented as a structure, but rather as a CYG_ADDRESS.
• The GNU C compiler will place its own built-in implementations instead of some C library functions. This can be turned off with the -fno-builtins option. The functions affected by this are abs(), cos(), fabs(), labs(), memcmp(), memcpy(), sin(), sqrt(), strcmp(), strcpy(), and strlen().
• For faster execution speed you should avoid this option and let the compiler use its built-ins. This can be turned off by invoking GCC with the -fno-builtins option.
• memcpy() and memset() are located in the infrastructure package, not in the C library package. This is because the compiler calls these functions, and the kernel needs to resolve them even if the C library is not configured.
• Error codes such as EDOM and ERANGE, as well as strerror(), are implemented in the error package. The error package is separate from the rest of the C and math libraries so that the rest of eCos can use these error handling facilities even if the C library is not configured.
• When free() is invoked, heap memory will normally be coalesced. If the CYGSEM_KERNEL_MEMORY_COALESCE configuration parameter is not set, memory will not be coalesced, which might cause programs to fail.
• Signals, as implemented by <signal.h>, are guaranteed to work correctly if raised using the raise() function from a normal working program context. Using signals from within an ISR or DSR context is not expected to work. Also, it is not guaranteed that if CYGSEM_LIBC_SIGNALS_HWEXCEPTIONS is set,
that handling a signal using \texttt{signal()} will necessarily catch that form of exception. For example, it may be expected that a divide-by-zero error would be caught by handling \texttt{SIGFPE}. However it depends on the underlying HAL implementation to implement the required hardware exception. And indeed the hardware itself may not be capable of detecting these exceptions so it may not be possible for the HAL implementer to do this in any case. Despite this lack of guarantees in this respect, the signals implementation is still ISO C compliant since ISO C does not offer any such guarantees either.

- The \texttt{getenv()} function is implemented (unless the \texttt{CYGPKG_LIBC_ENVIRONMENT} configuration option is turned off), but there is no shell or \texttt{putenv()} function to set the environment dynamically. The environment is set in a global variable \texttt{environ}, declared as:

  ```c
  extern char **environ; // Standard environment definition
  ```

  The environment can be statically initialized at startup time using the \texttt{CYG-DAT_LIBC_DEFAULT_ENVIRONMENT} option. If so, remember that the final entry of the array initializer must be \texttt{NULL}.

  Here is a minimal \texttt{eCos} program which demonstrates the use of environments (see also the test case in \texttt{language/c/libc/current/tests/stdlib/getenv.c}):

  ```c
  #include <stdio.h>
  #include <stdlib.h> // Main header for stdlib functions
  extern char **environ; // Standard environment definition
  
  int
  main( int argc, char *argv[] )
  {
    char *str;
    char *env[] = { "PATH=/usr/local/bin:/usr/bin",
                    "HOME=/home/fred",
                    "TEST=1234=5678",
                    "home=hatstand",
                    NULL };  
    printf("Display the current PATH environment variable\n");
    environ = (char **)&env;
    str = getenv("PATH");
    if (str==NULL) {
      printf("The current PATH is unset\n");
    } else {
      printf("The current PATH is \"%s\"\n", str);
    }
    return 0;
  }
  ```

### Thread safety

The ISO C library has configuration options that control thread safety, i.e. working behavior if multiple threads call the same function at the same time.

The following functionality has to be configured correctly, or used carefully in a multi-threaded environment:

- \texttt{mblen}()}
Chapter 8. C and math library overview

- mbtowc()
- wctomb()
- printf() (and all standard I/O functions except for sprintf() and sscanf())
- strtok()
- rand() and srand()
- signal() and raise()
- asctime(), ctime(), gmtime(), and localtime()
- the errno variable
- the environ variable
- date and time settings

In some cases, to make eCos development easier, functions are provided (as specified by POSIX 1003.1) that define re-entrant alternatives, i.e. rand_r(), strtok_r(), asctime_r(), ctime_r(), gmtime_r(), and localtime_r(). In other cases, configuration options are provided that control either locking of functions or their shared data, such as with standard I/O streams, or by using per-thread data, such as with the errno variable.

In some other cases, like the setting of date and time, no re-entrant or thread-safe alternative or configuration is provided as it is simply not a worthwhile addition (date and time should rarely need to be set.)

C library startup

The C library includes a function declared as:

```c
void cyg_iso_c_start( void )
```

This function is used to start an environment in which an ISO C style program can run in the most compatible way.

What this function does is to create a thread which will invoke main() — normally considered a program’s entry point. In particular, it can supply arguments to main() using the CYGDAT_LIBC_ARGUMENTS configuration option, and when returning from main(), or calling exit(), pending stdio file output is flushed and any functions registered with atexit() are invoked. This is all compliant with the ISO C standard in this respect.

This thread starts execution when the eCos scheduler is started. If the eCos kernel package is not available (and hence there is no scheduler), then cyg_iso_c_start() will invoke the main() function directly, i.e. it will not return until the main() function returns.

The main() function should be defined as the following, and if defined in a C++ file, should have “C” linkage:

```c
extern int main(
    int argc,
    char *argv[] )
```

The thread that is started by cyg_iso_c_start() can be manipulated directly, if you wish. For example you can suspend it. The kernel C API needs a handle to do this, which is available by including the following in your source code.

```c
extern cyg_handle_t cyg_libc_main_thread;
```

Then for example, you can suspend the thread with the line:
cyg_thread_suspend( cyg_libc_main_thread );

If you call cyg_iso_c_start() and do not provide your own main() function, the system will provide a main() for you which will simply return immediately.

In the default configuration, cyg_iso_c_start() is invoked automatically by the cyg_package_start() function in the infrastructure configuration. This means that in the simplest case, your program can indeed consist of simply:

```c
int main( int argc, char *argv[] )
{
    printf("Hello eCos\n");
}
```

If you override cyg_package_start() or cyg_start(), or disable the infrastructure configuration option CYGSEM_START_ISO_C_COMPATIBILITY then you must ensure that you call cyg_iso_c_start() yourself if you want to be able to have your program start at the entry point of main() automatically.
Chapter 8. C and math library overview
IV. I/O Package (Device Drivers)
Chapter 9. Introduction

The I/O package is designed as a general purpose framework for supporting device drivers. This includes all classes of drivers from simple serial to networking stacks and beyond.

Components of the I/O package, such as device drivers, are configured into the system just like all other components. Additionally, end users may add their own drivers to this set.

While the set of drivers (and the devices they represent) may be considered static, they must be accessed via an opaque “handle”. Each device in the system has a unique name and the cyg_io_lookup() function is used to map that name onto the handle for the device. This “hiding” of the device implementation allows for generic, named devices, as well as more flexibility. Also, the cyg_io_lookup() function provides drivers the opportunity to initialize the device when usage actually starts.

All devices have a name. The standard provided devices use names such as “/dev/console” and “/dev/serial0”, where the “/dev/” prefix indicates that this is the name of a device.

The entire I/O package API, as well as the standard set of provided drivers, is written in C.

Basic functions are provided to send data to and receive data from a device. The details of how this is done is left to the device [class] itself. For example, writing data to a block device like a disk drive may have different semantics than writing to a serial port.

Additional functions are provided to manipulate the state of the driver and/or the actual device. These functions are, by design, quite specific to the actual driver.

This driver model supports layering; in other words, a device may actually be created “on top of” another device. For example, the “tty” (terminal-like) devices are built on top of simple serial devices. The upper layer then has the flexibility to add features and functions not found at the lower layers. In this case the “tty” device provides for line buffering and editing not available from the simple serial drivers.

Some drivers will support visibility of the layers they depend upon. The “tty” driver allows information about the actual serial device to be manipulated by passing get/set config calls that use a serial driver “key” down to the serial driver itself.
Chapter 9. Introduction
Chapter 10. User API

All functions, except `cyg_io_lookup()` require an I/O “handle”.

All functions return a value of the type `Cyg_ErrNo`. If an error condition is detected, this value will be negative and the absolute value indicates the actual error, as specified in `cyg/error/codes.h`. The only other legal return value will be `ENOERR`. All other function arguments are pointers (references). This allows the drivers to pass information efficiently, both into and out of the driver. The most striking example of this is the “length” value passed to the read and write functions. This parameter contains the desired length of data on input to the function and the actual transferred length on return.

// Lookup a device and return its handle
Cyg_ErrNo cyg_io_lookup(
    const char *name,
    cyg_io_handle_t *handle
)

This function maps a device name onto an appropriate handle. If the named device is not in the system, then the error `-ENOENT` is returned. If the device is found, then the handle for the device is returned by way of the handle pointer `*handle`.

// Write data to a device
Cyg_ErrNo cyg_io_write(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len
)

This function sends data to a device. The size of data to send is contained in `*len` and the actual size sent will be returned in the same place.

// Read data from a device
Cyg_ErrNo cyg_io_read(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len
)

This function receives data from a device. The desired size of data to receive is contained in `*len` and the actual size obtained will be returned in the same place.

// Get the configuration of a device
Cyg_ErrNo cyg_io_get_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    void *buf,
    cyg_uint32 *len
)

This function is used to obtain run-time configuration about a device. The type of information retrieved is specified by the `key`. The data will be returned in the given buffer. The value of `*len` should contain the amount of data requested, which must be at least as large as the size appropriate to the selected key. The actual size of data retrieved is placed in `*len`. The appropriate key values differ for each driver and are all listed in the file `<cyg/io/config_keys.h>`.

// Change the configuration of a device
Cyg_ErrNo cyg_io_set_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
This function is used to manipulate or change the run-time configuration of a device. The type of information is specified by the key. The data will be obtained from the given buffer. The value of *len should contain the amount of data provided, which must match the size appropriate to the selected key. The appropriate key values differ for each driver and are all listed in the file `<cyg/io/config_keys.h>`. 
Chapter 11. Serial driver details

Two different classes of serial drivers are provided as a standard part of the eCos system. These are described as “raw serial” (serial) and “tty-like” (tty).

Raw Serial Driver

Use the include file `<cyg/io/serialio.h>` for this driver.

The raw serial driver is capable of sending and receiving blocks of raw data to a serial device. Controls are provided to configure the actual hardware, but there is no manipulation of the data by this driver.

There may be many instances of this driver in a given system, one for each serial channel. Each channel corresponds to a physical device and there will typically be a device module created for this purpose. The device modules themselves are configurable, allowing specification of the actual hardware details, as well as such details as whether the channel should be buffered by the serial driver, etc.

Runtime Configuration

Runtime configuration is achieved by exchanging data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions.

```c
typedef struct {
    cyg_serial_baud_rate_t baud;
    cyg_serial_stop_bits_t stop;
    cyg_serial_parity_t parity;
    cyg_serial_word_length_t word_length;
    cyg_uint32 flags;
} cyg_serial_info_t;
```

The field `word_length` contains the number of data bits per word (character). This must be one of the values:

- `CYGNUM_SERIAL_WORD_LENGTH_5`
- `CYGNUM_SERIAL_WORD_LENGTH_6`
- `CYGNUM_SERIAL_WORD_LENGTH_7`
- `CYGNUM_SERIAL_WORD_LENGTH_8`

The field `baud` contains a baud rate selection. This must be one of the values:

- `CYGNUM_SERIAL_BAUD_50`
- `CYGNUM_SERIAL_BAUD_75`
- `CYGNUM_SERIAL_BAUD_110`
- `CYGNUM_SERIAL_BAUD_134_5`
- `CYGNUM_SERIAL_BAUD_150`
- `CYGNUM_SERIAL_BAUD_200`
- `CYGNUM_SERIAL_BAUD_300`
- `CYGNUM_SERIAL_BAUD_600`
- `CYGNUM_SERIAL_BAUD_1200`
- `CYGNUM_SERIAL_BAUD_1800`
- `CYGNUM_SERIAL_BAUD_2400`
- `CYGNUM_SERIAL_BAUD_3600`
- `CYGNUM_SERIAL_BAUD_4800`
- `CYGNUM_SERIAL_BAUD_7200`
Chapter 11. Serial driver details

```c
CYGNUM_SERIAL_BAUD_9600
CYGNUM_SERIAL_BAUD_14400
CYGNUM_SERIAL_BAUD_19200
CYGNUM_SERIAL_BAUD_38400
CYGNUM_SERIAL_BAUD_57600
CYGNUM_SERIAL_BAUD_115200
CYGNUM_SERIAL_BAUD_234000
```

The field `stop` contains the number of stop bits. This must be one of the values:

```c
CYGNUM_SERIAL_STOP_1
CYGNUM_SERIAL_STOP_1_5
CYGNUM_SERIAL_STOP_2
```

**Note:** On most hardware, a selection of 1.5 stop bits is only valid if the word (character) length is 5.

The field `parity` contains the parity mode. This must be one of the values:

```c
CYGNUM_SERIAL_PARITY_NONE
CYGNUM_SERIAL_PARITY_EVEN
CYGNUM_SERIAL_PARITY_ODD
CYGNUM_SERIAL_PARITY_MARK
CYGNUM_SERIAL_PARITY_SPACE
```

The field `flags` is a bitmask which controls the behavior of the serial device driver. It should be built from the values `CYG_SERIAL_FLAGS_xxx` defined below:

```c
#define CYG_SERIAL_FLAGS_RTSCTS 0x0001
```

If this bit is set then the port is placed in “hardware handshake” mode. In this mode, the CTS and RTS pins control when data is allowed to be sent/received at the port. This bit is ignored if the hardware does not support this level of handshake.

```c
typedef struct {
    cyg_int32 rx_bufsize;
    cyg_int32 rx_count;
    cyg_int32 tx_bufsize;
    cyg_int32 tx_count;
} cyg_serial_buf_info_t;
```

The field `rx_bufsize` contains the total size of the incoming data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field `rx_count` contains the number of bytes currently occupied in the incoming data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field `tx_bufsize` contains the total size of the transmit data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field `tx_count` contains the number of bytes currently occupied in the transmit data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).
API Details

cyg_io_write

cyg_io_write(handle, buf, len)

Send the data from \textit{buf} to the device. The driver maintains a buffer to hold the data. The size of the intermediate buffer is configurable within the interface module. The data is not modified at all while it is being buffered. On return, \*\textit{len} contains the amount of characters actually consumed.

It is possible to configure the write call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option \texttt{CYGOPT\_IO\_SERIAL\_SUPPORT\_NONBLOCKING} to be enabled, and the specific device to be set to non-blocking mode for writes (see \texttt{cyg_io_set_config()}).

In blocking mode, the call will not return until there is space in the buffer and the entire contents of \textit{buf} have been consumed.

In non-blocking mode, as much as possible gets consumed from \textit{buf}. If everything was consumed, the call returns \texttt{ENOERR}. If only part of the \textit{buf} contents was consumed, \texttt{-EAGAIN} is returned and the caller must try again. On return, \*\textit{len} contains the number of characters actually consumed.

The call can also return \texttt{-EINTR} if interrupted via the \texttt{cyg_io_get_config()}/\texttt{ABORT} key.

cyg_io_read

cyg_io_read(handle, buf, len)

Receive data into the buffer, \textit{buf}, from the device. No manipulation of the data is performed before being transferred. An interrupt driven interface module will support data arriving when no read is pending by buffering the data in the serial driver. Again, this buffering is completely configurable. On return, \*\textit{len} contains the number of characters actually received.

It is possible to configure the read call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option \texttt{CYGOPT\_IO\_SERIAL\_SUPPORT\_NONBLOCKING} to be enabled, and the specific device to be set to non-blocking mode for reads (see \texttt{cyg_io_set_config()}).

In blocking mode, the call will not return until the requested amount of data has been read.

In non-blocking mode, data waiting in the device buffer is copied to \textit{buf}, and the call returns immediately. If there was enough data in the buffer to fulfill the request, \texttt{ENOERR} is returned. If only part of the request could be fulfilled, \texttt{-EAGAIN} is returned and the caller must try again. On return, \*\textit{len} contains the number of characters actually received.

The call can also return \texttt{-EINTR} if interrupted via the \texttt{cyg_io_get_config()}/\texttt{ABORT} key.

cyg_io_get_config

cyg_io_get_config(handle, key, buf, len)

This function returns current [runtime] information about the device and/or driver.
CYG_IO_GET_CONFIG_SERIAL_INFO

Buf type:

cyg_serial_info_t

Function:

This function retrieves the current state of the driver and hardware. This information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control.

CYG_IO_GET_CONFIG_SERIAL_BUFFER_INFO

Buf type:

cyg_serial_buf_info_t

Function:

This function retrieves the current state of the software buffers in the serial drivers. For both receive and transmit buffers it returns the total buffer size and the current number of bytes occupied in the buffer. It does not take into account any buffering such as FIFOs or holding registers that the serial device itself may have.

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_DRAIN

Buf type:

void *

Function:

This function waits for any buffered output to complete. This function only completes when there is no more data remaining to be sent to the device.

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_FLUSH

Buf type:

void *

Function:

This function discards any buffered output for the device.
Chapter 11. Serial driver details

CYG_IO_GET_CONFIG_SERIAL_INPUT_DRAIN

Buf type:
  void *

Function:
  This function discards any buffered input for the device.

CYG_IO_GET_CONFIG_SERIAL_ABORT

Buf type:
  void *

Function:
  This function will cause any pending read or write calls on this device to return with -EABORT.

CYG_IO_GET_CONFIG_SERIAL_READ_BLOCKING

Buf type:
  cyg_uint32 (values 0 or 1)

Function:
  This function will read back the blocking-mode setting for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

CYG_IO_GET_CONFIG_SERIAL_WRITE_BLOCKING

Buf type:
  cyg_uint32 (values 0 or 1)

Function:
  This function will read back the blocking-mode setting for write calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.
**cyg_io_set_config**

cyg_io_set_config(handle, key, buf, len)

This function is used to update or change runtime configuration of a port.

**CYG_IO_SET_CONFIG_SERIAL_INFO**

| Buf type: | cyg_serial_info_t |
| Function: | This function updates the information for the driver and hardware. The information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control. |

**CYG_IO_SET_CONFIG_SERIAL_READ_BLOCKING**

| Buf type: | cyg_uint32 (values 0 or 1) |
| Function: | This function will set the blocking-mode for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled. |

**CYG_IO_SET_CONFIG_SERIAL_WRITE_BLOCKING**

| Buf type: | cyg_uint32 (values 0 or 1) |
| Function: | This function will set the blocking-mode for write calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled. |

**TTY driver**

Use the include file `<cyg/io/ttyio.h>` for this driver.
Chapter 11. Serial driver details

This driver is built on top of the simple serial driver and is typically used for a device that interfaces with humans such as a terminal. It provides some minimal formatting of data on output and allows for line-oriented editing on input.

**Runtime configuration**

Runtime configuration is achieved by exchanging data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions.

```c
typedef struct {
    cyg_uint32 tty_out_flags;
    cyg_uint32 tty_in_flags;
} cyg_tty_info_t;
```

The field `tty_out_flags` is used to control what happens to data as it is send to the serial port. It contains a bitmap comprised of the bits as defined by the `CYG_TTY_OUT_FLAGS_xxx` values below.

```c
#define CYG_TTY_OUT_FLAGS_CRLF 0x0001 // Map '\n' = > '\r\n' on output
```

If this bit is set in `tty_out_flags`, any occurrence of the character "\n" will be replaced by the sequence "\r\n" before being sent to the device.

The field `tty_in_flags` is used to control how data is handled as it comes from the serial port. It contains a bitmap comprised of the bits as defined by the `CYG_TTY_IN_FLAGS_xxx` values below.

```c
#define CYG_TTY_IN_FLAGS_CR 0x0001 // Map '\r' = > '\n' on input
#define CYG_TTY_IN_FLAGS_CRLF 0x0002 // Map '\r\n' = > '\n' on input
#define CYG_TTY_IN_FLAGS_ECHO 0x0004 // Echo characters as processed
#define CYG_TTY_IN_FLAGS_BINARY 0x0008 // No input processing
```

If this bit is set in `tty_in_flags`, the character "\r" ("return" or “enter” on most keyboards) will be mapped to "\n".

If this bit is set in `tty_in_flags`, the character sequence "\r\n" (often sent by DOS/Windows based terminals) will be mapped to "\n".

If this bit is set in `tty_in_flags`, characters will be echoed back to the serial port as they are processed.

If this bit is set in `tty_in_flags`, the input will not be manipulated in any way before being placed in the user’s buffer.

**API details**

```c
cyg_io_read(handle, buf, len)
```

This function is used to read data from the device. In the default case, data is read until an end-of-line character ("\n" or "\r") is read. Additionally, the characters are echoed back to the [terminal] device. Minimal editing of the input is also supported.

**Note:** When connecting to a remote target via GDB it is not possible to provide console input while GDB is connected. The GDB remote protocol does not support input. Users must disconnect from GDB if this functionality is required.
Chapter 11. Serial driver details

```c
syscalls

cyg_io_write(handle, buf, len)
```

This function is used to send data to the device. In the default case, the end-of-line character "\n" is replaced by the sequence "\r\n".

```c
cyg_io_get_config(handle, key, buf, len)
```

This function is used to get information about the channel’s configuration at runtime.

```c
cyg_io_set_config(handle, key, buf, len)
```

This function is used to modify the channel’s configuration at runtime.

---

**Buf type:**

```c
cyg_tty_info_t
```

**Function:**

This function retrieves the current state of the driver.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.

---

**Buf type:**

```c
cyg_tty_info_t
```

**Function:**

This function changes the current state of the driver.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.
Chapter 12. How to Write a Driver

A device driver is nothing more than a named entity that supports the basic I/O functions - read, write, get config, and set config. Typically a device driver also uses and manages interrupts from the device. While the interface is generic and device driver independent, the actual driver implementation is completely up to the device driver designer.

That said, the reason for using a device driver is to provide access to a device from application code in as general purpose a fashion as reasonable. Most driver writers are also concerned with making this access as simple as possible while being as efficient as possible.

Most device drivers are concerned with the movement of information, for example data bytes along a serial interface, or packets in a network. In order to make the most efficient use of system resources, interrupts are used. This will allow other application processing to take place while the data transfers are under way, with interrupts used to indicate when various events have occurred. For example, a serial port typically generates an interrupt after a character has been sent “down the wire” and the interface is ready for another. It makes sense to allow further application processing while the data is being sent since this can take quite a long time. The interrupt can be used to allow the driver to send a character as soon as the current one is complete, without any active participation by the application code.

The main building blocks for device drivers are found in the include file: `<cyg/io/devtab.h>`

All device drivers in eCos are described by a device table entry, using the `cyg_devtab_entry_t` type. The entry should be created using the `DEVTAB_ENTRY()` macro, like this:

```
DEVTAB_ENTRY(l, name, dep_name, handlers, init, lookup, priv)
```

**Arguments**

- **l**
  - The "C" label for this device table entry.

- **name**
  - The "C" string name for the device.

- **dep_name**
  - For a layered device, the "C" string name of the device this device is built upon.

- **handlers**
  - A pointer to the I/O function "handlers" (see below).

- **init**
  - A function called when eCos is initialized. This function can query the device, setup hardware, etc.

- **lookup**
  - A function called when `cyg_io_lookup()` is called for this device.

- **priv**
  - A placeholder for any device specific data required by the driver.
Chapter 12. How to Write a Driver

The interface to the driver is through the `handlers` field. This is a pointer to a set of functions which implement the various `cyg_io_XXX()` routines. This table is defined by the macro:

```
DEVIO_TABLE(l, write, read, get_config, set_config)
```

**Arguments**

- `l`
  The "C" label for this table of handlers.
- `write`
  The function called as a result of `cyg_io_write()`.
- `read`
  The function called as a result of `cyg_io_read()`.
- `get_config`
  The function called as a result of `cyg_io_get_config()`.
- `set_config`
  The function called as a result of `cyg_io_set_config()`.

When eCos is initialized (sometimes called “boot” time), the `init()` function is called for all devices in the system. The `init()` function is allowed to return an error in which case the device will be placed “off line” and all I/O requests to that device will be considered in error.

The `lookup()` function is called whenever the `cyg_io_lookup()` function is called with this device name. The lookup function may cause the device to come “on line” which would then allow I/O operations to proceed. Future versions of the I/O system will allow for other states, including power saving modes, etc.

---

**How to Write a Serial Hardware Interface Driver**

The standard serial driver supplied with eCos is structured as a hardware independent portion and a hardware dependent interface module. To add support for a new serial port, the user should be able to use the existing hardware independent portion and just add their own interface driver which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the serial driver and serial implementation modules are contained in the file `<cyg/io/serial.h>`

**Note:** In the sections below we use the notation `<xx>` to mean a module specific value, referred to as “xx” below.

**DevTab Entry**

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```
DEVTAB_ENTRY(<module_name>,
             <device_name>,
             ...
```

---

102
Chapter 12. How to Write a Driver

0,
&serial_devio,
<<module_init>>, 
<<module_lookup>>, 
<<serial_channel>>
);

Arguments

module_name
The "C" label for this devtab entry

device_name
The "C" string for the device. E.g. /dev/serial0.

serial_devio
The table of I/O functions. This set is defined in the hardware independent serial driver and should be used.

module_init
The module initialization function.

module_lookup
The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the port, etc.

serial_channel
This table (defined below) contains the interface between the interface module and the serial driver proper.

Serial Channel Structure

Each serial device must have a "serial channel". This is a set of data which describes all operations on the device. It also contains buffers, etc., if the device is to be buffered. The serial channel is created by the macro:

SERIAL_CHANNEL_USING_INTERRUPTS(l, funs, dev_priv, baud, stop, parity, word_length,
flags, out_buf, out_buflen, in_buf, in_buflen)

Arguments

l
The "C" label for this structure.

funs
The set of interface functions (see below).

dev_priv
A placeholder for any device specific data for this channel.
Chapter 12. How to Write a Driver

**baud**

The initial baud rate value (cyg_serial_baud_t).

**stop**

The initial stop bits value (cyg_serial_stop_bits_t).

**parity**

The initial parity mode value (cyg_serial_parity_t).

**word_length**

The initial word length value (cyg_serial_word_length_t).

**flags**

The initial driver flags value.

**out_buf**

Pointer to the output buffer. NULL if none required.

**out buflen**

The length of the output buffer.

**in_buf**

Pointer to the input buffer. NULL if none required.

**in buflen**

The length of the input buffer.

If either buffer length is zero, no buffering will take place in that direction and only polled mode functions will be used.

The interface from the hardware independent driver into the hardware interface module is contained in the `funs` table. This is defined by the macro:

### Serial Functions Structure

```
SERIAL_FUNS(l, putc, getc, set_config, start_xmit, stop_xmit)
```

#### Arguments

**l**

The "C" label for this structure.

**putc**

```
bool (*putc)(serial_channel *priv, unsigned char c)
```

This function sends one character to the interface. It should return `true` if the character is actually consumed. It should return `false` if there is no space in the interface.
Chapter 12. How to Write a Driver

`getc`

```c
unsigned char (*getc)(serial_channel *priv)
```

This function fetches one character from the interface. It will be only called in a non-interrupt driven mode, thus it should wait for a character by polling the device until ready.

`set_config`

```c
bool (*set_config)(serial_channel *priv, cyg_serial_info_t *config)
```

This function is used to configure the port. It should return `true` if the hardware is updated to match the desired configuration. It should return `false` if the port cannot support some parameter specified by the given configuration. E.g. selecting 1.5 stop bits and 8 data bits is invalid for most serial devices and should not be allowed.

`start_xmit`

```c
void (*start_xmit)(serial_channel *priv)
```

In interrupt mode, turn on the transmitter and allow for transmit interrupts.

`stop_xmit`

```c
void (*stop_xmit)(serial_channel *priv)
```

In interrupt mode, turn off the transmitter.

**Callbacks**

The device interface module can execute functions in the hardware independent driver via `chan->callbacks`. These functions are available:

```c
void (*serial_init)( serial_channel *chan )
```

This function is used to initialize the serial channel. It is only required if the channel is being used in interrupt mode.

```c
void (*xmt_char)( serial_channel *chan )
```

This function would be called from an interrupt handler after a transmit interrupt indicating that additional characters may be sent. The upper driver will call the `putc` function as appropriate to send more data to the device.

```c
void (*rcv_char)( serial_channel *chan, unsigned char c )
```

This function is used to tell the driver that a character has arrived at the interface. This function is typically called from the interrupt handler.

Furthermore, if the device has a FIFO it should require the hardware independent driver to provide block transfer functionality (driver CDL should include "implements CYGINT_IO_SERIAL_BLOCK_TRANSFER"). In that case, the following functions are available as well:

```c
bool (*data_xmt_req)(serial_channel *chan,
    int space,
    int* chars_avail,
    unsigned char** chars)
```

```c
void (*data_xmt_done)(serial_channel *chan)
```

105
Chapter 12. How to Write a Driver

Instead of calling \texttt{xmt\_char()} to get a single character for transmission at a time, the driver should call \texttt{data\_xmt\_req()} in a loop, requesting character blocks for transfer. Call with a \texttt{space} argument of how much space there is available in the FIFO.

If the call returns \texttt{true}, the driver can read \texttt{chars\_avail} characters from \texttt{chars} and copy them into the FIFO. If the call returns \texttt{false}, there are no more buffered characters and the driver should continue without filling up the FIFO.

When all data has been unloaded, the driver must call \texttt{data\_xmt\_done()}.

\begin{verbatim}
bool (*data\_rcv\_req)(serial\_channel *chan,
                   int avail,
                   int* space\_avail,
                   unsigned char** space)

void (*data\_rcv\_done)(serial\_channel *chan)
\end{verbatim}

Instead of calling \texttt{rcv\_char()} with a single character at a time, the driver should call \texttt{data\_rcv\_req()} in a loop, requesting space to unload the FIFO to. \texttt{avail} is the number of characters the driver wishes to unload.

If the call returns \texttt{true}, the driver can copy \texttt{space\_avail} characters to \texttt{space}.

If the call returns \texttt{false}, the input buffer is full. It is up to the driver to decide what to do in that case (callback functions for registering overflow are being planned for later versions of the serial driver).

When all data has been unloaded, the driver must call \texttt{data\_rcv\_done()}.

Serial testing with \texttt{ser\_filter}

Rationale

Since some targets only have one serial connection, a serial testing harness needs to be able to share the connection with GDB (however, the test and GDB can also run on separate lines).

The \texttt{serial filter} (\texttt{ser\_filter}) sits between the serial port and GDB and monitors the exchange of data between GDB and the target. Normally, no changes are made to the data.

When a test request packet is sent from the test on the target, it is intercepted by the filter. The filter and target then enter a loop, exchanging protocol data between them which GDB never sees.

In the event of a timeout, or a crash on the target, the filter falls back into its pass-through mode. If this happens due to a crash it should be possible to start regular debugging with GDB. The filter will stay in the pass-through mode until GDB disconnects.

The Protocol

The protocol commands are prefixed with an "@" character which the serial filter is looking for. The protocol commands include:

\texttt{PING}

Allows the test on the target to probe for the filter. The filter responds with \texttt{OK}, while GDB would just ignore the command. This allows the tests to do nothing if they require the filter and it is not present.
Chapter 12. How to Write a Driver

CONFIG

Requests a change of serial line configuration. Arguments to the command specify baud rate, data bits, stop bits, and parity. [This command is not fully implemented yet - there is no attempt made to recover if the new configuration turns out to cause loss of data.]

BINARY

Requests data to be sent from the filter to the target. The data is checksummed, allowing errors in the transfer to be detected. Sub-options of this command control how the data transfer is made:

NO_ECHO

(serial driver receive test) Just send data from the filter to the target. The test verifies the checksum and PASS/FAIL depending on the result.

EOP_ECHO

(serial driver half-duplex receive and send test) As NO_ECHO but the test echoes back the data to the filter. The filter does a checksum on the received data and sends the result to the target. The test PASS/FAIL depending on the result of both checksum verifications.

DUPLEX_ECHO

(serial driver duplex receive and send test) Smaller packets of data are sent back and forth in a pattern that ensures that the serial driver will be both sending and receiving at the same time. Again, checksums are computed and verified resulting in PASS/FAIL.

TEXT

This is a test of the text translations in the TTY layer. Requests a transfer of text data from the target to the filter and possibly back again. The filter treats this as a binary transfer, while the target may be doing translations on the data. The target provides the filter with checksums for what it should expect to see. This test is not implemented yet.

The above commands may be extended, and new commands added, as required to test (new) parts of the serial drivers in eCos.

The Serial Tests

The serial tests are built as any other eCos test. After running the make tests command, the tests can be found in install/tests/io_serial/

serial1

A simple API test.

serial2

A simple serial send test. It writes out two strings, one raw and one encoded as a GDB O-packet

serial3 [ requires the serial filter ]

This tests the half-duplex send and receive capabilities of the serial driver.
Chapter 12. How to Write a Driver

serial4 [ requires the serial filter ]
This test attempts to use a few different serial configurations, testing the driver’s configuration/setup functionality.

serial5 [ requires the serial filter ]
This test attempts to use a few different serial configurations, testing the driver’s configuration/setup functionality.

All tests should complete in less than 30 seconds.

Serial Filter Usage

Running the ser_filter program with no (or wrong) arguments results in the following output:

Usage: ser_filter [-t -S] TcpIPport SerialPort BaudRate
or: ser_filter -n [-t -S] SerialPort BaudRate
-t: Enable tracing.
-S: Output data read from serial line.
-c: Output data on console instead of via GDB.
-n: No GDB.

The normal way to use it with GDB is to start the filter:

$ ser_filter -t 9000 com1 38400

In this case, the filter will be listening on port 9000 and connect to the target via the serial port COM1 at 38400 baud. On a UNIX host, replace "COM1" with a device such as "/dev/ttyS0".

The -t option enables tracing which will cause the filter to describe its actions on the console.

Now start GDB with one of the tests as an argument:

$ mips-tx39-elf-gdb -nw install/tests/io_serial/serial3

Then connect to the filter:

(gdb) target remote localhost:9000

This should result in a connection in exactly the same way as if you had connected directly to the target on the serial line.

(gdb) c

Which should result in output similar to the below:

Continuing.
INFO: <BINARY:16:1!>
PASS: <Binary test completed>
INFO: <BINARY:128:1!>
PASS: <Binary test completed>
INFO: <BINARY:256:1!>
PASS: <Binary test completed>
INFO: <BINARY:1024:1!>
PASS: <Binary test completed>
INFO: <BINARY:512:0!>
PASS: <Binary test completed>
...
PASS: <Binary test completed>
INFO: <BINARY:16384:0!>
If any of the individual tests fail the testing will terminate with a FAIL.

With tracing enabled, you would also see the filter’s status output:

The PING command sent from the target to determine the presence of the filter:

```
[400 11:35:16] Dispatching command PING
[400 11:35:16] Responding with status OK
```

Each of the binary commands result in output similar to:

```
[400 11:35:16] Dispatching command BINARY
[400 11:35:16] Binary data (Size:16, Flags:1).
[400 11:35:16] Reading 16 bytes from target.
[400 11:35:16] Done. in_crc 170231, out_crc 170231.
[400 11:35:16] Responding with status OK
[400 11:35:16] Received DONE from target.
```

This tracing output is normally sent as O-packets to GDB which will display the tracing text. By using the `-c` option, the tracing text can be redirected to the console from which `ser_filter` was started.

### A Note on Failures

A serial connection (especially when driven at a high baud rate) can garble the transmitted data because of noise from the environment. It is not the job of the serial driver to ensure data integrity - that is the job of protocols layering on top of the serial driver.

In the current implementation the serial tests and the serial filter are not resilient to such data errors. This means that the test may crash or hang (possibly without reporting a FAIL). It also means that you should be aware of random errors - a FAIL is not necessarily caused by a bug in the serial driver.

Ideally, the serial testing infrastructure should be able to distinguish random errors from consistent errors - the former are most likely due to noise in the transfer medium, while the latter are more likely to be caused by faulty drivers. The current implementation of the infrastructure does not have this capability.

### Debugging

If a test fails, the serial filter’s output may provide some hints about what the problem is. If the option `-S` is used when starting the filter, data received from the target is printed out:

```
[400 11:35:16] 0000 50 41 53 53 3a 3c 42 69 'PASS:<Bi'
[400 11:35:16] 0008 6e 61 72 79 20 74 65 73 'nary.tes'
[400 11:35:16] 0010 74 65 64 3e 0d 0a 49 4e 'ted..IN'
[400 11:35:16] 0020 74 72 79 3a 31 32 38 3a 'RY:128:'
[400 11:35:16] 0030 31 21 .. .. .. .. .. .. '1!' 109
```
In the case of an error during a testing command the data received by the filter will be printed out, as will the data that was expected. This allows the two data sets to be compared which may give some idea of what the problem is.
Chapter 13. Device Driver Interface to the Kernel

This chapter describes the API that device drivers may use to interact with the kernel and HAL. It is primarily concerned with the control and management of interrupts and the synchronization of ISRs, DSRs and threads.

The same API will be present in configurations where the kernel is not present. In this case the functions will be supplied by code acting directly on the HAL.

Interrupt Model

eCos presents a three level interrupt model to device drivers. This consists of Interrupt Service Routines (ISRs) that are invoked in response to a hardware interrupt; Deferred Service Routines (DSRs) that are invoked in response to a request by an ISR; and threads that are the clients of the driver.

Hardware interrupts are delivered with minimal intervention to an ISR. The HAL decodes the hardware source of the interrupt and calls the ISR of the attached interrupt object. This ISR may manipulate the hardware but is only allowed to make a restricted set of calls on the driver API. When it returns, an ISR may request that its DSR should be scheduled to run.

A DSR will be run when it is safe to do so without interfering with the scheduler. Most of the time the DSR will run immediately after the ISR, but if the current thread is in the scheduler, it will be delayed until the thread is finished. A DSR is allowed to make a larger set of driver API calls, including, in particular, being able to call cyg_drv_cond_signal() to wake up waiting threads.

Finally, threads are able to make all API calls and in particular are allowed to wait on mutexes and condition variables.

For a device driver to receive interrupts it must first define ISR and DSR routines as shown below, and then call cyg_drv_interrupt_create(). Using the handle returned, the driver must then call cyg_drv_interrupt_attach() to actually attach the interrupt to the hardware vector.

Synchronization

There are three levels of synchronization supported:

1. Synchronization with ISRs. This normally means disabling interrupts to prevent the ISR running during a critical section. In an SMP environment, this will also require the use of a spinlock to synchronize with ISRs, DSRs or threads running on other CPUs. This is implemented by the cyg_drv_isr_lock() and cyg_drv_isr_unlock() functions. This mechanism should be used sparingly and for short periods only. For finer grained synchronization, individual spinlocks are also supplied.

2. Synchronization with DSRs. This will be implemented in the kernel by taking the scheduler lock to prevent DSRs running during critical sections. In non-kernel configurations it will be implemented by non-kernel code. This is implemented by the cyg_drv_dsr_lock() and cyg_drv_dsr_unlock() functions. As with ISR synchronization, this mechanism should be used sparingly. Only DSRs and threads may use this synchronization mechanism. ISRs are not allowed to do this.

3. Synchronization with threads. This is implemented with mutexes and condition variables. Only threads may lock the mutexes and wait on the condition variables, although DSRs may signal condition variables.
Any data that is accessed from more than one level must be protected against concurrent access. Data that is accessed by ISRs must be protected with the ISR lock, or a spinlock at all times, even in ISRs. Data that is shared between DSRs and threads should be protected with the DSR lock. Data that is only accessed by threads must be protected with mutexes.

SMP Support

Some eCos targets contain support for Symmetric Multi-Processing (SMP) configurations, where more than one CPU may be present. This option has a number of ramifications for the way in which device drivers must be written if they are to be SMP-compatible.

Since it is possible for the ISR, DSR and thread components of a device driver to execute on different CPUs, it is important that SMP-compatible device drivers use the driver API routines correctly.

Synchronization between threads and DSRs continues to require that the thread-side code use `cyg_drv_dsr_lock()` and `cyg_drv_dsr_unlock()` to protect access to shared data. While it is not strictly necessary for DSR code to claim the DSR lock, since DSRs are run with it claimed already, it is good practice to do so.

Synchronization between ISRs and DSRs or threads requires that access to sensitive data be protected, in all places, by calls to `cyg_drv_isr_lock()` and `cyg_drv_isr_unlock()`. Disabling or masking interrupts is not adequate, since the thread or DSR may be running on a different CPU and interrupt enable/disable only work on the current CPU.

The ISR lock, for SMP systems, not only disables local interrupts, but also acquires a spinlock to protect against concurrent access from other CPUs. This is necessary because ISRs are not run with the scheduler lock claimed. Hence they can run in parallel with the other components of the device driver.

The ISR lock provided by the driver API is just a shared spinlock that is available for use by all drivers. If a driver needs to implement a finer grain of locking, it can use private spinlocks, accessed via the `cyg_drv_spinlock_*()` functions.

Device Driver Models

There are several ways in which device drivers may be built. The exact model chosen will depend on the properties of the device and the behavior desired. There are three basic models that may be adopted.

The first model is to do all device processing in the ISR. When it is invoked the ISR programs the device hardware directly and accesses data to be transferred directly in memory. The ISR should also call `cyg_drv_interrupt_acknowledge()`. When it is finished it may optionally request that its DSR be invoked.

The DSR does nothing but call `cyg_drv_cond_signal()` to cause a thread to be woken up. Thread level code must call `cyg_drv_isr_lock()`, or `cyg_drv_interrupt_mask()` to prevent ISRs running while it manipulates shared memory.

The second model is to defer device processing to the DSR. The ISR simply prevents further delivery of interrupts by either programming the device, or by calling `cyg_drv_interrupt_mask()`. It must then call `cyg_drv_interrupt_acknowledge()` to allow other interrupts to be delivered and then request that its DSR be called. When the DSR runs it does the majority of the device handling, optionally signals a condition variable to wake a thread, and finishes by calling `cyg_drv_interrupt_unmask()` to re-allow device interrupts. Thread level code uses `cyg_drv_dsr_lock()` to prevent DSRs running while it manipulates shared memory. The eCos serial device drivers use this approach.
The third model is to defer device processing even further to a thread. The ISR behaves exactly as in the previous model and simply blocks and acknowledges the interrupt before requesting that the DSR run. The DSR itself only calls \texttt{cyg_drv_cond_signal()} to wake the thread. When the thread awakens it performs all device processing, and has full access to all kernel facilities while it does so. It should finish by calling \texttt{cyg_drv_interrupt_unmask()} to re-allow device interrupts. The eCos ethernet device drivers are written to this model.

The first model is good for devices that need immediate processing and interact infrequently with thread level. The second model trades a little latency in dealing with the device for a less intrusive synchronization mechanism. The last model allows device processing to be scheduled with other threads and permits more complex device handling.

### Synchronization Levels

Since it would be dangerous for an ISR or DSR to make a call that might reschedule the current thread (by trying to lock a mutex for example) all functions in this API have an associated synchronization level. These levels are:

**Thread**

This function may only be called from within threads. This is usually the client code that makes calls into the device driver. In a non-kernel configuration, this will be code running at the default non-interrupt level.

**DSR**

This function may be called by either DSR or thread code.

**ISR**

This function may be called from ISR, DSR or thread code.

The following table shows, for each API function, the levels at which it may be called:

<table>
<thead>
<tr>
<th>Function</th>
<th>Callable from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISR</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>\texttt{cygDrvIsrLock}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvIsrUnlock}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockInit}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockDestroy}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockSpin}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockClear}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockTry}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockTest}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockSpinIntsave}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvSpinLockClearIntsave}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvDarLock}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvDarUnlock}</td>
<td>X</td>
</tr>
<tr>
<td>\texttt{cygDrvMutexInit}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvMutexDestroy}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvMutexLock}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvMutexTryLock}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvMutexUnlock}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvMutexRelease}</td>
<td></td>
</tr>
<tr>
<td>\texttt{cygDrvCondInit}</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 13. Device Driver Interface to the Kernel

The API

This section details the Driver Kernel Interface. Note that most of these functions are identical to Kernel C API calls, and will in most configurations be wrappers for them. In non-kernel configurations they will be supported directly by the HAL, or by code to emulate the required behavior.

This API is defined in the header file `<cyg/hal/drv_api.h>`.

**cyg_drv_isr_lock**

Function:

```c
void cyg_drv_isr_lock()
```

Arguments:

None

Result:

None

Level:

ISR

Description:

Disables delivery of interrupts, preventing all ISRs running. This function maintains a counter of the number of times it is called.

**cyg_drv_isr_unlock**

Function:

```c
void cyg_drv_isr_unlock()
```
Arguments:
None

Result:
None

Level:
ISR

Description:
Re-enables delivery of interrupts, allowing ISRs to run. This function decrements the counter maintained by cyg_drv_isr_lock(), and only re-allows interrupts when it goes to zero.

cyg_drv_spinlock_init

Function:

void cyg_drv_spinlock_init(cyg_spinlock_t *lock, cyg_bool_t locked )

Arguments:

lock - pointer to spinlock to initialize
locked - initial state of lock

Result:
None

Level:
Thread

Description:
Initialize a spinlock. The locked argument indicates how the spinlock should be initialized: TRUE for locked or FALSE for unlocked state.

cyg_drv_spinlock_destroy

Function:

void cyg_drv_spinlock_destroy(cyg_spinlock_t *lock )

Arguments:

lock - pointer to spinlock destroy

Result:
None

Level:
Thread
Chapter 13. Device Driver Interface to the Kernel

Description:
Destroy a spinlock that is no longer of use. There should be no CPUs attempting to claim the lock at the
time this function is called, otherwise the behavior is undefined.

cyg_drv_spinlock_spin

Function:
void cyg_drv_spinlock_spin(cyg_spinlock_t *lock )

Arguments:
lock - pointer to spinlock to claim

Result:
None

Level:
ISR

Description:
Claim a spinlock, waiting in a busy loop until it is available. Wherever this is called from, this operation
effectively pauses the CPU until it succeeds. This operations should therefore be used sparingly, and in
situations where deadlocks/livelocks cannot occur. Also see cyg_drv_spinlock_spin_intsave().

cyg_drv_spinlock_clear

Function:
void cyg_drv_spinlock_clear(cyg_spinlock_t *lock )

Arguments:
lock - pointer to spinlock to clear

Result:
None

Level:
ISR

Description:
Clear a spinlock. This clears the spinlock and allows another CPU to claim it. If there is more than one
CPU waiting in cyg_drv_spinlock_spin() then just one of them will be allowed to proceed.
cyg_drv_spinlock_try

Function:

cyg_bool_t cyg_drv_spinlock_try(cyg_spinlock_t *lock )

Arguments:

lock - pointer to spinlock to try

Result:

TRUE if the spinlock was claimed, FALSE otherwise.

Level:

ISR

Description:

Try to claim the spinlock without waiting. If the spinlock could be claimed immediately then TRUE is returned. If the spinlock is already claimed then the result is FALSE.

cyg_drv_spinlock_test

Function:

cyg_bool_t cyg_drv_spinlock_test(cyg_spinlock_t *lock )

Arguments:

lock - pointer to spinlock to test

Result:

TRUE if the spinlock is available, FALSE otherwise.

Level:

ISR

Description:

Inspect the state of the spinlock. If the spinlock is not locked then the result is TRUE. If it is locked then the result will be FALSE.

cyg_drv_spinlock_spin_intsave

Function:

void cyg_drv_spinlock_spin_intsave(cyg_spinlock_t *lock, cyg_addrword_t *istate )

Arguments:

lock - pointer to spinlock to claim

istate - pointer to interrupt state save location
Chapter 13. Device Driver Interface to the Kernel

Result:
None

Level:
ISR

Description:
This function behaves exactly like cyg_drv_spinlock_spin() except that it also disables interrupts before attempting to claim the lock. The current interrupt enable state is saved in *istate. Interrupts remain disabled once the spinlock had been claimed and must be restored by calling cyg_drv_spinlock_clear_intsave().

In general, device drivers should use this function to claim and release spinlocks rather than the non-_intsave() variants, to ensure proper exclusion with code running on both other CPUs and this CPU.

cyg_drv_spinlock_clear_intsave

Function:

void cyg_drv_spinlock_clear_intsave( cyg_spinlock_t *lock,
                     cyg_addrword_t istate )

Arguments:

lock - pointer to spinlock to clear
istate - interrupt state to restore

Result:
None

Level:
ISR

Description:
This function behaves exactly like cyg_drv_spinlock_clear() except that it also restores an interrupt state saved by cyg_drv_spinlock_spin_intsave(). The istate argument must have been initialized by a previous call to cyg_drv_spinlock_spin_intsave().

cyg_drv_dsr_lock

Function:

void cyg_drv_dsr_lock()

Arguments:
None

Result:
None
Level:
  DSR

Description:
  Disables scheduling of DSRs. This function maintains a counter of the number of times it has been called.

\textbf{cygDrvDsrUnlock}

Function:
  \texttt{void cygDrvDsrUnlock()}

Arguments:
  None

Result:
  None

Level:
  DSR

Description:
  Re-enables scheduling of DSRs. This function decrements the counter incremented by \texttt{cygDrvDsrLock()}. DSRs are only allowed to be delivered when the counter goes to zero.

\textbf{cygDrvMutexInit}

Function:
  \texttt{void cygDrvMutexInit(cygDrvMutex_t *mutex)}

Arguments:
  \texttt{mutex} - pointer to mutex to initialize

Result:
  None

Level:
  Thread

Description:
  Initialize the mutex pointed to by the \texttt{mutex} argument.
### cyg_drv_mutex_destroy

**Function:**

```c
void cyg_drv_mutex_destroy( cyg_drv_mutex_t *mutex )
```

**Arguments:**

- `mutex` - pointer to mutex to destroy

**Result:**

None

**Level:**

Thread

**Description:**

Destroy the mutex pointed to by the `mutex` argument. The mutex should be unlocked and there should be no threads waiting to lock it when this call is made.

### cyg_drv_mutex_lock

**Function:**

```c
cyg_bool cyg_drv_mutex_lock( cyg_drv_mutex_t *mutex )
```

**Arguments:**

- `mutex` - pointer to mutex to lock

**Result:**

TRUE if the thread has claimed the lock, FALSE otherwise.

**Level:**

Thread

**Description:**

Attempt to lock the mutex pointed to by the `mutex` argument. If the mutex is already locked by another thread then this thread will wait until that thread is finished. If the result from this function is FALSE then the thread was broken out of its wait by some other thread. In this case the mutex will not have been locked.

### cyg_drv_mutex_trylock

**Function:**

```c
cyg_bool cyg_drv_mutex_trylock( cyg_drv_mutex_t *mutex )
```
Arguments:

mutex - pointer to mutex to lock

Result:

TRUE if the mutex has been locked, FALSE otherwise.

Level:

Thread

Description:

Attempt to lock the mutex pointed to by the mutex argument without waiting. If the mutex is already locked by some other thread then this function returns FALSE. If the function can lock the mutex without waiting, then TRUE is returned.

**cyg_drv_mutex_unlock**

Function:

```c
void cyg_drv_mutex_unlock( cyg_drv_mutex_t *mutex )
```

Arguments:

mutex - pointer to mutex to unlock

Result:

None

Level:

Thread

Description:

Unlock the mutex pointed to by the mutex argument. If there are any threads waiting to claim the lock, one of them is woken up to try and claim it.

**cyg_drv_mutex_release**

Function:

```c
void cyg_drv_mutex_release( cyg_drv_mutex_t *mutex )
```

Arguments:

mutex - pointer to mutex to release

Result:

None

Level:

Thread
Chapter 13. Device Driver Interface to the Kernel

Description:

Release all threads waiting on the mutex pointed to by the `mutex` argument. These threads will return from `cyg_drv_mutex_lock()` with a FALSE result and will not have claimed the mutex. This function has no effect on any thread that may have the mutex claimed.

### cyg_drv_cond_init

Function:

```c
void cyg_drv_cond_init( cyg_drv_cond_t *cond, cyg_drv_mutex_t *mutex )
```

Arguments:

- `cond` - condition variable to initialize
- `mutex` - mutex to associate with this condition variable

Result:

None

Level:

Thread

Description:

Initialize the condition variable pointed to by the `cond` argument. The `mutex` argument must point to a mutex with which this condition variable is associated. A thread may only wait on this condition variable when it has already locked the associated mutex. Waiting will cause the mutex to be unlocked, and when the thread is reawakened, it will automatically claim the mutex before continuing.

### cyg_drv_cond_destroy

Function:

```c
void cyg_drv_cond_destroy( cyg_drv_cond_t *cond )
```

Arguments:

- `cond` - condition variable to destroy

Result:

None

Level:

Thread

Description:

Destroy the condition variable pointed to by the `cond` argument.
cyg_drv_cond_wait

Function:

```c
void cyg_drv_cond_wait( cyg_drv_cond_t *cond )
```

Arguments:

- `cond` - condition variable to wait on

Result:

None

Level:

Thread

Description:

Wait for a signal on the condition variable pointed to by the `cond` argument. The thread must have locked the associated mutex, supplied in `cyg_drv_cond_init()`, before waiting on this condition variable. While the thread waits, the mutex will be unlocked, and will be re-locked before this function returns. It is possible for threads waiting on a condition variable to occasionally wake up spuriously. For this reason it is necessary to use this function in a loop that re-tests the condition each time it returns. Note that this function performs an implicit scheduler unlock/relock sequence, so that it may be used within an explicit `cyg_drv_dsr_lock() ... cyg_drv_dsr_unlock()` structure.

cyg_drv_cond_signal

Function:

```c
void cyg_drv_cond_signal( cyg_drv_cond_t *cond )
```

Arguments:

- `cond` - condition variable to signal

Result:

None

Level:

DSR

Description:

Signal the condition variable pointed to by the `cond` argument. If there are any threads waiting on this variable at least one of them will be awakened. Note that in some configurations there may not be any difference between this function and `cyg_drv_cond_broadcast()`.

cyg_drv_cond_broadcast

Function:

```c
void cyg_drv_cond_broadcast( cyg_drv_cond_t *cond )
```
Arguments:

cond - condition variable to broadcast to

Result:

None

Level:

DSR

Description:

Signal the condition variable pointed to by the cond argument. If there are any threads waiting on this variable they will all be awakened.

cyg_drv_interrupt_create

Function:

void cyg_drv_interrupt_create( cyg_vector_t vector,
                                cyg_priority_t priority,
                                cyg_addrword_t data,
                                cyg_ISR_t *isr,
                                cyg_DSR_t *dsr,
                                cyg_handle_t *handle,
                                cyg_interrupt *intr
                            )

Arguments:

vector - vector to attach to
priority - queuing priority
data - data pointer
isr - interrupt service routine
dsr - deferred service routine
handle - returned handle
intr - put interrupt object here

Result:

None

Level:

Thread

Description:

Create an interrupt object and returns a handle to it. The object contains information about which interrupt vector to use and the ISR and DSR that will be called after the interrupt object is attached to the vector. The interrupt object will be allocated in the memory passed in the intr parameter. The interrupt object is not immediately attached; it must be attached with the cyg_interrupt_attach() call.
cyg_drv_interrupt_delete

Function:

```c
void cyg_drv_interrupt_delete( cyg_handle_t interrupt );
```

Arguments:

- `interrupt` - interrupt to delete

Result:

None

Level:

Thread

Description:

Detach the interrupt from the vector and free the memory passed in the `intr` argument to `cyg_drv_interrupt_create()` for reuse.

cyg_drv_interrupt_attach

Function:

```c
void cyg_drv_interrupt_attach( cyg_handle_t interrupt );
```

Arguments:

- `interrupt` - interrupt to attach

Result:

None

Level:

ISR

Description:

Attach the interrupt to the vector so that interrupts will be delivered to the ISR when the interrupt occurs.

cyg_drv_interrupt_detach

Function:

```c
void cyg_drv_interrupt_detach( cyg_handle_t interrupt );
```

Arguments:

- `interrupt` - interrupt to detach

Result:

None
Chapter 13. Device Driver Interface to the Kernel

Level:

ISR

Description:

Detach the interrupt from the vector so that interrupts will no longer be delivered to the ISR.

cyg_drv_interrupt_mask

Function:

void cyg_drv_interrupt_mask(cyg_vector_t vector )

Arguments:

vector - vector to mask

Result:

None

Level:

ISR

Description:

Program the interrupt controller to stop delivery of interrupts on the given vector. On architectures which implement interrupt priority levels this may also disable all lower priority interrupts.

cyg_drv_interrupt_mask_intunsafe

Function:

void cyg_drv_interrupt_mask_intunsafe(cyg_vector_t vector )

Arguments:

vector - vector to mask

Result:

None

Level:

ISR

Description:

Program the interrupt controller to stop delivery of interrupts on the given vector. On architectures which implement interrupt priority levels this may also disable all lower priority interrupts. This version differs from cyg_drv_interrupt_mask() in not being interrupt safe. So in situations where, for example, interrupts are already known to be disabled, this may be called to avoid the extra overhead.
cyg_drv_interrupt_unmask

Function:

\[
\text{void cyg_drv_interrupt_unmask(cyg_vector_t vector })
\]

Arguments:

\[
\text{vector - vector to unmask}
\]

Result:

None

Level:

ISR

Description:

Program the interrupt controller to re-allow delivery of interrupts on the given \text{vector}.

cyg_drv_interrupt_unmask_intunsafe

Function:

\[
\text{void cyg_drv_interrupt_unmask_intunsafe(cyg_vector_t vector })
\]

Arguments:

\[
\text{vector - vector to unmask}
\]

Result:

None

Level:

ISR

Description:

Program the interrupt controller to re-allow delivery of interrupts on the given \text{vector}. This version differs from \text{cyg_drv_interrupt_unmask()} in not being interrupt safe.

cyg_drv_interrupt_acknowledge

Function:

\[
\text{void cyg_drv_interrupt_acknowledge( cyg_vector_t vector })
\]

Arguments:

\[
\text{vector - vector to acknowledge}
\]

Result:

None
Level:
ISR
Description:
Perform any processing required at the interrupt controller and in the CPU to cancel the current interrupt request on the vector. An ISR may also need to program the hardware of the device to prevent an immediate re-triggering of the interrupt.

cyg_drv_interrupt_configure

Function:

```c
void cyg_drv_interrupt_configure( cyg_vector_t vector,
                                 cyg_bool_t level,
                                 cyg_bool_t up
)
```

Arguments:

- `vector` - vector to configure
- `level` - level or edge triggered
- `up` - rising/falling edge, high/low level

Result:
None

Level:
ISR
Description:
Program the interrupt controller with the characteristics of the interrupt source. The `level` argument chooses between level- or edge-triggered interrupts. The `up` argument chooses between high and low level for level triggered interrupts or rising and falling edges for edge triggered interrupts. This function only works with interrupt controllers that can control these parameters.

cyg_drv_interrupt_level

Function:

```c
void cyg_drv_interrupt_level( cyg_vector_t vector,
                              cyg_priority_t level
)
```

Arguments:

- `vector` - vector to configure
- `level` - level to set

Result:
None
Level:
    ISR
Description:
    Program the interrupt controller to deliver the given interrupt at the supplied priority level. This function
only works with interrupt controllers that can control this parameter.

**cyg_drv_interrupt_set_cpu**

Function:
    
    void cyg_drv_interrupt_set_cpu( cyg_vector_t vector,
                                    cyg_cpu_t cpu
    )

Arguments:
    
    *vector* - interrupt vector to route
    *cpu* - destination CPU

Result:
    None

Level:
    ISR
Description:
    This function causes all interrupts on the given vector to be routed to the specified CPU. Subsequently,
all such interrupts will be handled by that CPU. This only works if the underlying hardware is capable of
performing this kind of routing. This function does nothing on a single CPU system.

**cyg_drv_interrupt_get_cpu**

Function:
    
    cyg_cpu_t cyg_drv_interrupt_set_cpu( cyg_vector_t vector )

Arguments:
    
    *vector* - interrupt vector to query

Result:
    The CPU to which this vector is routed

Level:
    ISR
Description:
    In multi-processor systems this function returns the id of the CPU to which interrupts on the given vector
are current being delivered. In single CPU systems this function returns zero.
Chapter 13. Device Driver Interface to the Kernel

**cyg_ISR_t**

Type:

```c
typedef cyg_uint32 cyg_ISR_t( cyg_vector_t vector,
   cyg_addrword_t data )
```

Fields:

- `vector` - vector being delivered
- `data` - data value supplied by client

Result:

Bit mask indicating whether interrupt was handled and whether the DSR should be called.

Description:

Interrupt Service Routine definition. A pointer to a function with this prototype is passed to `cyg_interrupt_create()` when an interrupt object is created. When an interrupt is delivered the function will be called with the vector number and the data value that was passed to `cyg_interrupt_create()`.

The return value is a bit mask containing one or both of the following bits:

- **CYG_ISR_HANDLED**
  
  indicates that the interrupt was handled by this ISR. It is a configuration option whether this will prevent further ISR being run.

- **CYG_ISR_CALL_DSR**
  
  causes the DSR that was passed to `cyg_interrupt_create()` to be scheduled to be called.

**cyg_DSR_t**

Type:

```c
typedef void cyg_DSR_t( cyg_vector_t vector,
   cyg_ucount32 count,
   cyg_addrword_t data )
```

Fields:

- `vector` - vector being delivered
- `count` - number of times DSR has been scheduled
- `data` - data value supplied by client

Result:

None
Description:

Deferred Service Routine prototype. A pointer to a function with this prototype is passed to `cyg_interrupt_create()` when an interrupt object is created. When the ISR requests the scheduling of its DSR, this function will be called at some later point. In addition to the `vector` and `data` arguments, which will be the same as those passed to the ISR, this routine is also passed a `count` of the number of times the ISR has requested that this DSR be scheduled. This counter is zeroed each time the DSR actually runs, so it indicates how many interrupts have occurred since it last ran.
V. File System Support Infrastructure
Chapter 14. Introduction

This document describes the filesystem infrastructure provided in eCos. This is implemented by the FILEIO package and provides POSIX compliant file and IO operations together with the BSD socket API. These APIs are described in the relevant standards and original documentation and will not be described here. See Chapter 35 for details of which parts of the POSIX standard are supported.

This document is concerned with the interfaces presented to client filesystems and network protocol stacks.

The FILEIO infrastructure consist mainly of a set of tables containing pointers to the primary interface functions of a file system. This approach avoids problems of namespace pollution (for example several filesystems can have a function called read(), so long as they are static). The system is also structured to eliminate the need for dynamic memory allocation.

New filesystems can be written directly to the interfaces described here. Existing filesystems can be ported very easily by the introduction of a thin veneer porting layer that translates FILEIO calls into native filesystem calls.

The term filesystem should be read fairly loosely in this document. Object accessed through these interfaces could equally be network protocol sockets, device drivers, fifos, message queues or any other object that can present a file-like interface.
Chapter 15. File System Table

The filesystem table is an array of entries that describe each filesystem implementation that is part of the system image. Each resident filesystem should export an entry to this table using the FSTAB_ENTRY() macro.

Note: At present we do not support dynamic addition or removal of table entries. However, an API similar to mount() would allow new entries to be added to the table.

The table entries are described by the following structure:

```c
struct cyg_fstab_entry
{
    const char *name; // filesystem name
    CYG_ADDRWORD data; // private data value
    cyg_uint32 syncmode; // synchronization mode
    int (*mount) ( cyg_fstab_entry *fste, cyg_mtab_entry *mte );
    int (*umount) ( cyg_mtab_entry *mte );
    int (*open) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  int mode, cyg_file *fte );
    int (*unlink) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
    int (*mkdir) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
    int (*rmdir) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
    int (*rename) ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                    cyg_dir dir2, const char *name2 );
    int (*link) ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                 cyg_dir dir2, const char *name2, int type );
    int (*opendir) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                    cyg_file *fte );
    int (*chdir) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  cyg_dir *dir_out );
    int (*stat) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                 struct stat *buf );
    int (*getinfo) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                     int key, char *buf, int len );
    int (*setinfo) ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                     int key, char *buf, int len );
};
```

The `name` field points to a string that identifies this filesystem implementation. Typical values might be "romfs", "msdos", "ext2" etc.

The `data` field contains any private data that the filesystem needs, perhaps the root of its data structures.

The `syncmode` field contains a description of the locking protocol to be used when accessing this filesystem. It will be described in more detail in Chapter 19.

The remaining fields are pointers to functions that implement filesystem operations that apply to files and directories as whole objects. The operation implemented by each function should be obvious from the names, with a few exceptions:

The `opendir()` function pointer opens a directory for reading. See Chapter 18 for details.

The `getinfo()` and `setinfo()` function pointers provide support for various minor control and information functions such as `pathconf()` and `access()`.
Chapter 15. File System Table

With the exception of the `mount()` and `umount()` functions, all of these functions take three standard arguments, a pointer to a mount table entry (see later) a directory pointer (also see later) and a file name relative to the directory. These should be used by the filesystem to locate the object of interest.
Chapter 16. Mount Table

The mount table records the filesystems that are actually active. These can be seen as being analogous to mount points in Unix systems.

There are two sources of mount table entries. Filesystems (or other components) may export static entries to the table using the `MTAB_ENTRY()` macro. Alternatively, new entries may be installed at run time using the `mount()` function. Both types of entry may be unmounted with the `umount()` function.

A mount table entry has the following structure:

```c
c struct cyg_mtab_entry
{
    const char *name; // name of mount point
    const char *fsname; // name of implementing filesystem
    const char *devname; // name of hardware device
    CYG_ADDRWORD data; // private data value
    cyg_bool valid; // Valid entry?
    cyg fstab_entry *fs; // pointer to fstab entry
    cyg_dir root; // root directory pointer
};
```

The `name` field identifies the mount point. This is used to direct rooted filenames (filenames that begin with "/") to the correct filesystem. When a file name that begins with "/" is submitted, it is matched against the `name` fields of all valid mount table entries. The entry that yields the longest match terminating before a "/", or end of string, wins and the appropriate function from the filesystem table entry is then passed the remainder of the file name together with a pointer to the table entry and the value of the `root` field as the directory pointer.

For example, consider a mount table that contains the following entries:

```c
{ 
    "/", "msdos", "/dev/hd0", ... 
}
{ 
    "/fd", "msdos", "/dev/fd0", ... 
}
{ 
    "/rom", "romfs", ",", ... 
}
{ 
    "/tmp", "ramfs", ",", ... 
}
{ 
    "/dev", "devfs", ",", ... 
}
```

An attempt to open "/tmp/foo" would be directed to the RAM filesystem while an open of "/bar/bundy" would be directed to the hard disc MSDOS filesystem. Opening "/dev/tty0" would be directed to the device management filesystem for lookup in the device table.

Unrooted file names (those that do not begin with a "/") are passed straight to the filesystem that contains the current directory. The current directory is represented by a pair consisting of a mount table entry and a directory pointer.

The `fsname` field points to a string that should match the `name` field of the implementing filesystem. During initialization the mount table is scanned and the `fsname` entries looked up in the filesystem table. For each match, the filesystem’s `_mount_` function is called and if successful the mount table entry is marked as valid and the `fs` pointer installed.

The `devname` field contains the name of the device that this filesystem is to use. This may match an entry in the device table (see later) or may be a string that is specific to the filesystem if it has its own internal device drivers.

The `data` field is a private data value. This may be installed either statically when the table entry is defined, or may be installed during the `mount()` operation.
The valid field indicates whether this mount point has actually been mounted successfully. Entries with a false valid field are ignored when searching for a name match.

The fs field is installed after a successful mount() operation to point to the implementing filesystem.

The root field contains a directory pointer value that the filesystem can interpret as the root of its directory tree. This is passed as the dir argument of filesystem functions that operate on rooted filenames. This field must be initialized by the filesystem’s mount() function.
Chapter 17. File Table

Once a file has been opened it is represented by an open file object. These are allocated from an array of available file objects. User code accesses these open file objects via a second array of pointers which is indexed by small integer offsets. This gives the usual Unix file descriptor functionality, complete with the various duplication mechanisms.

A file table entry has the following structure:

```c
struct CYG_FILE_TAG
{
    cyg_uint32 f_flag;  /* file state */
    cyg_uint16 f_ucount;  /* use count */
    cyg_uint16 f_type;  /* descriptor type */
    cyg_uint32 f_syncmode;  /* synchronization protocol */
    struct CYG_FILEOPS_TAG *f_ops;  /* file operations */
    off_t f_offset;  /* current offset */
    CYG_ADDRWORD f_data;  /* file or socket */
    CYG_ADDRWORD f_xops;  /* extra type specific ops */
    cyg_mtab_entry *f_mte;  /* mount table entry */
};
```

The `f_flag` field contains some FILEIO control bits and some bits propagated from the `flags` argument of the `open()` call (defined by `CYG_FILE_MODE_MASK`).

The `f_ucount` field contains a use count that controls when a file will be closed. Each duplicate in the file descriptor array counts for one reference here. It is also incremented around each I/O operation to ensure that the file cannot be closed while it has current I/O operations.

The `f_type` field indicates the type of the underlying file object. Some of the possible values here are `CYG_FILE_TYPE_FILE`, `CYG_FILE_TYPE_SOCKET` or `CYG_FILE_TYPE_DEVICE`.

The `f_syncmode` field is copied from the `syncmode` field of the implementing filesystem. Its use is described in Chapter 19.

The `f_offset` field records the current file position. It is the responsibility of the file operation functions to keep this field up to date.

The `f_data` field contains private data placed here by the underlying filesystem. Normally this will be a pointer to, or handle on, the filesystem object that implements this file.

The `f_xops` field contains a pointer to any extra type specific operation functions. For example, the socket I/O system installs a pointer to a table of functions that implement the standard socket operations.

The `f_mte` field contains a pointer to the parent mount table entry for this file. It is used mainly to implement the synchronization protocol. This may contain a pointer to some other data structure in file objects not derived from a filesystem.

The `f_ops` field contains a pointer to a table of file I/O operations. This has the following structure:

```c
struct CYG_FILEOPS_TAG
{
    int (*fo_read) (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int (*fo_write) (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int (*fo_lseek) (struct CYG_FILE_TAG *fp, off_t *pos, int whence);
    int (*fo_ioctl) (struct CYG_FILE_TAG *fp, CYG_ADDRWORD com,
                     CYG_ADDRWORD data);
};
```
Chapter 17. File Table

int (*fo_select) (struct CYG_FILE_TAG *fp, int which, CYG_ADDRWORD info);
int (*fo_fsync) (struct CYG_FILE_TAG *fp, int mode);
int (*fo_close) (struct CYG_FILE_TAG *fp);
int (*fo_fstat) (struct CYG_FILE_TAG *fp, struct stat *buf);
int (*fo_getinfo) (struct CYG_FILE_TAG *fp, int key, char *buf, int len);
int (*fo_setinfo) (struct CYG_FILE_TAG *fp, int key, char *buf, int len);

It should be obvious from the names of most of these functions what their responsibilities are. The fo_getinfo() and fo_setinfo() function pointers, like their counterparts in the filesystem structure, implement minor control and info functions such as fpathconf().

The second argument to the fo_read() and fo_write() function pointers is a pointer to a UIO structure:

struct CYG_UIO_TAG
{
    struct CYG_IOVEC_TAG *uio_iov; /* pointer to array of iovecs */
    int uio_iovcnt; /* number of iovecs in array */
    off_t uio_offset; /* offset into file this uio corresponds to */
    ssize_t uio_resid; /* residual i/o count */
    enum cyg_uio_seg uio_segflg; /* see above */
    enum cyg_uio_rw uio_rw; /* see above */
};

struct CYG_IOVEC_TAG
{
    void *iov_base; /* Base address. */
    ssize_t iov_len; /* Length. */
};

This structure encapsulates the parameters of any data transfer operation. It provides support for scatter/gather operations and records the progress of any data transfer. It is also compatible with the I/O operations of any BSD-derived network stacks and filesystems.

When a file is opened (or a file object created by some other means, such as socket() or accept()) it is the responsibility of the filesystem open operation to initialize all the fields of the object except the f_ucount, f_syncmode and f_mte fields. Since the f_flag field will already contain bits belonging to the FILEIO infrastructure, any changes to it must be made with the appropriate logical operations.
Chapter 18. Directories

Filesystem operations all take a directory pointer as one of their arguments. A directory pointer is an opaque handle managed by the filesystem. It should encapsulate a reference to a specific directory within the filesystem. For example, it may be a pointer to the data structure that represents that directory (such as an inode), or a pointer to a pathname for the directory.

The `chdir()` filesystem function pointer has two modes of use. When passed a pointer in the `dir_out` argument, it should locate the named directory and place a directory pointer there. If the `dir_out` argument is NULL then the `dir` argument is a previously generated directory pointer that can now be disposed of. When the infrastructure is implementing the `chdir()` function it makes two calls to filesystem `chdir()` functions. The first is to get a directory pointer for the new current directory. If this succeeds the second is to dispose of the old current directory pointer.

The `opendir()` function is used to open a directory for reading. This results in an open file object that can be read to return a sequence of struct dirent objects. The only operations that are allowed on this file are `read`, `lseek` and `close`. Each read operation on this file should return a single struct dirent object. When the end of the directory is reached, zero should be returned. The only seek operation allowed is a rewind to the start of the directory, by supplying an offset of zero and a `whence` specifier of `SEEK_SET`.

Most of these considerations are invisible to clients of a filesystem since they will access directories via the POSIX `opendir()`, readdir() and closedir() functions. The `struct dirent` object returned by readdir() will always contain `d_name` as required by POSIX. When `CYGPKG_FILEIO_DIRENT_DTYPE` is enabled it will also contain `d_type`, which is not part of POSIX, but often implemented by OSes. Currently only the FATFS, RAMFS, ROMFS and JFFS2 filesystem sets this value. For other filesystems a value of 0 will be returned in the member.

Support for the `getcwd()` function is provided by three mechanisms. The first is to use the `FS_INFO_GETCWD` getinfo key on the filesystem to use any internal support that it has for this. If that fails it falls back on one of the two other mechanisms. If `CYGPKG_IO_FILEIO_TRACK_CWD` is set then the current directory is tracked textually in `chdir()` and the result of that is reported in `getcwd()`. Otherwise an attempt is made to traverse the directory tree to its root using ".." entries.

This last option is complicated and expensive, and relies on the filesystem supporting "." and ".." entries. This is not always the case, particularly if the filesystem has been ported from a non-UNIX-compatible source. Tracking the pathname textually will usually work, but might not produce optimum results when symbolic links are being used.
Chapter 19. Synchronization

The FILEIO infrastructure provides a synchronization mechanism for controlling concurrent access to filesystems. This allows existing filesystems to be ported to eCos, even if they do not have their own synchronization mechanisms. It also allows new filesystems to be implemented easily without having to consider the synchronization issues.

The infrastructure maintains a mutex for each entry in each of the main tables: filesystem table, mount table and file table. For each class of operation each of these mutexes may be locked before the corresponding filesystem operation is invoked.

The synchronization protocol required by a filesystem is described by the syncmode field of the filesystem table entry. This is a combination of the following flags:

- CYG_SYNCMODE_FILE_FILESYSTEM
  Lock the filesystem table entry mutex during all filesystem level operations.

- CYG_SYNCMODE_FILE_MOUNTPOINT
  Lock the mount table entry mutex during all filesystem level operations.

- CYG_SYNCMODE_IO_FILE
  Lock the file table entry mutex during all I/O operations.

- CYG_SYNCMODE_IO_FILESYSTEM
  Lock the filesystem table entry mutex during all I/O operations.

- CYG_SYNCMODE_IO_MOUNTPOINT
  Lock the mount table entry mutex during all I/O operations.

- CYG_SYNCMODE_SOCK_FILE
  Lock the file table entry mutex during all socket operations.

- CYG_SYNCMODE_SOCK_NETSTACK
  Lock the network stack table entry mutex during all socket operations.

- CYG_SYNCMODE_NONE
  Perform no locking at all during any operations.

The value of the syncmode field in the filesystem table entry will be copied by the infrastructure to the open file object after a successful open() operation.
Chapter 20. Initialization and Mounting

As mentioned previously, mount table entries can be sourced from two places. Static entries may be defined by using the \texttt{MTAB\_ENTRY()} macro. Such entries will be automatically mounted on system startup. For each entry in the mount table that has a non-null \texttt{name} field the filesystem table is searched for a match with the \texttt{fsname} field. If a match is found the filesystem’s \texttt{mount} entry is called and if successful the mount table entry marked valid and the \texttt{fs} field initialized. The \texttt{mount()} function is responsible for initializing the \texttt{root} field.

The size of the mount table is defined by the configuration value \texttt{CYGNUS\_FILEIO\_MTAB\_MAX}. Any entries that have not been statically defined are available for use by dynamic mounts.

A filesystem may be mounted dynamically by calling \texttt{mount()}. This function has the following prototype:

\begin{verbatim}
int mount( const char *devname,
           const char *dir,
           const char *fsname);
\end{verbatim}

The \texttt{devname} argument identifies a device that will be used by this filesystem and will be assigned to the \texttt{devname} field of the mount table entry.

The \texttt{dir} argument is the mount point name, it will be assigned to the \texttt{name} field of the mount table entry.

The \texttt{fsname} argument is the name of the implementing filesystem, it will be assigned to the \texttt{fsname} entry of the mount table entry.

The process of mounting a filesystem dynamically is as follows. First a search is made of the mount table for an entry with a NULL \texttt{name} field to be used for the new mount point. The filesystem table is then searched for an entry whose name matches \texttt{fsname}. If this is successful then the mount table entry is initialized and the filesystem’s \texttt{mount()} operation called. If this is successful, the mount table entry is marked valid and the \texttt{fs} field initialized.

Unmounting a filesystem is done by the \texttt{umount()} function. This can unmount filesystems whether they were mounted statically or dynamically.

The \texttt{umount()} function has the following prototype:

\begin{verbatim}
int umount( const char *name );
\end{verbatim}

The mount table is searched for a match between the \texttt{name} argument and the entry \texttt{name} field. When a match is found the filesystem’s \texttt{umount()} operation is called and if successful, the mount table entry is invalidated by setting its \texttt{valid} field false and the \texttt{name} field to NULL.
Chapter 21. Sockets

If a network stack is present, then the FILEIO infrastructure also provides access to the standard BSD socket calls.

The netstack table contains entries which describe the network protocol stacks that are in the system image. Each resident stack should export an entry to this table using the NSTAB_ENTRY() macro.

Each table entry has the following structure:

```c
struct cyg_nstab_entry
{
    cyg_bool valid;  // true if stack initialized
    cyg_uint32 syncmode;  // synchronization protocol
    char *name;  // stack name
    char *devname;  // hardware device name
    CYG_ADDRWORD data;  // private data value

    int (*init)( cyg_nstab_entry *nste );
    int (*socket)( cyg_nstab_entry *nste, int domain, int type,
                   int protocol, cyg_file *file );
};
```

This table is analogous to a combination of the filesystem and mount tables.

The valid field is set true if the stack’s init() function returned successfully and the syncmode field contains the CYG_SYNCMODE_SOCK_* bits described above.

The name field contains the name of the protocol stack.

The devname field names the device that the stack is using. This may reference a device under "/dev", or may be a name that is only meaningful to the stack itself.

The init() function pointer is called during system initialization to start the protocol stack running. If it returns non-zero the valid field is set false and the stack will be ignored subsequently.

The socket() function is called to attempt to create a socket in the stack. When the socket() API function is called the netstack table is scanned and for each valid entry the socket() function pointer is called. If this returns non-zero then the scan continues to the next valid stack, or terminates with an error if the end of the table is reached.

The result of a successful socket call is an initialized file object with the f_xops field pointing to the following structure:

```c
struct cyg_sock_ops
{
    int (*bind) ( cyg_file *fp, const sockaddr *sa, socklen_t len );
    int (*connect) ( cyg_file *fp, const sockaddr *sa, socklen_t len );
    int (*accept) ( cyg_file *fp, cyg_file *new_fp,
                    struct sockaddr *name, socklen_t *anamelen );
    int (*listen) ( cyg_file *fp, int len );
    int (*getname) ( cyg_file *fp, sockaddr *sa, socklen_t *len, int peer );
    int (*shutdown) ( cyg_file *fp, int flags );
    int (*getsockopt)( cyg_file *fp, int level, int optname,
                       void *optval, socklen_t *optlen);  
    int (*setsockopt)( cyg_file *fp, int level, int optname,
                       const void *optval, socklen_t optlen);  
};
```
Chapter 21. Sockets

```c
int (*sendmsg) ( cyg_file *fp, const struct msghdr *m,
                  int flags, ssize_t *retsize );
int (*recvmsg) ( cyg_file *fp, struct msghdr *m,
                  socklen_t *namelen, ssize_t *retsize );
```

It should be obvious from the names of these functions which API calls they provide support for. The `getname()` function pointer provides support for both `getsockname()` and `getpeername()` while the `sendmsg()` and `recvmsg()` function pointers provide support for `send()`, `sendto()`, `sendmsg()`, `recv()`, `recvfrom()` and `recvmsg()` as appropriate.
Chapter 22. Select

The infrastructure provides support for implementing a select mechanism. This is modeled on the mechanism in the BSD kernel, but has been modified to make it implementation independent.

The main part of the mechanism is the select() API call. This processes its arguments and calls the fo_select() function pointer on all file objects referenced by the file descriptor sets passed to it. If the same descriptor appears in more than one descriptor set, the fo_select() function will be called separately for each appearance.

The which argument of the fo_select() function will either be CYG_FREAD to test for read conditions, CYG_FWRITE to test for write conditions or zero to test for exceptions. For each of these options the function should test whether the condition is satisfied and if so return true. If it is not satisfied then it should call cyg_selrecord() with the info argument that was passed to the function and a pointer to a cyg_selinfo structure.

The cyg_selinfo structure is used to record information about current select operations. Any object that needs to support select must contain an instance of this structure. Separate cyg_selinfo structures should be kept for each of the options that the object can select on - read, write or exception.

If none of the file objects report that the select condition is satisfied, then the select() API function puts the calling thread to sleep waiting either for a condition to become satisfied, or for the optional timeout to expire. A selectable object must have some asynchronous activity that may cause a select condition to become true - either via interrupts or the activities of other threads. Whenever a selectable condition is satisfied, the object should call cyg_selwakeup() with a pointer to the appropriate cyg_selinfo structure. If the thread is still waiting, this will cause it to wake up and repeat its poll of the file descriptors. This time around, the object that caused the wakeup should indicate that the select condition is satisfied, and the select() API call will return.

Note that select() does not exhibit real time behaviour: the iterative poll of the descriptors, and the wakeup mechanism mitigate against this. If real time response to device or socket I/O is required then separate threads should be devoted to each device of interest and should use blocking calls to wait for a condition to become ready.
Chapter 22. Select
Chapter 23. Devices

Devices are accessed by means of a pseudo-filesystem, "devfs", that is mounted on "/dev". Open operations are translated into calls to cyg_io_lookup() and if successful result in a file object whose f_ops functions translate filesystem API functions into calls into the device API.
Chapter 24. Writing a New Filesystem

To create a new filesystem it is necessary to define the fstab entry and the file IO operations. The easiest way to do this is to copy an existing filesystem: either the test filesystem in the FILEIO package, or the RAM or ROM filesystem packages.

To make this clearer, the following is a brief tour of the FILEIO relevant parts of the RAM filesystem.

First, it is necessary to provide forward definitions of the functions that constitute the filesystem interface:

---
// Forward definitions

// Filesystem operations
static int ramfs_mount ( cyg_fstab_entry *fste, cyg_mtab_entry *mte );
static int ramfs_unmount ( cyg_mtab_entry *mte );
static int ramfs_open ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                      int mode, cyg_file *fte );
static int ramfs_unlink ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_mkdir ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_rmdir ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_rename ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                       cyg_dir dir2, const char *name2 );
static int ramfs_link ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                      cyg_dir dir2, const char *name2, int type );
static int ramfs_opendir ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                         cyg_file *fte );
static int ramfs_chdir ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                     cyg_dir *dir_out );
static int ramfs_stat ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                      struct stat *buf);}
static int ramfs_getinfo ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                       int key, void *buf, int len );
static int ramfs_setinfo ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                       int key, void *buf, int len );

// File operations
static int ramfs_fo_read (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_write (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_lseek (struct CYG_FILE_TAG *fp, off_t *pos, int whence );
static int ramfs_fo_ioctl (struct CYG_FILE_TAG *fp, CYG_ADDRWORD com,
                         CYG_ADDRWORD data);
static int ramfs_fo_fsync (struct CYG_FILE_TAG *fp, int mode );
static int ramfs_fo_close (struct CYG_FILE_TAG *fp);
static int ramfs_fo_fstat (struct CYG_FILE_TAG *fp, struct stat *buf );
static int ramfs_fo_getinfo (struct CYG_FILE_TAG *fp, int key, void *buf, int len );
static int ramfs_fo_setinfo (struct CYG_FILE_TAG *fp, int key, void *buf, int len );

// Directory operations
static int ramfs_fo_dirread (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_dirlseek (struct CYG_FILE_TAG *fp, off_t *pos, int whence );

We define all of the fstab entries and all of the file IO operations. We also define alternatives for the fo_read and fo_lseek file IO operations.

We can now define the filesystem table entry. There is a macro, FSTAB_ENTRY to do this:
Chapter 24. Writing a New Filesystem

//==========================================================================
// Fstab entry.
// This defines the entry in the filesystem table.
// For simplicity we use _FILESYSTEM synchronization for all accesses since
// we should never block in any filesystem operations.

FSTAB_ENTRY( ramfs_fste, "ramfs", 0,
      CYG_SYNCMODE_FILE_FILESYSTEM|CYG_SYNCMODE_IO_FILESYSTEM,
      ramfs_mount,
      ramfs_umount,
      ramfs_open,
      ramfs_unlink,
      ramfs_mkdir,
      ramfs_rmdir,
      ramfs_rename,
      ramfs_link,
      ramfs_opendir,
      ramfs_chdir,
      ramfs_stat,
      ramfs_getinfo,
      ramfs_setinfo);

The first argument to this macro gives the fstab entry a name, the remainder are initializers for the field of the
structure.

We must also define the file operations table that is installed in all open file table entries:

static cyg_fileops ramfs_fileops =
{
      ramfs_fo_read,
      ramfs_fo_write,
      ramfs_fo_lseek,
      ramfs_fo_ioctl,
      cyg_fileio_seltrue,
      ramfs_fo_fsync,
      ramfs_fo_close,
      ramfs_fo_fstat,
      ramfs_fo_getinfo,
      ramfs_fo_setinfo
};

These all point to functions supplied by the filesystem except the fo_select field which is filled with a pointer
to cyg_fileio_seltrue(). This is provided by the FILEIO package and is a select function that always
returns true to all operations.

Finally, we need to define a set of file operations for use when reading directories. This table only defines the
fo_read and fo_lseek operations. The rest are filled with stub functions supplied by the FILEIO package
that just return an error code.

static cyg_fileops ramfs_fileops_dir =
{
      ramfs_fo_read,
      ramfs_fo_write,
      ramfs_fo_lseek,
      ramfs_fo_ioctl,
      cyg_fileio_seltrue,
      ramfs_fo_fsync,
      ramfs_fo_close,
      ramfs_fo_fstat,
      ramfs_fo_getinfo,
      ramfs_fo_setinfo
};
// point to error-returning stub functions. Only the read, lseek and // close entries are functional.

static cyg_fileops ramfs_dirops =
{
    ramfs_fo_dirread,
    (cyg_fileop_write *)cyg_fileio_enosys,
    ramfs_fo_dirlseek,
    (cyg_fileop_ioctl *)cyg_fileio_enosys,
    cyg_fileio_seltrue,
    (cyg_fileop_fsync *)cyg_fileio_enosys,
    ramfs_fo_close,
    (cyg_fileop_fstat *)cyg_fileio_enosys,
    (cyg_fileop_getinfo *)cyg_fileio_enosys,
    (cyg_fileop_setinfo *)cyg_fileio_enosys
};

If the filesystem wants to have an instance automatically mounted on system startup, it must also define a mount table entry. This is done with the MTAB_ENTRY macro. This is an example from the test filesystem of how this is used:

MTAB_ENTRY( testfs_mte1,
    "/",
    "testfs",
    ",",
    0);

The first argument provides a name for the table entry. The following arguments provide initialization for the name, fsname, devname and data fields respectively.

These definitions are adequate to let the new filesystem interact with the FILEIO package. The new filesystem now needs to be fleshed out with implementations of the functions defined above. Obviously, the exact form this takes will depend on what the filesystem is intended to do. Take a look at the RAM and ROM filesystems for examples of how this has been done.
VI. PCI Library
Chapter 25. The eCos PCI Library

The eCos PCI library is an optional part of eCos, and is only applicable to some platforms.

PCI Library

The eCos PCI library provides the following functionality:

1. Scan the PCI bus for specific devices or devices of a certain class.
2. Read and change generic PCI information.
3. Read and change device-specific PCI information.
4. Allocate PCI memory and IO space to devices.
5. Translate a device’s PCI interrupts to equivalent HAL vectors.

Example code fragments are from the pci1 test (see io/pci/<release>/tests/pci1.c).

All of the functions described below are declared in the header file <cyg/io/pci.h> which all clients of the PCI library should include.

PCI Overview

The PCI bus supports several address spaces: memory, IO, and configuration. All PCI devices must support mandatory configuration space registers. Some devices may also present IO mapped and/or memory mapped resources. Before devices on the bus can be used, they must be configured. Basically, configuration will assign PCI IO and/or memory address ranges to each device and then enable that device. All PCI devices have a unique address in configuration space. This address is comprised of a bus number, a device number, and a function number. Special devices called bridges are used to connect two PCI busses together. The PCI standard supports up to 255 busses with each bus having up to 32 devices and each device having up to 8 functions.

The environment in which a platform operates will dictate if and how eCos should configure devices on the PCI bus. If the platform acts as a host on a single PCI bus, then devices may be configured individually from the relevant device driver. If the platform is not the primary host, such as a PCI card plugged into a PC, configuration of PCI devices may be left to the PC BIOS. If PCI-PCI bridges are involved, configuration of all devices is best done all at once early in the boot process. This is because all devices on the secondary side of a bridge must be evaluated for their IO and memory space requirements before the bridge can be configured.

Initializing the bus

The PCI bus needs to be initialized before it can be used. This only needs to be done once - some HALs may do it as part of the platform initialization procedure, other HALs may leave it to the application or device drivers to do it. The following function will do the initialization only once, so it’s safe to call from multiple drivers:

```c
void cyg_pci_init( void );
```

Scanning for devices

After the bus has been initialized, it is possible to scan it for devices. This is done using the function:
Chapter 25. The eCos PCI Library

cyg_bool cyg_pci_find_next( cyg_pci_device_id cur_devid,
       cyg_pci_device_id *next_devid );

It will scan the bus for devices starting at cur_devid. If a device is found, its devid is stored in next_devid
and the function returns true.

The pci1 test's outer loop looks like:

```c
   cyg_pci_init();
   if (cyg_pci_find_next(CYG_PCI_NULL_DEVID, &devid)) {
       do {
           <use devid>
       } while (cyg_pci_find_next(devid, &devid));
   }
```

What happens is that the bus gets initialized and a scan is started. CYG_PCI_NULL_DEVID causes
`cyg_pci_find_next()` to restart its scan. If the bus does not contain any devices, the first call to
`cyg_pci_find_next()` will return false.

If the call returns true, a loop is entered where the found devid is used. After devid processing has completed,
the next device on the bus is searched for; `cyg_pci_find_next()` continues its scan from the current devid.
The loop terminates when no more devices are found on the bus.

This is the generic way of scanning the bus, enumerating all the devices on the bus. But if the application is
looking for a device of a given device class (e.g., a SCSI controller), or a specific vendor device, these functions
simplify the task a bit:

```c
   cyg_bool cyg_pci_find_class( cyg_uint32 dev_class,
                                  cyg_pci_device_id *devid );
   cyg_bool cyg_pci_find_device( cyg_uint16 vendor, cyg_uint16 device,
                                  cyg_pci_device_id *devid );
```

They work just like `cyg_pci_find_next()`, but only return true when the dev_class or vendor/device qualifiers
match those of a device on the bus. The devid serves as both an input and an output operand: the scan starts at
the given device, and if a device is found devid is updated with the value for the found device.

The `<cyg/io/pci_cfg.h>` header file (included by `pci.h`) contains definitions for PCI class, vendor and
device codes which can be used as arguments to the find functions. The list of vendor and device codes
is not complete: add new codes as necessary. If possible also register the codes at the PCI Code List
(http://www.yourvote.com/pci) which is where the eCos definitions are
generated from.

Generic config information

When a valid device ID (devid) is found using one of the above functions, the associated device can be queried
and controlled using the functions:

```c
   void cyg_pci_get_device_info ( cyg_pci_device_id devid,
                                  cyg_pci_device *dev_info );
   void cyg_pci_set_device_info ( cyg_pci_device_id devid,
                                  cyg_pci_device *dev_info );
```

The `cyg_pci_device` structure (defined in `pci.h`) primarily holds information as described by the PCI specification[1]. The pci1 test prints out some of this information:

```c
   // Get device info
   cyg_pci_get_device_info(devid, &dev_info);
   diag_printf("\n Command 0x%04x, Status 0x%04x\n",
```
dev_info.command, dev_info.status);

The command register can also be written to, controlling (among other things) whether the device responds to
IO and memory access from the bus.

**Specific config information**

The above functions only allow access to generic PCI config registers. A device can have extra config registers
not specified by the PCI specification. These can be accessed with these functions:

```c
void cyg_pci_read_config_uint8( cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint8 *val);
void cyg_pci_read_config_uint16( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint16 *val);
void cyg_pci_read_config_uint32( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint32 *val);
void cyg_pci_write_config_uint8( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint8 val);
void cyg_pci_write_config_uint16( cyg_pci_device_id devid,
                                   cyg_uint8 offset, cyg_uint16 val);
void cyg_pci_write_config_uint32( cyg_pci_device_id devid,
                                   cyg_uint8 offset, cyg_uint32 val);
```

The write functions should only be used for device-specific config registers since using them on generic regis-
ters may invalidate the contents of a previously fetched cyg_pci_device structure.

**Allocating memory**

A PCI device ignores all IO and memory access from the PCI bus until it has been activated. Activation cannot
happen until after device configuration. Configuration means telling the device where it should map its IO and
memory resources. This is done with one of the following functions:

```c
cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info );
cyg_bool cyg_pci_configure_bus( cyg_uint8 bus, cyg_uint8 *next_bus );
```

The `cyg_pci_configure_device` handles all IO and memory regions that need configuration on non-bridge
devices. On platforms with multiple busses connected by bridges, the `cyg_pci_configure_bus` function
should be used. It will recursively configure all devices on the given `bus` and all subordinate busses.
`cyg_pci_configure_bus` will use `cyg_pci_configure_device` to configure individual non-bridge devices.

Each region is represented in the PCI device’s config space by BARs (Base Address Registers) and is handled
individually according to type using these functions:

```c
cyg_bool cyg_pci_allocate_memory( cyg_pci_device *dev_info,
                                   cyg_uint32 bar, 
                                   CYG_PCI_ADDRESS64 *base );
cyg_bool cyg_pci_allocate_io( cyg_pci_device *dev_info,
                               cyg_uint32 bar, 
                               CYG_PCI_ADDRESS32 *base );
```

The memory bases (in two distinct address spaces) are increased as memory regions are allocated to devices.
Allocation will fail (the function returns false) if the base exceeds the limits of the address space (IO is 1MB,
memory is 2^32 or 2^64 bytes).

These functions can also be called directly by the application/driver if necessary, but this should not be neces-
sary.
The bases are initialized with default values provided by the HAL. It is possible for an application to override these using the following functions:

```c
void cyg_pci_set_memory_base( CYG_PCI_ADDRESS64 base );
void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
```

When a device has been configured, the `cyg_pci_device` structure will contain the physical address in the CPU’s address space where the device’s memory regions can be accessed.

This information is provided in `base_map[]` - there is a 32 bit word for each of the device’s BARs. For 32 bit PCI memory regions, each 32 bit word will be an actual pointer that can be used immediately by the driver: the memory space will normally be linearly addressable by the CPU.

However, for 64 bit PCI memory regions, some (or all) of the region may be outside of the CPUs address space. In this case the driver will need to know how to access the region in segments. This functionality may be adopted by the eCos HAL if deemed useful in the future. The 2GB available on many systems should suffice though.

**Interrupts**

A device may generate interrupts. The HAL vector associated with a given device on the bus is platform specific. This function allows a driver to find the actual interrupt vector for a given device:

```c
bool cyg_pci_translate_interrupt( cyg_pci_device *dev_info,
                                 CYG_ADDRWORD *vec );
```

If the function returns false, no interrupts will be generated by the device. If it returns true, the `CYG_ADDRWORD` pointed to by `vec` is updated with the HAL interrupt vector the device will be using. This is how the function is used in the `pci1` test:

```c
  if (cyg_pci_translate_interrupt(&dev_info, &irq))
    diag_printf(" Wired to HAL vector %d\n", irq);
  else
    diag_printf(" Does not generate interrupts.\n");
```

The application/drive should attach an interrupt handler to a device’s interrupt before activating the device.

**Activating a device**

When the device has been allocated memory space it can be activated. This is not done by the library since a driver may have to initialize more state on the device before it can be safely activated.

Activating the device is done by enabling flags in its command word. As an example, see the `pci1` test which can be configured to enable the devices it finds. This allows these to be accessed from GDB (if a breakpoint is set on `cyg_test_exit`):

```c
#ifdef ENABLE_PCI_DEVICES
{
  cyg_uint16 cmd;
  // Don’t use cyg_pci_set_device_info since it clears
  // some of the fields we want to print out below.
  cyg_pci_read_config_uint16(dev_info.devid,
                             CYG_PCI_CFG_COMMAND, &cmd);
  cmd |= CYG_PCI_CFG_COMMAND_IO|CYG_PCI_CFG_COMMAND_MEMORY;
  cyg_pci_write_config_uint16(dev_info.devid,
```
Chapter 25. The eCos PCI Library

The best way to activate a device is actually through `cyg_pci_set_device_info()`, but in this particular case the `cyg_pci_device` structure contents from before the activation is required for printout further down in the code.

**Links**

See these links for more information about PCI:

1. http://www.pcisig.com/ - information on the PCI specifications
2. http://www.yourvote.com/pci/ - list of vendor and device IDs

**PCI Library reference**

This document defines the PCI Support Library for eCos.

The PCI support library provides a set of routines for accessing the PCI bus configuration space in a portable manner. This is provided by two APIs. The high level API is used by device drivers, or other code, to access the PCI configuration space portably. The low level API is used by the PCI library itself to access the hardware in a platform-specific manner, and may also be used by device drivers to access the PCI configuration space directly.

Underlying the low-level API is HAL support for the basic configuration space operations. These should not generally be used by any code other than the PCI library, and are present in the HAL to allow low level initialization of the PCI bus and devices to take place if necessary.

**PCI Library API**

The PCI library provides the following routines and types for accessing the PCI configuration space.

The API for the PCI library is found in the header file `<cyg/io/pci.h>`.

**Definitions**

The header file contains definitions for the common configuration structure offsets and specimen values for device, vendor and class code.
Types and data structures

The following types are defined:

typedef CYG_WORD32 cyg_pci_device_id;

This is comprised of the bus number, device number and functional unit numbers packed into a single word. The macro \texttt{CYG_PCI_DEV_MAKE_ID()}, in conjunction with the \texttt{CYG_PCI_DEV_MAKE_DEVFN()} macro, may be used to construct a device id from the bus, device and functional unit numbers. Similarly the macros \texttt{CYG_PCI_DEV_GET_BUS()}, \texttt{CYG_PCI_DEV_GET_DEVFN()}, \texttt{CYG_PCI_DEV_GET_DEV()}, and \texttt{CYG_PCI_DEV_GET_FN()} may be used to extract the constituent parts of a device id. It should not be necessary to use these macros under normal circumstances. The following code fragment demonstrates how these macros may be used:

```c
// Create a packed representation of device 1, function 0
cyg_uint8 devfn = CYG_PCI_DEV_MAKE_DEVFN(1,0);

// Create a packed devid for that device on bus 2
cyg_pci_device_id devid = CYG_PCI_DEV_MAKE_ID(2, devfn);

diag_printf("bus %d, dev %d, func %d\n",
    CYG_PCI_DEV_GET_BUS(devid),
    CYG_PCI_DEV_GET_DEV(CYG_PCI_DEV_GET_DEVFN(devid)),
    CYG_PCI_DEV_GET_FN(CYG_PCI_DEV_GET_DEVFN(devid)));
```

typedef struct cyg_pci_device;

This structure is used to contain data read from a PCI device’s configuration header by \texttt{cyg_pci_get_device_info()}. It is also used to record the resource allocations made to the device.

typedef CYG_WORD64 CYG_PCI_ADDRESS64;
typedef CYG_WORD32 CYG_PCI_ADDRESS32;

Pointers in the PCI address space are 32 bit (IO space) or 32/64 bit (memory space). In most platform and device configurations all of PCI memory will be linearly addressable using only 32 bit pointers as read from \texttt{base_map[]}.

The 64 bit type is used to allow handling 64 bit devices in the future, should it be necessary, without changing the library’s API.

Functions

```c
void cyg_pci_init(void);
```

Initialize the PCI library and establish contact with the hardware. This function is idempotent and can be called either by all drivers in the system, or just from an application initialization function.

```c
cyg_bool cyg_pci_find_device( cyg_uint16 vendor,
    cyg_uint16 device,
    cyg_pci_device_id *devid );
```

Searches the PCI bus configuration space for a device with the given \texttt{vendor} and \texttt{device} ids. The search starts at the device pointed to by \texttt{devid}, or at the first slot if it contains \texttt{CYG_PCI_NULL_DEVID}. \texttt{*devid} will be updated with the ID of the next device found. Returns \texttt{true} if one is found and \texttt{false} if not.

```c
cyg_bool cyg_pci_find_class( cyg_uint32 dev_class,
    cyg_pci_device_id *devid );
```
Chapter 25. The eCos PCI Library

Searches the PCI bus configuration space for a device with the given dev_class class code. The search starts at the device pointed to by devid, or at the first slot if it contains CYG_PCI_NULL_DEVID.

*devid will be updated with the ID of the next device found. Returns true if one is found and false if not.

cyg_bool cyg_pci_find_next( cyg_pci_device_id cur_devid,
       cyg_pci_device_id *next_devid );

Searches the PCI configuration space for the next valid device after cur_devid. If cur_devid is given the value CYG_PCI_NULL_DEVID, then the search starts at the first slot. It is permitted for next_devid to point to cur_devid. Returns true if another device is found and false if not.

cyg_bool cyg_pci_find_matching( cyg_pci_match_func *matchp,
       void * match_callback_data,
       cyg_pci_device_id *devid );

Searches the PCI bus configuration space for a device whose properties match those required by the caller supplied cyg_pci_match_func. The search starts at the device pointed to by devid, or at the first slot if it contains CYG_PCI_NULL_DEVID. The devid will be updated with the ID of the next device found. This function returns true if a matching device is found and false if not.

The match_func has a type declared as:

typedef cyg_bool (cyg_pci_match_func)( cyg_uint16 vendor,
       cyg_uint16 device,
       cyg_uint32 class,
       void * user_data);

The vendor, device, and class are from the device configuration space. The user_data is the callback data passed to cyg_pci_find_matching.

void cyg_pci_get_device_info ( cyg_pci_device_id devid,
       cyg_pci_device *dev_info );

This function gets the PCI configuration information for the device indicated in devid. The common fields of the cyg_pci_device structure, and the appropriate fields of the relevant header union member are filled in from the device's configuration space. If the device has not been enabled, then this function will also fetch the size and type information from the base address registers and place it in the base_size[] array.

void cyg_pci_set_device_info ( cyg_pci_device_id devid,
       cyg_pci_device *dev_info );

This function sets the PCI configuration information for the device indicated in devid. Only the configuration space registers that are writable are actually written. Once all the fields have been written, the device info will be read back into *dev_info, so that it reflects the true state of the hardware.

void cyg_pci_read_config_uint8( cyg_pci_device_id devid,
       cyg_uint8 offset, cyg_uint8 *val );
void cyg_pci_read_config_uint16( cyg_pci_device_id devid,
       cyg_uint8 offset, cyg_uint16 *val );
void cyg_pci_read_config_uint32( cyg_pci_device_id devid,
       cyg_uint8 offset, cyg_uint32 *val );

These functions read registers of the appropriate size from the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through cyg_pci_get_device_info().

void cyg_pci_write_config_uint8( cyg_pci_device_id devid,
       cyg_uint8 offset, cyg_uint8 val );
Chapter 25. The eCos PCI Library

void cyg_pci_write_config_uint16( cyg_pci_device_id devid, cyg_uint8 offset, cyg_uint16 val );
void cyg_pci_write_config_uint32( cyg_pci_device_id devid, cyg_uint8 offset, cyg_uint32 val );

These functions write registers of the appropriate size to the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through cyg_pci_get_device_info(). Writing the general registers this way may render the contents of a cyg_pci_device structure invalid.

Resource allocation

These routines allocate memory and I/O space to PCI devices.

cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info )

Allocate memory and IO space to all base address registers using the current memory and IO base addresses in the library. The allocated base addresses, translated into directly usable values, will be put into the matching base_map[] entries in *dev_info. If *dev_info does not contain valid base_size[] entries, then the result is false. This function will also call cyg_pci_translate_interrupt() to put the interrupt vector into the HAL vector entry.

cyg_bool cyg_pci_configure_bus( cyg_uint8 bus, cyg_uint8 *next_bus )

Allocate memory and IO space to all base address registers on all devices on the given bus and all subordinate busses. If a PCI-PCI bridge is found on bus, this function will call itself recursively in order to configure the bus on the other side of the bridge. Because of the nature of bridge devices, all devices on the secondary side of a bridge must be allocated memory and IO space before the memory and IO windows on the bridge device can be properly configured. The next_bus argument points to the bus number to assign to the next subordinate bus found. The number will be incremented as new busses are discovered. If successful, true is returned. Otherwise, false is returned.

cyg_bool cyg_pci_translate_interrupt( cyg_pci_device *dev_info, CYG_ADDRWORD *vec );

Translate the device’s PCI interrupt (INTA#-INTD#) to the associated HAL vector. This may also depend on which slot the device occupies. If the device may generate interrupts, the translated vector number will be stored in vec and the result is true. Otherwise the result is false.

cyg_bool cyg_pci_allocate_memory( cyg_pci_device *dev_info, cyg_uint32 bar, CYG_PCI_ADDRESS64 *base );
cyg_bool cyg_pci_allocate_io( cyg_pci_device *dev_info, cyg_uint32 bar, CYG_PCI_ADDRESS32 *base );

These routines allocate memory or I/O space to the base address register indicated by bar. The base address in *base will be correctly aligned and the address of the next free location will be written back into it if the allocation succeeds. If the base address register is of the wrong type for this allocation, or dev_info does not contain valid base_size[] entries, the result is false. These functions allow a device driver to set up its own mappings if it wants. Most devices should probably use cyg_pci_configure_device().

void cyg_pci_set_memory_base( CYG_PCI_ADDRESS64 base );
void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
These routines set the base addresses for memory and I/O mappings to be used by the memory allocation routines. Normally these base addresses will be set to default values based on the platform. These routines allow these to be changed by application code if necessary.

**PCI Library Hardware API**

This API is used by the PCI library to access the PCI bus configuration space. Although it should not normally be necessary, this API may also be used by device driver or application code to perform PCI bus operations not supported by the PCI library.

```c
void cyg_pcihw_init(void);
```

Initialize the PCI hardware so that the configuration space may be accessed.

```c
void cyg_pcihw_read_config_uint8( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint8 *val);
void cyg_pcihw_read_config_uint16( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint16 *val);
void cyg_pcihw_read_config_uint32( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint32 *val);
```

These functions read a register of the appropriate size from the PCI configuration space at an address composed from the `bus`, `devfn` and `offset` arguments.

```c
void cyg_pcihw_write_config_uint8( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint8 val);
void cyg_pcihw_write_config_uint16( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint16 val);
void cyg_pcihw_write_config_uint32( cyg_uint8 bus,
        cyg_uint8 devfn, cyg_uint8 offset, cyg_uint32 val);
```

These functions write a register of the appropriate size to the PCI configuration space at an address composed from the `bus`, `devfn` and `offset` arguments.

```c
cyg_bool cyg_pcihw_translate_interrupt( cyg_uint8 bus,
        cyg_uint8 devfn,
        CYG_ADDRWORD *vec);
```

This function interrogates the device and determines which HAL interrupt vector it is connected to.

**HAL PCI support**

HAL support consists of a set of C macros that provide the implementation of the low level PCI API.

```c
HAL_PCI_INIT()
```

Initialize the PCI bus.

```c
HAL_PCI_READ_UINT8( bus, devfn, offset, val )
HAL_PCI_READ_UINT16( bus, devfn, offset, val )
HAL_PCI_READ_UINT32( bus, devfn, offset, val )
```

Read a value from the PCI configuration space of the appropriate size at an address composed from the `bus`, `devfn` and `offset` arguments.

```c
HAL_PCI_WRITE_UINT8( bus, devfn, offset, val )
```
HAL_PCI_WRITE_UINT16( bus, devfn, offset, val )
HAL_PCI_WRITE_UINT32( bus, devfn, offset, val )

Write a value to the PCI configuration space of the appropriate size at an address composed from the bus, devfn and offset.

HAL_PCI_TRANSLATE_INTERRUPT( bus, devfn, *vec, valid )

Translate the device’s interrupt line into a HAL interrupt vector.

HAL_PCI_ALLOC_BASE_MEMORY
HAL_PCI_ALLOC_BASE_IO

These macros define the default base addresses used to initialize the memory and I/O allocation pointers.

HAL_PCI_PHYSICAL_MEMORY_BASE
HAL_PCI_PHYSICAL_IO_BASE

PCI memory and IO range do not always correspond directly to physical memory or IO addresses. Frequently the PCI address spaces are windowed into the processor’s address range at some offset. These macros define offsets to be added to the PCI base addresses to translate PCI bus addresses into physical memory addresses that can be used to access the allocated memory or IO space.

**Note:** The chunk of PCI memory space directly addressable though the window by the CPU may be smaller than the amount of PCI memory actually provided. In that case drivers will have to access PCI memory space in segments. Doing this will be platform specific and is currently beyond the scope of the HAL.

HAL_PCI_IGNORE_DEVICE( bus, dev, fn )

This macro, if defined, may be used to limit the devices which are found by the bus scanning functions. This is sometimes necessary for devices which need special handling. If this macro evaluates to true, the given device will not be found by cyg_pci_find_next or other bus scanning functions.

HAL_PCI_IGNORE_BAR( dev_info, bar_num )

This macro, if defined, may be used to limit which BARs are discovered and configured. This is sometimes necessary for platforms with limited PCI windows. If this macro evaluates to true, the given BAR will not be discovered by cyg_pci_get_device_info and therefore not configured by cyg_pci_configure_device.
VII. FLASH Library
Chapter 26. The eCos FLASH Library

The FLASH library is an optional part of eCos, and is only applicable to some platforms.
The eCos FLASH library provides the following functionality:

1. Identifying installed device of a FLASH family.
2. Read, erasing and writing to FLASH blocks.
3. Validating an address is within the FLASH.
4. Determining the number and size of FLASH blocks.

There are two APIs with the flash library. The old API is retained for backwards compatibility reasons, but should slowly be replaced with the new API which is much more flexible and does not pollute the name space as much.

Notes on using the FLASH library

FLASH devices cannot be read from when an erase or write operation is active. This means it is not possible to execute code from flash while an erase or write operation is active. It is possible to use the library when the executable image is resident in FLASH. The low level drivers are written such that the linker places the functions that actually manipulate the flash into RAM. However the library may not be interrupt safe. An interrupt must not cause execution of code that is resident in FLASH. This may be the image itself, or RedBoot. In some configurations of eCos, ^C on the serial port or debugging via Ethernet may cause an interrupt handler to call RedBoot. If RedBoot is resident in FLASH this will cause a crash. Similarly, if another thread invokes a virtual vector function to access RedBoot, eg to perform a diag_printf() a crash could result.

Thus with a ROM based image or a ROM based Redboot it is recommended to disable interrupts while erasing or programming flash. Using both a ROMRAM or RAM images and a ROMRAM or RAM RedBoot are safe and there is no need to disable interrupts.

Danger, Will Robinson! Danger!

Unlike nearly every other aspect of embedded system programming, getting it wrong with FLASH devices can render your target system useless. Most targets have a boot loader in the FLASH. Without this boot loader the target will obviously not boot. So before starting to play with this library its worth investigating a few things. How do you recover your target if you delete the boot loader? Do you have the necessary JTAG cable? Or is specialist hardware needed? Is it even possible to recover the target boards or must it be thrown into the rubbish bin? How does killing the board affect your project schedule?
Chapter 26. The eCos FLASH Library
Chapter 27. The Version 2 eCos FLASH API

There are two APIs described here. The first is the application API which programs should use. The second API is that between the FLASH IO library and the device drivers.

FLASH user API

All of the functions described below are declared in the header file `<cyg/io/flash.h>` which all users of the FLASH library should include.

Initializing the FLASH library

The FLASH library needs to be initialized before other FLASH operations can be performed. This only needs to be done once. The following function will only do the initialization once so it’s safe to call multiple times:

`__externC int cyg_flash_init(cyg_flash_printf *pf);`

The parameter `pf` must always be set to NULL. It exists solely for backward compatibility and other settings are deprecated and obsolete. Past use of this parameter has now been replaced with use of the `cyg_flash_set_global_printf` function.

Retrieving information about FLASH devices

The following five functions return information about the FLASH.

`__externC int cyg_flash_get_info(cyg_uint32 devno, cyg_flash_info_t * info);`
`__externC int cyg_flash_get_info_addr(cyg_flashaddr_t flash_base, cyg_flash_info_t * info);`
`__externC int cyg_flash_verify_addr(const flashaddr_t address);`
`__extern size_t cyg_flash_block_size(const cyg_flashaddr_t flash_base);`

```
typedef struct cyg_flash_block_info
{
    size_t                  block_size;
    cyg_uint32             blocks;
} cyg_flash_block_info_t;
```

```
typedef struct {
    cyg_flashaddr_t        start;     // First address
    cyg_flashaddr_t        end;       // Last address
    cyg_uint32             num_block_infos; // Number of entries
    const cyg_flash_block_info_t *blocks_info; // Info about one block size
} cyg_flash_info_t;
```

cyg_flash_get_info() is the main function to get information about installed flash devices. Parameter `devno` is used to iterate over the available flash devices, starting from 0. If the devno’th device exists, the structure pointed to by `info` is filled in and `CYG_FLASH_ERR_OK` is returned, otherwise `CYG_FLASH_ERR_INVALID`. cyg_flash_get_info() is similar, but returns the information about the flash device at the given address. cyg_flash_block_size() returns the size of the block at the given address. cyg_flash_verify_addr() tests if the target addresses is within one of the FLASH devices, returning `CYG_FLASH_ERR_OK` if so.
Chapter 27. The Version 2 eCos FLASH API

Reading from FLASH

There are two methods for reading from FLASH. The first is to use the following function.

```c
__externC int cyg_flash_read(cyg_flashaddr_t flash_base, void *ram_base, size_t len, cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to read from. `ram_base` indicates where the data read from flash should be placed into RAM. `len` is the number of bytes to be read from the FLASH and `err_address` is used to return the location in FLASH that any error occurred while reading.

The second method is to simply `memcpy()` directly from the FLASH. This is not recommended since some types of device cannot be read in this way, eg NAND FLASH. Using the FLASH library function to read the FLASH will always work so making it easy to port code from one FLASH device to another.

Erasing areas of FLASH

Blocks of FLASH can be erased using the following function:

```c
__externC int cyg_flash_erase(cyg_flashaddr_t flash_base, size_t len, cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to erase from. `len` is the minimum number of bytes to erase in the FLASH and `err_address` is used to return the location in FLASH that any error occurred while erasing. It should be noted that FLASH devices are block oriented when erasing. It is not possible to erase a few bytes within a block, the whole block will be erased. `flash_base` may be anywhere within the first block to be erased and `flash_base+len` may be anywhere in the last block to be erased.

Programming the FLASH

Programming of the flash is achieved using the following function.

```c
__externC int cyg_flash_program(cyg_flashaddr_t flash_base, void *ram_base, size_t len, cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to program from. `ram_base` indicates where the data to be programmed into FLASH should be read from in RAM. `len` is the number of bytes to be program into the FLASH and `err_address` is used to return the location in FLASH that any error occurred while programming.

Locking and unlocking blocks

Some flash devices have the ability to lock and unlock blocks. A locked block cannot be erased or programmed without it first being unlocked. For devices which support this feature and when `CYGHWR_IO_FLASH_BLOCK_LOCKING` is enabled then the following two functions are available:

```c
__externC int cyg_flash_lock(const cyg_flashaddr_t flash_base, size_t len, cyg_flashaddr_t *err_address);
__externC int cyg_flash_unlock(const cyg_flashaddr_t flash_base, size_t len, cyg_flashaddr_t *err_address);
```

Locking FLASH mutexes

When the eCos kernel package is included in the eCos configuration, the FLASH IO library will perform mutex locking on FLASH operations. This makes the API defined here thread safe. However applications may wish to directly access the contents of the FLASH. In order for this to be thread safe it is necessary for the application...
to use the following two functions to inform the FLASH IO library that the FLASH devices are being used and other API calls should be blocked.

```c
__externC int cyg_flash_mutex_lock(const cyg_flashaddr_t from, size_t len);
__externC int cyg_flash_mutex_unlock(const cyg_flashaddr_t from, size_t len);
```

### Configuring diagnostic output

Each FLASH device can have an associated function which is called to perform diagnostic output. The function to be used can be configured with the following functions:

```c
__externC int cyg_flash_set_printf(const cyg_flashaddr_t flash_base,
                                cyg_flash_printf *pf);
__externC void cyg_flash_set_global_printf(cyg_flash_printf *pf);
typedef int cyg_flash_printf(const char *fmt, ...);
```

The parameter `pf` is a pointer to a function which is to be used for diagnostic output. Typically the function `diag_printf()` will be passed. Normally this function is not used by the higher layer of the library unless `CYGSEM_IO_FLASH_CHATTER` is enabled. Passing a `NULL` causes diagnostic output from lower level drivers to be discarded.

`cyg_flash_set_printf` is used to set a diagnostic output function which will be used specifically when diagnostic output is attempted from the FLASH device driver associated with the base address of `flash_base`. An error will be returned if no FLASH device is found for this address, or the FLASH subsystem has not yet been initialised with `cyg_flash_init`.

`cyg_flash_set_global_printf` sets a diagnostic output function for all available FLASH devices. Any previous setting of a diagnostic output function (including with `cyg_flash_set_printf`) will be discarded. This function may be called prior to `cyg_flash_init`.

### Return values and errors

All the functions above return one of the following return values.

- `CYG_FLASH_ERR_OK` No error - operation complete
- `CYG_FLASH_ERR_INVALID` Invalid FLASH address
- `CYG_FLASH_ERR_ERASE_ERROR` Error trying to erase
- `CYG_FLASH_ERR_LOCK_ERROR` Error trying to lock/unlock
- `CYG_FLASH_ERR_PROGRAM_ERROR` Error trying to program
- `CYG_FLASH_ERR_PROTOCOL_ERROR` Generic error
- `CYG_FLASH_ERR_PROTECT` Device/region is write-protected
- `CYG_FLASH_ERR_NOT_INIT` FLASH info not yet initialized
- `CYG_FLASH_ERR_HWR` Hardware (configuration?) problem
- `CYG_FLASH_ERR_ERASE_SUSPEND` Device is in erase suspend mode
- `CYG_FLASH_ERR_PROGRAM_SUSPEND` Device is in program suspend mode
- `CYG_FLASH_ERR_DRV_VERIFY` Driver failed to verify data
- `CYG_FLASH_ERR_DRV_TIMEOUT` Driver timed out waiting for device
- `CYG_FLASH_ERR_DRV_WRONG_PART` Driver does not support device
- `CYG_FLASH_ERR_LOW_VOLTAGE` Not enough juice to complete job

To turn an error code into a human readable string the following function can be used:

```c
__externC const char *cyg_flash_errmsg(const int err);
```
Chapter 27. The Version 2 eCos FLASH API

FLASH device API

This section describes the API between the FLASH IO library and the FLASH device drivers.

The FLASH device Structure

This structure keeps all the information about a single driver.

```c
struct cyg_flash_dev {
    const struct cyg_flash_dev_funs *funs; // Function pointers
    cyg_uint32 flags; // Device characteristics
    cyg_flashaddr_t start; // First address
    cyg_flashaddr_t end; // Last address
    cyg_uint32 num_block_infos; // Number of entries
    const cyg_flash_block_info_t *block_info; // Info about one block size
    const void *priv; // Devices private data

    // The following are only written to by the FLASH IO layer.
    cyg_flash_printf *pf; // Pointer to diagnostic printf
    bool init; // Device has been initialised

    #ifdef CYGPKG_KERNEL
    cyg_mutex_t mutex; // Mutex for thread safeness
    #endif

    #if (CYGHWR_IO_FLASH_DEVICE > 1)
        struct cyg_flash_dev *next; // Pointer to next device
    #endif

    struct cyg_flash_dev_funs {
        int (*flash_init)(struct cyg_flash_dev *dev);
        size_t (*flash_query)(struct cyg_flash_dev *dev, void *data, size_t len);
        int (*flash_erase_block)(struct cyg_flash_dev *dev, cyg_flashaddr_t block_base);
        int (*flash_program)(struct cyg_flash_dev *dev, cyg_flashaddr_t base, const void* data, size_t len);
        int (*flash_read)(struct cyg_flash_dev *dev, const cyg_flashaddr_t base, void* data, size_t len);
    #ifdef CYGHWR_IO_FLASH_BLOCK_LOCKING
        int (*flash_block_lock)(struct cyg_flash_dev *dev, const cyg_flashaddr_t block_base);
        int (*flash_block_unlock)(struct cyg_flash_dev *dev, const cyg_flashaddr_t block_base);
    #endif
    };
};
```

The FLASH IO layer will only pass requests for operations on a single block.
Chapter 28. The legacy Version 1 eCos FLASH API

The library has a number of limitations:

1. Only one family of FLASH device may be supported at once.
2. Multiple devices of one family are supported, but they must be contiguous in memory.
3. The library is not thread or interrupt safe under some conditions.
4. The library currently does not use the eCos naming convention for its functions. This may change in the future but backward compatibility is likely to be kept.

There are two APIs described here. The first is the application API which programs should use. The second API is that between the FLASH io library and the device drivers.

FLASH user API

All of the functions described below are declared in the header file `<cyg/io/flash.h>` which all users of the FLASH library should include.

Initializing the FLASH library

The FLASH library needs to be initialized before other FLASH operations can be performed. This only needs to be done once. The following function will only do the initialization once so it’s safe to call multiple times:

```c
externC int flash_init( _printf *pf );
typedef int _printf(const char *fmt, ...);
```

The parameter `pf` is a pointer to a function which is to be used for diagnostic output. Typically the function `diag_printf()` will be passed. Normally this function is not used by the higher layer of the library unless `CYGSEM_IO_FLASH_CHATTER` is enabled. Passing a `NULL` is not recommended, even when `CYGSEM_IO_FLASH_CHATTER` is disabled. The lower layers of the library may unconditionally call this function, especially when errors occur, probably resulting in a more serious error/crash!

Retrieving information about the FLASH

The following four functions return information about the FLASH.

```c
externC int flash_get_block_info(int *block_size, int *blocks);
externC int flash_get_limits(void *target, void **start, void **end);
externC int flash_verify_addr(void *target);
externC bool flash_code_overlaps(void *start, void *end);
```

The function `flash_get_block_info()` returns the size and number of blocks. When the device has a mixture of block sizes, the size of the "normal" block will be returned. Please read the source code to determine exactly what this means. `flash_get_limits()` returns the lower and upper memory address the FLASH occupies. The
target parameter is current unused. flash_verify_addr() tests if the target addresses is within the flash, returning FLASH_ERR_OK if so. Lastly, flash_code_overlaps() checks if the executing code is resident in the section of flash indicated by start and end. If this function returns true, erase and program operations within this range are very likely to cause the target to crash and burn horribly. Note the FLASH library does allow you to shoot yourself in the foot in this way.

Reading from FLASH

There are two methods for reading from FLASH. The first is to use the following function.

```c
externC int flash_read(void *flash_base, void *ram_base, int len, void **err_address);
```

flash_base is where in the flash to read from. ram_base indicates where the data read from flash should be placed into RAM. len is the number of bytes to be read from the FLASH and err_address is used to return the location in FLASH that any error occurred while reading.

The second method is to simply memcpy() directly from the FLASH. This is not recommended since some types of device cannot be read in this way, eg NAND FLASH. Using the FLASH library function to read the FLASH will always work so making it easy to port code from one FLASH device to another.

Erasing areas of FLASH

Blocks of FLASH can be erased using the following function:

```c
externC int flash_erase(void *flash_base, int len, void **err_address);
```

flash_base is where in the flash to erase from. len is the minimum number of bytes to erase in the FLASH and err_address is used to return the location in FLASH that any error occurred while erasing. It should be noted that FLASH devices are block oriented when erasing. It is not possible to erase a few bytes within a block, the whole block will be erased. flash_base may be anywhere within the first block to be erased and flash_base+len may be anywhere in the last block to be erased.

Programming the FLASH

Programming of the flash is achieved using the following function.

```c
externC int flash_program(void *flash_base, void *ram_base, int len, void **err_address);
```

flash_base is where in the flash to program from. ram_base indicates where the data to be programmed into FLASH should be read from in RAM. len is the number of bytes to be program into the FLASH and err_address is used to return the location in FLASH that any error occurred while programming.

Locking and unlocking blocks

Some flash devices have the ability to lock and unlock blocks. A locked block cannot be erased or programmed without it first being unlocked. For devices which support this feature and when CYGHWR_IO_FLASH_BLOCK_LOCKING is enabled then the following two functions are available:

```c
externC int flash_lock(void *flash_base, int len, void **err_address);
externC int flash_unlock(void *flash_base, int len, void **err_address);
```
Return values and errors

All the functions above, except flash_code_overlaps() return one of the following return values.

- FLASH_ERR_OK: No error - operation complete
- FLASH_ERR_INVALID: Invalid FLASH address
- FLASH_ERR_ERASE: Error trying to erase
- FLASH_ERR_LOCK: Error trying to lock/unlock
- FLASH_ERR_PROGRAM: Error trying to program
- FLASH_ERR_PROTOCOL: Generic error
- FLASH_ERR_PROTECT: Device/region is write-protected
- FLASH_ERR_NOT_INIT: FLASH info not yet initialized
- FLASH_ERR_HWR: Hardware (configuration?) problem
- FLASH_ERR_ERASE_SUSPEND: Device is in erase suspend mode
- FLASH_ERR_PROGRAM_SUSPEND: Device is in program suspend mode
- FLASH_ERR_DRV_VERIFY: Driver failed to verify data
- FLASH_ERR_DRV_TIMEOUT: Driver timed out waiting for device
- FLASH_ERR_DRV_WRONG_PART: Driver does not support device
- FLASH_ERR_LOW_VOLTAGE: Not enough juice to complete job

To turn an error code into a human readable string the following function can be used:

```
externC char *flash_errmsg(int err);
```

Notes on using the FLASH library

The FLASH library evolved from the needs and environment of RedBoot rather than being a general purpose eCos component. This history explains some of the problems with the library.

The library is not thread safe. Multiple simultaneous calls to its library functions will likely fail and may cause a crash. It is the callers responsibility to use the necessary mutex’s if needed.

FLASH device API

This section describes the API between the FLASH IO library the FLASH device drivers.

The flash_info structure

The flash_info structure is used by both the FLASH IO library and the device driver.

```c
struct flash_info {
    int block_size; // Assuming fixed size "blocks"
    int blocks; // Number of blocks
    int buffer_size; // Size of write buffer (only defined for some devices)
    unsigned long block_mask;
    void *start, *end; // Address range
    int init; // FLASH API initialised
    _printf *pf; // printf like function for diagnostics
};
```

block_mask is used internally in the FLASH IO library. It contains a mask which can be used to turn an arbitrary address in flash to the base address of the block which contains the address.
There exists one global instance of this structure with the name `flash_info`. All calls into the device driver makes use of this global structure to maintain state.

**Initializing the device driver**

The FLASH IO library will call the following function to initialize the device driver:

```c
externC int flash_hwr_init(void);
```

The device driver should probe the hardware to see if the FLASH devices exist. If it does it should fill in `start`, `end`, `blocks` and `block_size`. If the FLASH contains a write buffer the size of this should be placed in `buffer_size`. On successful probing the function should return `FLASH_ERR_OK`. When things go wrong it can be assumed that `pf` points to a printf like function for outputting error messages.

**Querying the FLASH**

FLASH devices can be queried to return there manufacture ID, size etc. This function allows this information to be returned.

```c
int flash_query(unsigned char *data);
```

The caller must know the size of data to be returned and provide an appropriately sized buffer pointed to be parameter `data`. This function is generally used by `flash_hwr_init()`.

**Erasing a block of FLASH**

So that the FLASH IO layer can erase a block of FLASH the following function should be provided.

```c
int flash_erase_block(volatile flash_t *block, unsigned int block_size);
```

**Programming a region of FLASH**

The following function must be provided so that data can be written into the FLASH.

```c
int flash_program_buf(volatile flash_t *addr, flash_t *data, int len,
                      unsigned long block_mask, int buffer_size);
```

The device will only be asked to program data in one block of the flash. The FLASH IO layer will break longer user requests into a smaller writes.

**Reading a region from FLASH**

Some FLASH devices are not memory mapped so it is not possible to read there contents directly. The following function read a region of FLASH.

```c
int flash_read_buf(volatile flash_t* addr, flash_t* data, int len);
```

As with writing to the flash, the FLASH IO layer will break longer user requests for data into a number of reads which are at maximum one block in size.
A device which cannot be read directly should set CYGSEM_IO_FLASH_READ_INDIRECT so that the IO layer makes use of the flash_read_buf() function.

### Locking and unlocking FLASH blocks

Some flash devices allow blocks to be locked so that they cannot be written to. The device driver should provide the following functions to manipulate these locks.

```c
int flash_lock_block(volatile flash_t *block);
int flash_unlock_block(volatile flash_t *block, int block_size, int blocks);
```

These functions are only used if CYGHWR_IO_FLASH_BLOCK_LOCKING

### Mapping FLASH error codes to FLASH IO error codes

The functions `flash_erase_block()`, `flash_program_buf()`, `flash_read_buf()`, `flash_lock_block()` and `flash_unlock_block()` return an error code which is specific to the flash device. To map this into a FLASH IO error code, the driver should provide the following function:

```c
int flash_hwr_map_error(int err);
```

### Determining if code is in FLASH

Although a general function, the device driver is expected to provide the implementation of the function `flash_code_overlaps()`.

### Implementation Notes

The FLASH IO layer will manipulate the caches as required. The device drivers do not need to enable/disable caches when performing operations of the FLASH.

Device drivers should keep all chatter to a minimum when CYGSEM_IO_FLASH_CHATTER is not defined. All output should use the print function in the pf in `flash_info` and not `diag_printf()`.

Device driver functions which manipulate the state of the flash so that it cannot be read from for program execute need to ensure there code is placed into RAM. The linker will do this if the appropriate attribute is added to the function. e.g:

```c
int flash_program_buf(volatile flash_t *addr, flash_t *data, int len,
                      unsigned long block_mask, int buffer_size)
    __attribute__((section(".2ram.flash_program_buf")));}
```
Chapter 29. FLASH I/O devices

It can be useful to be able to access FLASH devices using the generic I/O infrastructure found in CYGPKG_USERIO, and the generic FLASH layer provides an optional ability to do so. This allows the use of functions like cyg_io_lookup(), cyg_io_read(), cyg_io_write() etc.

Additionally it means that, courtesy of the "devfs" pseudo-filesystem in the file I/O layer (CYGPKG_USER_FILEIO), functions like open(), read(), write() etc. can even be used directly on the FLASH devices.

Overview and CDL Configuration

This package implements support for FLASH as an I/O device by exporting it as if it is a block device. To enable this support, the CDL option titled “Provide /dev block devices”, also known as CYGPKG_USER_FLASH_BLOCK_DEVICE, must be enabled. (There is also a legacy format alternative which is now deprecated).

There are two methods of addressing FLASH as a block device:

1. Using the FLASH Information System (FIS) - this is a method of defining and naming FLASH partitions, usually in RedBoot. This option is only valid if RedBoot is resident and was used to boot the application. To reference FLASH partitions in this way, you would use a device name of the form /dev/flash/fis/partition-name, for example /dev/flash/fis/jffs2 to reference a FIS partition named JFFS2.

The CDL option CYGFUN_USER_FLASH_BLOCK_FROM_FIS must be enabled for this support.

2. Referencing by device number, offset and length - this method extracts addressing information from the name itself. The form of the device would be /dev/flash/device-number/offset[,length]

   device-number
   
   This is a fixed number allocated to identify each FLASH region in the system. The first region is numbered 0, the second 1, and so on. If you have only one FLASH device, it will be numbered 0.

   offset
   
   This is the index into the FLASH region in bytes to use. It may be specified as decimal, or if prefixed with 0x, then hexadecimal.

   length
   
   This field is optional and defaults to the remainder of the FLASH region. Again it may be specified in decimal or hexadecimal.

Some examples:

/dev/flash/0/0

This defines a block device that uses the entirety of FLASH region 0.
Chapter 29. FLASH I/O devices

/dev/flash/1/0x20000,65536

This defines a block device which points inside FLASH region 1, starting at offset 0x20000 (128Kb) and extending for 64Kb.

/dev/flash/0/65536

This defines a block device which points inside FLASH region 0, starting at offset 64Kb and continuing up to the end of the device.

Obviously great care is required when constructing the device names as using the wrong specification may subsequently overwrite important areas of FLASH, such as RedBoot. Using the alternative via FIS names is preferable as these are less error-prone to configure, and also allows for the FLASH region to be relocated without requiring program recompilation.

Using FLASH I/O devices

The FLASH I/O block devices can be accessed, read and written using the standard interface supplied by the generic I/O (CYGPKG_IO) package. These include the functions: cyg_io_lookup() to access the device and get a handle, cyg_io_read() and cyg_io_write() for sequential read and write operations, cyg_io_bread() and cyg_io_bwrite() for random access read and write operations, and cyg_io_get_config() and cyg_io_setconfig() for run-time configuration inspection and control.

However there are two aspects that differ from some other I/O devices accessed this way:

1. The first is that the lookup operation uses up resources which must be subsequently freed when the last user of the I/O handle is finished. The number of FLASH I/O devices that may be simultaneously opened is configured with the CYGNUM_IO_FLASH_BLOCK_DEVICES CDL option. After the last user is finished, the device may be closed using cyg_io_setconfig() with the CYG_IO_SET_CONFIG_CLOSE key. Reference counting to ensure that it is only the last user that causes a close, is left to higher layers.

2. The second is that write operations assume that the flash is already erased. Attempting to write to Flash that has already been written to may result in errors. Instead FLASH must be erased before it may be written.

FLASH block devices can also be read and written using the standard POSIX primitives, open(), close(), read(), write(), lseek(), and so on if the POSIX file I/O package (CYGPKG_FILEIO) is included in the configuration. As with the eCos generic I/O interface you must call close() to ensure resources are freed when the device is no longer used.

Other configuration keys are provided to perform FLASH erase operations, and to retrieve device sizes, and FLASH block sizes at a particular address. These operations are accessed with cyg_io_get_config() (or if using the POSIX file I/O API, cyg_fs_getinfo()) with the following keys:

CYG_IO_GET_CONFIG_FLASH_ERASE

This erases a region of FLASH. cyg_io_get_config() must be passed a structure defined as per the following, which is also supplied in <cyg/io/flash.h>:

typedef struct {
    CYG_ADDRESS offset;
    size_t len;
    int flasherr;
    cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_erase_t;
In this structure, `offset` specifies the offset within the block device to erase, `len` specifies the amount to address, `flasherr` is set on return to specify an error with the FLASH erase operation itself, and `err_address` is used if there was an error to specify at which address the error happened.

**CYG_IO_GET_CONFIG_FLASH_LOCK**

This protects a region of FLASH using the locking facilities available on the card, if provided by the underlying driver. `cyg_io_get_config()` must be passed a structure defined as per the following:

```c
typedef struct {
    CYG_ADDRESS offset;
    size_t len;
    int flasherr;
    cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_lock_t;
```

In this structure, `offset` specifies the offset within the block device to lock, `len` specifies the amount to address, `flasherr` is set on return to specify an error with the FLASH lock operation itself, and `err_address` is used if there was an error to specify at which address the error happened. If locking support is not available -EINVALER will be returned from `cyg_io_get_config()`.

**CYG_IO_GET_CONFIG_FLASH_UNLOCK**

This disables protection for a region of FLASH using the unlocking facilities available on the card, if provided by the underlying driver. `cyg_io_get_config()` must be passed a structure defined as per the following:

```c
typedef struct {
    CYG_ADDRESS offset;
    size_t len;
    int flasherr;
    cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_unlock_t;
```

In this structure, `offset` specifies the offset within the block device to unlock, `len` specifies the amount to address, `flasherr` is set on return to specify an error with the FLASH unlock operation itself, and `err_address` is used if there was an error to specify at which address the error happened. If unlocking support is not available -EINVALER will be returned from `cyg_io_get_config()`.

**CYG_IO_GET_CONFIG_FLASH_DEVSIZE**

This returns the size of the FLASH block device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```c
typedef struct {
    size_t dev_size;
} cyg_io_flash_getconfig_devsize_t;
```

In this structure, `dev_size` is used to return the size of the FLASH device.

**CYG_IO_GET_CONFIG_FLASH_DEVADDR**

This returns the address in the virtual memory map that the generic flash layer has been informed that this FLASH device is mapped to. Note that some flash devices such as dataflash are not truly memory mapped, and so this function only returns useful information when used with a true memory mapped FLASH device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```c
typedef struct {
    cyg_flashaddr_t dev_addr;
} cyg_io_flash_getconfig_devaddr_t;
```
In this structure, `dev_addr` is used to return the address corresponding to the base of the FLASH device in the virtual memory map.

`CYG_IO_GET_CONFIG_FLASH_BLOCKSIZE`

This returns the size of a FLASH block at a supplied offset in the FLASH block device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```c
typedef struct {
    CYG_ADDRESS offset;
    size_t block_size;
} cyg_io_flash_getconfig_blocksize_t;
```

In this structure, `offset` specifies the address within the block device of which the FLASH block size is required - a single FLASH device may contain blocks of differing sizes. The `block_size` field is used to return the block size at the specified offset.
VIII. SPI Support
Overview

Name

Overview — eCos Support for SPI, the Serial Peripheral Interface

Description

The Serial Peripheral Interface (SPI) is one of a number of serial bus technologies. It can be used to connect a processor to one or more peripheral chips, for example analog-to-digital convertors or real time clocks, using only a small number of pins and PCB tracks. The technology was originally developed by Motorola but is now also supported by other vendors.

A typical SPI system might look like this:

At the start of a data transfer the master cpu asserts one of the chip select signals and then generates a clock signal. During each clock tick the cpu will output one bit on its master-out-slave-in line and read one bit on the master-in-slave-out line. Each device is connected to the clock line, the two data lines, and has its own chip select. If a device’s chip select is not asserted then it will ignore any incoming data and will tristate its output. If a device’s chip select is asserted then during each clock tick it will read one bit of data on its input pin and output one bit on its output pin.

The net effect is that the cpu can write an arbitrary amount of data to one of the attached devices at a time, and simultaneously read the same amount of data. Some devices are inherently uni-directional. For example an LED unit would only accept data from the cpu: it will not send anything meaningful back; the cpu will still sample its input every clock tick, but this should be discarded.

A useful feature of SPI is that there is no flow control from the device back to the cpu. If the cpu tries to communicate with a device that is not currently present, for example an MMC socket which does not contain
Overview

a card, then the I/O will still proceed. However the cpu will read random data. Typically software-level CRC
checksums or similar techniques will be used to allow the cpu to detect this.

SPI communication is not fully standardized. Variations between devices include the following:

1. Many devices involve byte transfers, where the unit of data is 8 bits. Others use larger units, up to 16 bits.
2. Chip selects may be active-high or active-low. If the attached devices use a mixture of polarities then this
can complicate things.
3. Clock rates can vary from 128KHz to 20MHz or greater. With some devices it is necessary to interrogate
the device using a slow clock, then use the obtained information to select a faster clock for subsequent
transfers.
4. The clock is inactive between data transfers. When inactive the clock’s polarity can be high or low.
5. Devices depend on the phase of the clock. Data may be sampled on either the rising edge or the falling
edge of the clock.
6. A device may need additional delays, for example between asserting the chip select and the first clock
tick.
7. Some devices involve complicated transactions: perhaps a command from cpu to device; then an initial
status response from the device; a data transfer; and a final status response. From the cpu’s perspective
these are separate stages and it may be necessary to abort the operation after the initial status response.
However the device may require that the chip select remain asserted for the whole transaction. A side effect
of this is that it is not possible to do a quick transfer with another device in the middle of the transaction.
8. Certain devices, for example MMC cards, depend on a clock signal after a transfer has completed and the
chip select has dropped. This clock is used to finish some processing within the device.

Inside the cpu the clock and data signals are usually managed by dedicated hardware. Alternatively SPI can
be implemented using bit-banging, but that approach is normally used for other serial bus technologies such as
I2C. The chip selects may also be implemented by the dedicated SPI hardware, but often GPIO pins are used
instead.

eCos Support for SPI

The eCos SPI support for any given platform is spread over a number of different packages:

- This package, CYGPKG_IO_SPI, exports an API for accessing devices attached to an SPI bus. This API
  handles issues such as locking between threads. The package does not contain any hardware-specific code,
  instead it will call into an SPI bus driver package.

  In future this package may be extended with a bit-banging implementation of an SPI bus driver. This would
  depend on lower-level code for manipulating the GPIO pins used for the clock, data and chip select signals,
  but timing and framing could be handled by generic code.

- There will be a bus driver package for the specific SPI hardware on the target hardware, for example
  CYGPKG_DEVS_SPI_MCF52xx_QSPI. This is responsible for the actual I/O. A bus driver may be used on
  many different boards, all with the same SPI bus but with different devices attached to that bus. Details of
  the actual devices should be supplied by other code.

- The generic API depends on cyg_spi_device data structures. These contain the information needed by a
  bus driver, for example the appropriate clock rate and the chip select to use. Usually the data structures are
  provided by the platform HAL since it is that package which knows about all the devices on the board.
On some development boards the SPI pins are brought out to expansion connectors, allowing end users to add extra devices. In such cases the platform HAL may not know about all the devices on the board. Data structures for the additional devices can instead be supplied by application code.

- Some types of SPI devices may have their own driver package. For example one common use for SPI buses is to provide low-cost MultiMediaCard (MMC) support. An MMC is a non-trivial device so there is an eCos package specially for that, providing a block device interface for higher-level code such as file systems. Other SPI devices such as analog-to-digital converters are much simpler and come in many varieties. There are no dedicated packages to support each such device: the chances are low that another board would use the exact same device, so there are no opportunities for code re-use. Instead the devices may be accessed directly by application code or by extra functions in the platform HAL.

Typically all appropriate packages will be loaded automatically when you configure eCos for a given target. If the application does not use any of the SPI I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the target, so the platform HAL documentation should be checked for further details.

There is one important exception to this: if the SPI devices are attached to an expansion connector then the platform HAL will not know about these devices. Instead more work will have to be done by application code.
Overview
**SPI Interface**

**Name**

SPI Functions — allow applications and other packages to access SPI devices

**Synopsis**

```c
#include <cyg/io/spi.h>

void cyg_spi_transfer(cyg_spi_device* device, cyg_bool polled, cyg_uint32 count, const cyg_uint8* tx_data, cyg_uint8* rx_data);
void cyg_spi_tick(cyg_spi_device* device, cyg_bool polled, cyg_uint32 count);
int cyg_spi_get_config(cyg_spi_device* device, cyg_uint32 key, void* buf, cyg_uint32* len);
int cyg_spi_set_config(cyg_spi_device* device, cyg_uint32 key, const void* buf, cyg_uint32* len);
void cyg_spi_transaction_begin(cyg_spi_device* device);
cyg_bool cyg_spi_transaction_begin_nb(cyg_spi_device* device);
void cyg_spi_transaction_transfer(cyg_spi_device* device, cyg_bool polled, cyg_uint32 count, const cyg_uint8* tx_data, cyg_uint8* rx_data, cyg_bool drop_cs);
void cyg_spi_transaction_tick(cyg_spi_device* device, cyg_bool polled, cyg_uint32 count);
void cyg_spi_transaction_end(cyg_spi_device* device);
```

**Description**

All SPI functions take a pointer to a cyg_spi_device structure as their first argument. This is an opaque data structure, usually provided by the platform HAL. It contains the information needed by the SPI bus driver to interact with the device, for example the required clock rate and polarity.

An SPI transfer involves the following stages:

1. Perform thread-level locking on the bus. Only one thread at a time is allowed to access an SPI bus. This eliminates the need to worry about locking at the bus driver level. If a platform involves multiple SPI buses then each one will have its own lock. Prepare the bus for transfers to the specified device, for example by making sure it will tick at the right clock rate.

2. Assert the chip select on the specified device, then transfer data to and from the device. There may be a single data transfer or a sequence. It may or may not be necessary to keep the chip select asserted throughout a sequence.

3. Optionally generate some number of clock ticks without asserting a chip select, for those devices which need this to complete an operation.

4. Return the bus to a quiescent state. Then unlock the bus, allowing other threads to perform SPI operations on devices attached to this bus.

The simple functions `cyg_spi_transfer` and `cyg_spi_tick` perform all these steps in a single call. These are suitable for simple I/O operations. The alternative transaction-oriented functions each perform just one of these steps. This makes it possible to perform multiple transfers while only locking and unlocking the bus once, as required for more complicated devices.
**SPI Interface**

With the exception of `cyg_spi_transaction_begin_nb` all the functions will block until completion. There are no error conditions. An SPI transfer will always take a predictable amount of time, depending on the transfer size and the clock rate. The SPI bus does not receive any feedback from a device about possible errors, instead those have to be handled by software at a higher level. If a thread cannot afford the time it will take to perform a complete large transfer then a number of smaller transfers can be used instead.

SPI operations should always be performed at thread-level or during system initialization, and not inside an ISR or DSR. This greatly simplifies locking. Also a typical ISR or DSR should not perform a blocking operation such as an SPI transfer.

SPI transfers can happen in either polled or interrupt-driven mode. Typically polled mode should be used during system initialization, before the scheduler has been started and interrupts have been enabled. Polled mode should also be used in single-threaded applications such as RedBoot. A typical multi-threaded application should normally use interrupt-driven mode because this allows for more efficient use of cpu cycles. Polled mode may be used in a multi-threaded application but this is generally undesirable: the cpu will spin while waiting for a transfer to complete, wasting cycles; also the current thread may get preempted or timesliced, making the timing of an SPI transfer much less predictable. On some hardware interrupt-driven mode is impossible or would be very inefficient. In such cases the bus drivers will only support polled mode and will ignore the `polled` argument.

**Simple Transfers**

`cyg_spi_transfer` can be used for SPI operations to simple devices. It takes the following arguments:

- `cyg_spi_device* device`
  - This identifies the SPI device that should be used.

- `cyg_bool polled`
  - Polled mode should be used during system initialization and in a single-threaded application. Interrupt-driven mode should normally be used in a multi-threaded application.

- `cyg_uint32 count`
  - This identifies the number of data items to be transferred. Usually each data item is a single byte, but some devices use a larger size up to 16 bits.

- `const cyg_uint8* tx_data`
  - The data to be transferred to the device. If the device will only output data and ignore its input then a null pointer can be used. Otherwise the array should contain `count` data items, usually bytes. For devices where each data item is larger than one byte the argument will be interpreted as an array of shorts instead, and should be aligned to a 2-byte boundary. The bottom n bits of each short will be sent to the device. The buffer need not be aligned to a cache-line boundary, even for SPI devices which use DMA transfers, but some bus drivers may provide better performance if the buffer is suitably aligned. The buffer will not be modified by the transfer.

- `cyg_uint8* rx_data`
  - A buffer for the data to be received from the device. If the device does not generate any output then a null pointer can be used. The same size and alignment rules apply as for `tx_data`.

`cyg_spi_transfer` performs all the stages of an SPI transfer: locking the bus; setting it up correctly for the specified device; asserting the chip select and transferring the data; dropping the chip select at the end of the transfer; returning the bus to a quiescent state; and unlocking the bus.
SPI Interface

Additional Clock Ticks

Some devices require a number of clock ticks on the SPI bus between transfers so that they can complete some internal processing. These ticks must happen at the appropriate clock rate but no chip select should be asserted and no data transfer will happen. `cyg_spi_tick` provides this functionality. The `device` argument identifies the SPI bus, the required clock rate and the size of each data item. The `polled` argument has the usual meaning. The `count` argument specifies the number of data items that would be transferred, which in conjunction with the size of each data item determines the number of clock ticks.

Transactions

A transaction-oriented API is available for interacting with more complicated devices. This provides separate functions for each of the steps in an SPI transfer.

`cyg_spi_transaction_begin` must be used at the start of a transaction. This performs thread-level locking on the bus, blocking if it is currently in use by another thread. Then it prepares the bus for transfers to the specified device, for example by making sure it will tick at the right clock rate.

`cyg_spi_transaction_begin_nb` is a non-blocking variant, useful for threads which cannot afford to block for an indefinite period. If the bus is currently locked the function returns false immediately. If the bus is not locked then it acts as `cyg_spi_transaction_begin` and returns true.

Once the bus has been locked it is possible to perform one or more data transfers by calling `cyg_spi_transaction_transfer`. This takes the same arguments as `cyg_spi_transfer`, plus an additional one `drop_cs`. A non-zero value specifies that the device’s chip select should be dropped at the end of the transfer, otherwise the chip select remains asserted. It is essential that the chip select be dropped in the final transfer of a transaction. If the protocol makes this difficult then `cyg_spi_transaction_tick` can be used to generate dummy ticks with all chip selects dropped.

If the device requires additional clock ticks in the middle of a transaction without being selected, `cyg_spi_transaction_tick` can be used. This will drop the device’s chip select if necessary, then generate the appropriate number of ticks. The arguments are the same as for `cyg_spi_tick`.

`cyg_spi_transaction_end` should be called at the end of a transaction. It returns the SPI bus to a quiescent state, then unlocks it so that other threads can perform I/O.

A typical transaction might involve the following. First a command should be sent to the device, consisting of four bytes. The device will then respond with a single status byte, zero for failure, non-zero for success. If successful then the device can accept another n bytes of data, and will generate a 2-byte response including a checksum. The device’s chip select should remain asserted throughout. The code for this would look something like:

```c
#include <cyg/io/spi.h>
#include <cyg/hal/hal_io.h>  // Defines the SPI devices
...

cyg_spi_transaction_begin(&hal_spi_emrom);
// Interrupt-driven transfer, four bytes of command

cyg_spi_transaction_transfer(&hal_spi_emrom, 0, 4, command, NULL, 0);
// Read back the status

cyg_spi_transaction_transfer(&hal_spi_emrom, 0, 1, NULL, status, 0);
if (!status[0]) {
    // Command failed, generate some extra ticks to drop the chip select
    cyg_spi_transaction_tick(&hal_spi_emrom, 0, 1);
} else {
    // Transfer the data, then read back the final status. The
    // chip select should be dropped at the end of this.
    cyg_spi_transaction_transfer(&hal_spi_emrom, 0, n, data, NULL, 0);
}`
SPI Interface

cyg_spi_transaction_transfer(shal_spi_eprom, 0, 2, NULL, status, 1);
    // Code for checking the final status should go here
}

// Transaction complete so clean up
cyg_spi_transaction_end(&hal_spi_eprom);

A number of variations are possible. For example the command and status could be packed into the beginning and end of two 5-byte arrays, allowing a single transfer.

Device Configuration

The functions `cyg_spi_get_config` and `cyg_spi_set_config` can be used to examine and change parameters associated with SPI transfers. The only keys that are defined for all devices are `CYG_IO_GET_CONFIG_SPI_CLOCKRATE` and `CYG_IO_SET_CONFIG_SPI_CLOCKRATE`. Some types of device, for example MMC cards, support a range of clock rates. The `cyg_spi_device` structure will be initialized with a low clock rate. During system initialization the device will be queried for the optimal clock rate, and the `cyg_spi_device` should then be updated. The argument should be a clock rate in Hertz. For example the following code switches communication to 1Mbit/s:

cyg_uint32 new_clock_rate = 1000000;
cyg_uint32 len = sizeof(cyg_uint32);
if (cyg_spi_set_config(&hal_mmc_device,
    CYG_IO_SET_CONFIG_SPI_CLOCKRATE,
    (const void *)&new_clock_rate, &len)) {
    // Error recovery code
}

If an SPI bus driver does not support the exact clock rate specified it will normally use the nearest valid one. SPI bus drivers may define additional keys appropriate for specific hardware. This means that the valid keys are not known by the generic code, and theoretically it is possible to use a key that is not valid for the SPI bus to which the device is attached. It is also possible that the argument used with one of these keys is invalid. Hence both `cyg_spi_get_config` and `cyg_spi_set_config` can return error codes. The return value will be 0 for success, non-zero for failure. The SPI bus driver’s documentation should be consulted for further details.

Both configuration functions will lock the bus, in the same way as `cyg_spi_transfer`. Changing the clock rate in the middle of a transfer or manipulating other parameters would have unexpected consequences.
Porting to New Hardware

Name
Porting — Adding SPI support to new hardware

Description
Adding SPI support to an eCos port can take two forms. If there is already an SPI bus driver for the target hardware then both that driver and this generic SPI package CYGPKG_IO_SPI should be included in the ecos.db target entry. Typically the platform HAL will need to supply some platform-specific information needed by the bus driver. In addition the platform HAL should provide cyg_spi_device structures for every device attached to the bus. The exact details of this depend on the bus driver so its documentation should be consulted for further details. If there is no suitable SPI bus driver yet then a new driver package will have to be written.

Adding a Device
The generic SPI package CYGPKG_IO_SPI defines a data structure cyg_spi_device. This contains the information needed by the generic package, but not the additional information needed by a bus driver to interact with the device. Each bus driver will define a larger data structure, for example cyg_mcf52xx_qspi_device, which contains a cyg_spi_device as its first field. This is analogous to C++ base and derived classes, but without any use of virtual functions. The bus driver package should be consulted for the details.

During initialization an SPI bus driver may need to know about all the devices attached to that bus. For example it may need to know which cpu pins should be configured as chip selects rather than GPIO pins. To achieve this all device definitions should specify the particular bus to which they are attached, for example:

```c
struct cyg_mcf52xx_qspi_device hal_spi_atod CYG_SPI_DEVICE_ON_BUS(0) = {
    .spi_common.spi_bus = &cyg_mcf52xx_qspi_bus,
    ...
};
```

The CYG_SPI_DEVICE_ON_BUS macro adds information to the structure which causes the linker to group all such structures in a single table. The bus driver’s initialization code can then iterate over this table.

Adding Bus Support
An SPI bus driver usually involves a new hardware package. This needs to perform the following:

1. Define a device structure which contains a cyg_spi_device as its first element. This should contain all the information needed by the bus driver to interact with a device on that bus.

2. Provide functions for the following operations:
   - spi_transaction_begin
   - spi_transaction_transfer
   - spi_transaction_tick
   - spi_transaction_end
   - spi_get_config
   - spi_set_config

   These correspond to the main API functions, but can assume that the bus is already locked so no other
thread will be manipulating the bus or any of the attached devices. Some of these operations may be no-ops.

3. Define a bus structure which contains a cyg_spi_bus as its first element. This should contain any additional information needed by the bus driver.

4. Optionally, instantiate the bus structure. The instance should have a well-known name since it needs to be referenced by the device structure initializers. For some drivers it may be best to create the bus inside the driver package. For other drivers it may be better to leave this to the platform HAL or the application. It depends on how much platform-specific knowledge is needed to fill in the bus structure.

5. Create a HAL table for the devices attached to this bus.

6. Arrange for the bus to be initialized early on during system initialization. Typically this will happen via a prioritized static constructor with priority CYG_INIT_BUS_SPI. As part of this initialization the bus driver should invoke the CYG_SPI_BUS_COMMON_INIT macro on its cyg_spi_bus field.

7. Provide the appropriate documentation, including details of how the SPI device structures should be initialized.

There are no standard SPI testcases. It is not possible to write SPI code without knowing about the devices attached to the bus, and those are inherently hardware-specific.
IX. I2C Support
Overview

Name

Overview — eCos Support for I2C, the Inter IC Bus

Description

The Inter IC Bus (I2C) is one of a number of serial bus technologies. It can be used to connect a processor to one or more peripheral chips, for example analog-to-digital convertors or real time clocks, using only a small number of pins and PCB tracks. The technology was originally developed by Philips Semiconductors but is supported by many other vendors. The bus specification is freely available.

In a typical I2C system the processor acts as the I2C bus master. The peripheral chips act as slaves. The bus consists of just two wires: SCL carries a clock signal generated by the master, and SDA is a bi-directional data line. The normal clock frequency is 100KHz. Each slave has a 7-bit address. With some chips the address is hard-wired, and it is impossible to have two of these chips on the same bus. With other chips it is possible to choose between one of a small number of addresses by connecting spare pins to either VDD or GND.

An I2C data transfer involves a number of stages:

1. The bus master generates a start condition, a high-to-low transition on the SDA line while SCL is kept high. This signalling cannot occur during data transfer.
2. The bus master clocks the 7-bit slave address onto the SDA line, followed by a direction bit to distinguish between reads and writes.
3. The addressed device acknowledges. If the master does not see an acknowledgement then this suggests it is using the wrong address for the slave device.
4. If the master is transmitting data to the slave then it will send this data one byte at a time. The slave acknowledges each byte. If the slave is unable to accept more data, for example because it has run out of buffer space, then it will generate a nack and the master should stop sending.
5. If the master is receiving data from the slave then the slave will send this data one byte at a time. The master should acknowledge each byte, until the last one. When the master has received all the data it wants it should generate a nack and the slave will stop sending. This nack is essential because it causes the slave to stop driving the SDA line, releasing it back to the master.
6. It is possible to switch direction in a single transfer, using what is known as a repeated start. This involves generating another start condition, sending the 7-bit address again, followed by a new direction bit.
7. At the end of a transfer the master should generate a stop condition, a low-to-high transition on the SDA line while SCL is kept high. Again this signalling does not occur at other times.

There are a number of extensions. The I2C bus supports multiple bus masters and there is an arbitration procedure to allow a master to claim the bus. Some devices can have 10-bit addresses rather than 7-bit addresses. There is a fast mode operating at 400KHz instead of the usual 100KHz, and a high-speed mode operating at 3.4MHz. Currently most I2C-based systems do not involve any of these extensions.

At the hardware level I2C bus master support can be implemented in one of two ways. Some processors provide a dedicated I2C device, with the hardware performing much of the work. On other processors the I2C device is implemented in software, by bit-hanging some GPIO pins. The latter approach can consume a significant number of cpu cycles, but is often acceptable because only occasional access to the I2C devices is needed.
eCos Support for I2C

The eCos I2C support for any given platform is spread over a number of different packages:

- **This package, CYGPKG_IO_I2C**, exports a generic API for accessing devices attached to an I2C bus. This API handles issues such as locking between threads. The package does not contain any hardware-specific code. Instead it will use a separate I2C bus driver to handle the hardware, and it defines the interface that such bus drivers should provide. The package only provides support for a bus master, not for acting as a slave device. CYGPKG_IO_I2C also provides the hardware-independent portion of a bit-banged bus implementation. This needs to be complemented by a hardware-specific function that actually manipulates the SDA and SCL lines.

- If the processor has a dedicated I2C device then there will be a bus driver package for that hardware. The processor may be used on many different platforms and the same bus driver can be used on each one. The actual I2C devices attached to the bus will vary from one platform to the next.

- The generic API depends on cyg_i2c_device data structures. These contain the information needed by a bus driver, for example the device address. Usually the data structures are provided by the platform HAL since it is that package which knows about all the devices on the platform.

  On some development boards the I2C lines are brought out to expansion connectors, allowing end users to add extra devices. In such cases the platform HAL may not know about all the devices on the board. Data structures for the additional devices can instead be supplied by application code.

- If the board uses a bit-banged bus then typically the platform HAL will also instantiate the bus instance, providing the function that handles the low-level SDA and SCL manipulation. Usually this code cannot be shared because each board may use different GPIO pins for driving SCL and SDA, so the code belongs in the platform HAL rather than in a separate package.

- Some types of I2C devices may have their own driver package. For example a common type of I2C device is a battery-backed wallclock, and eCos defines how these devices should be supported. Such an I2C device will have its own wallclock device driver and the device will not be accessed directly by application code.

  For other types of device eCos does not define an API and there will not be separate device driver packages. Instead application code is expected to use the cyg_i2c_device structures directly to access the hardware.

Typically all appropriate packages will be loaded automatically when you configure eCos for a given platform. If the application does not use any of the I2C I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the platform, so the platform HAL documentation should be checked for further details.

There is one important exception to this: if the I2C devices are attached to an expansion connector then the platform HAL will not know about these devices. Instead more work will have to be done by application code.
## I2C Interface

### Name

I2C Functions — allow applications and other packages to access I2C devices

### Synopsis

```c
#include <cyg/io/i2c.h>

cyg_uint32 cyg_i2c_tx(const cyg_i2c_device* device, const cyg_uint8* tx_data, cyg_uint32 count);
cyg_uint32 cyg_i2c_rx(const cyg_i2c_device* device, cyg_uint8* rx_data, cyg_uint32 count);
void cyg_i2c_transaction_begin(const cyg_i2c_device* device);
cyg_bool cyg_i2c_transaction_begin_nb(const cyg_i2c_device* device);
cyg_uint32 cyg_i2c_transaction_tx(const cyg_i2c_device* device, cyg_bool send_start, const cyg_uint8* tx_data, cyg_uint32 count, cyg_bool send_stop);
cyg_uint32 cyg_i2c_transaction_rx(const cyg_i2c_device* device, cyg_bool send_start, cyg_uint8* rx_data, cyg_uint32 count, cyg_bool send_nack, cyg_bool send_stop);
void cyg_i2c_transaction_stop(const cyg_i2c_device* device);
void cyg_i2c_transaction_end(const cyg_i2c_device* device);
```

### Description

All I2C functions take a pointer to a cyg_i2c_device structure as their first argument. These structures are usually provided by the platform HAL. They contain the information needed by the I2C bus driver to interact with the device, for example the device address.

An I2C transaction involves the following stages:

1. Perform thread-level locking on the bus. Only one thread at a time is allowed to access an I2C bus. This eliminates the need to worry about locking at the bus driver level. If a platform involves multiple I2C buses then each one will have its own lock.
2. Generate a start condition, send the address and direction bit, and wait for an acknowledgement from the addressed device.
3. Either transmit data to or receive data from the addressed device.
4. The previous two steps may be repeated several times, allowing data to move in both directions during a single transfer.
5. Generate a stop condition, ending the current data transfer. It is now possible to start another data transfer while the bus is still locked, if desired.
6. End the transaction by unlocking the bus, allowing other threads to access other devices on the bus.

The simple functions `cyg_i2c_tx` and `cyg_i2c_rx` perform all these steps in a single call, making them suitable for many I/O operations. The alternative transaction-oriented functions provide greater control when appropriate, for example if a repeated start is necessary for a bi-directional data transfer.

With the exception of `cyg_i2c_transaction_begin_nb` all the functions will block until completion. The tx routines will return 0 if the specified device does not respond to its address, or the number of bytes actually
transferred. This may be less than the number requested if the device sends an early nack, for example because it has run out of buffer space. The rx routines will return 0 or the number of bytes received. Usually this will be the same as the count parameter. A slave device has no way of indicating to the master that no more data is available, so the rx operation cannot complete early.

I2C operations should always be performed at thread-level or during system initialization, and not inside an ISR or DSR. This greatly simplifies locking. Also a typical ISR or DSR should not perform a blocking operation such as an I2C transfer.

Simple Transfers

cyg_i2c_tx and cyg_i2c_rx can be used for simple data transfers. They both go through the following steps: lock the bus, generate the start condition, send the device address and the direction bit, either send or receive the data, generate the stop condition, and unlock the bus. At the end of a transfer the bus is back in its idle state, ready for the next transfer.

I2C transfers return the number of bytes actually transmitted. This may be 0 if the device does not respond when its address is sent out. It may be less than the number of bytes requested if the device generates an early nack, typically because it has run out of buffer space.

cyg_i2c_rx returns 0 if the device does not respond when its address is sent out, or the number of bytes actually received. Typically this will be the number of bytes requested because an I2C slave device has no way of aborting an rx operation early.

Transactions

To allow multiple threads to access devices on the I2C some locking is required. This is encapsulated inside transactions. The cyg_i2c_tx and cyg_i2c_rx functions implicitly use such transactions, but the functionality is also available directly to application code. Amongst other things transactions can be used for more complicated interactions with I2C devices, in particular ones involving repeated starts.

Transactions can be used for more complicated interactions with I2C devices, in particular ones involving repeated starts.

cyg_i2c_transaction_begin must be used at the start of a transaction. This performs thread-level locking on the bus, blocking if it is currently in use by another thread.

cyg_i2c_transaction_begin_nb is a non-blocking variant, useful for threads which cannot afford to block for an indefinite period. If the bus is currently locked the function returns false immediately. If the bus is not locked then it acts as cyg_i2c_transaction_begin and returns true.

Once the bus has been locked it is possible to perform one or more data transfers by calling cyg_i2c_transaction_rx, cyg_i2c_transaction_tx and cyg_i2c_transaction_stop. Code should ensure that a stop condition has been generated by the end of a transaction.

Once the transaction is complete cyg_i2c_transaction_end should be called. This unlocks the bus, allowing other threads to perform I2C I/O to devices on the same bus.

As an example consider reading the registers in an FS6377 programmable clock generator. The first step is to write a byte 0 to the device, setting the current register to 0. Then a repeated start condition should be generated and it is possible to read the 16 byte-wide registers, starting with the current one. Typical code for this might look like:

```c
    cyg_uint8 tx_data[1];
    cyg_uint8 rx_data[16];

    cyg_i2c_transaction_begin(&hal_alaia_i2c_fs6377);
    tx_data[0] = 0x00;
    cyg_i2c_transaction_tx(&hal_alaia_i2c_fs6377,
                       206)
Here `hal_alaia_i2c_fs6377` is a `cyg_i2c_device` structure provided by the platform HAL. A transaction is begun, locking the bus. Then there is a transmit for a single byte. This transmit involves generating a start condition and sending the address and direction bit, but not a stop condition. Next there is a receive for 16 bytes. This also involves a start condition, which the device will interpret as a repeated start because it has not yet seen a stop. The start condition will be followed by the address and direction bit, and then the device will start transmitting the register contents. Once all 16 bytes have been received the rx routine will send a nack rather than an ack, halting the transfer, and then a stop condition is generated. Finally the transaction is ended, unlocking the bus.

The arguments to `cyg_i2c_transaction_tx` are as follows:

- `const cyg_i2c_device* device`  
  This identifies the I2C device that should be used.

- `cyg_bool send_start`  
  If true, generate a start condition and send the address and direction bit. If false, skip those steps and go straight to transmitting the actual data. The latter can be useful if the data to be transmitted is spread over several buffers. The first tx call will involve generating the start condition but subsequent tx calls can skip this and just continue from the previous one.

- `const cyg_uint8* tx_data`
- `cyg_uint32 count`  
  These arguments specify the data to be transmitted to the device.

- `cyg_bool send_stop`  
  If true, generate a stop condition at the end of the transmit. Usually this is done only if the transmit is the last operation in a transaction.

The arguments to `cyg_i2c_transaction_rx` are as follows:

- `const cyg_i2c_device* device`  
  This identifies the I2C device that should be used.

- `cyg_bool send_start`  
  If true, generate a start condition and send the address and direction bit. If false, skip those steps and go straight to receiving the actual data. The latter can be useful if the incoming data should be spread over several buffers. The first rx call will involve generating the start condition but subsequent rx calls can skip this and just continue from the previous one. Another use is for devices which can send variable length data, consisting of an initial length and then the actual data. The first rx will involve generating the start condition and reading the length, a subsequent rx will then just read the data.

- `send_start` must be true if the rx call is the first operation in a transaction, if the previous call was a tx or stop, or if the previous call was an rx and the `send_nack` flag was set.
I2C Interface

```c

cyg_uint8* rx_data
cyg_uint32 count

These arguments specify how much data should be received and where it should be placed.

```c

cyg_bool send_nack

If true generate a nack instead of an ack for the last byte received. This causes the slave to end its transmit. The next operation should either involve a repeated start or a stop. `send_nack` should be set to false only if `send_stop` is also false, the next operation will be another rx, and that rx does not specify `send_start`.

```c

cyg_bool send_stop

If true, generate a stop condition at the end of the transmit. Usually this is done only if the transmit is the last operation in a transaction.

The final transaction-oriented function is `cyg_i2c_transaction_stop`. This just generates a stop condition. It should be used if the previous operation was a tx or rx that, for some reason, did not set the `send_stop` flag. A stop condition must be generated before the transaction is ended.

Initialization

The generic package CYGPKG_IO_I2C arranges for all I2C bus devices to be initialized via a single prioritized C++ static constructor. This constructor will run early on during system startup, before any application code, with priority CYG_INIT_BUS_I2C. Other code should not try to access any of the I2C devices until after the buses have been initialized.
Porting to New Hardware

Name
Porting — Adding I2C support to new hardware

Description
Adding I2C support to an eCos port involves a number of steps. The generic I2C package `CYGPKG_IO_I2C` should be included in the appropriate ecos.db target entry or entries. Next `cyg_i2c_device` structures should be provided for every device on the bus. Usually this is the responsibility of the platform HAL. In the case of development boards where the I2C SDA and SCL lines are accessible via an expansion connector, more devices may have been added and it will be the application’s responsibility to provide the structures. Finally there is a need for one or more `cyg_i2c_bus` structures. Amongst other things these structures provide functions for actually driving the bus. If the processor has dedicated I2C hardware then this structure will usually be provided by a device driver package. If the bus is implemented by bit-banging then the bus structure will usually be provided by the platform HAL.

Adding a Device
The eCos I2C API works in terms of `cyg_i2c_device` structures, and these provide the information needed to access the hardware. A `cyg_i2c_device` structure contains the following fields:

- `cyg_i2c_bus* i2c_bus`
  This specifies the bus which the slave device is connected to. Most boards will only have a single I2C bus, but multiple buses are possible.

- `cyg_uint16 i2c_address`
  For most devices this will be the 7-bit I2C address the device will respond to. There is room for future expansion, for example to support 10-bit addresses.

- `cyg_uint16 i2c_flags`
  This field is not used at present. It exists for future expansion, for example to allow for fast mode or high-speed mode, and incidentally pads the structure to a 32-bit boundary.

- `cyg_uint32 i2c_delay`
  This holds the clock period which should be used when interacting with the device, in nanoseconds. Usually this will be 10000 ns, corresponding to a 100KHz clock, and the header `cyg/io/i2c.h` provides a `#define CYG_I2C_DEFAULT_DELAY` for this. Sometimes it may be desirable to use a slower clock, for example to reduce noise problems.

The normal way to instantiate a `cyg_i2c_device` structure uses the `CYG_I2C_DEVICE` macro, also provided by `cyg/io/i2c.h`:

```c
#include <cyg/io/i2c.h>

CYG_I2C_DEVICE(cyg_i2c_wallclock_ds1307,
               &hal_alia_i2c_bus,
               0x68,
               0x00,
               209
```
Porting to New Hardware

The arguments to the macro are the variable name, an I2C bus pointer, the device address, the flags field, and the delay field. The above code fragment defines two I2C device variables, cyg_i2c_wallclock_ds1307 and hal_alaia_i2c_fs6377, which can be used for the first argument to the cyg_i2c functions. Both devices are on the same bus. The device addresses are 0x68 and 0x58 respectively, and the devices do not have any special requirements.

When the platform HAL provides these structures it should also export them for use by the application and other packages. Usually this involves an entry in cyg/hal/plf_io.h, which gets included automatically via one of the main exported HAL header files cyg/hal/hal_io.h. Unfortunately exporting the structures directly can be problematical because of circular dependencies between the I2C header and the HAL headers. Instead the platform HAL should define a macro HAL_I2C_EXPORTED_DEVICES:

```c
#define HAL_I2C_EXPORTED_DEVICES
extern cyg_i2c_bus hal_alaia_i2c_bus;
extern cyg_i2c_device cyg_i2c_wallclock_ds1307;
extern cyg_i2c_device hal_alaia_i2c_fs6377;
```

This macro gets expanded automatically by cyg/io/i2c.h once the data structures themselves have been defined, so application code can just include that header and all the buses and devices will be properly exported and usable.

There is no single convention for naming the I2C devices. If the device will be used by some other package then typically that specifies the name that should be used. For example the DS1307 wallclock driver expects the I2C device to be called cyg_i2c_wallclock_ds1307, so failing to observe that convention will lead to compile-time and link-time errors. If the device will not be used by any other package then it is up to the platform HAL to select the name, and as long as reasonable care is taken to avoid name space pollution the exact name does not matter.

Bit-banged Bus

Some processors come with dedicated I2C hardware. On other hardware the I2C bus involves simply connecting some GPIO pins to the SCL and SDA lines and then using software to implement the I2C protocol. This is usually referred to as bit-banging the bus. The generic I2C package CYGPKG_IO_I2C provides the main code for a bit-banged implementation, requiring one platform-specific function that does the actual GPIO pin manipulation. This function is usually hardware-specific because different boards will use different pins for the I2C bus, so typically it is left to the platform HAL to provide this function and instantiate the I2C bus object. There is no point in creating a separate package for this because the code cannot be re-used for other platforms.

Instantiating a bit-banged I2C bus requires the following:

```c
#include <cyg/io/i2c.h>

static cyg_bool
hal_alaia_i2c_bitbang(cyg_i2c_bus* bus, cyg_i2c_bitbang_op op)
{
    cyg_bool result = 0;
```
Porting to New Hardware

```c
switch(op) {
    ...

    return result;
}

CYG_I2C_BITBANG_BUS(hal_alaia_i2c_bus, &hal_alaia_i2c_bitbang);
```

This gives a structure `hal_alaia_i2c_bus` which can be used when defining the `cyg_i2c_device` structures. The second argument specifies the function which will do the actual bit-banging. It takes two arguments. The first identifies the bus, which can be useful if the hardware has multiple I2C buses. The second specifies the bit-bang operation that should be performed. To understand these operations consider how I2C devices should be wired up according to the specification:

```
VDD

SDA

SCL

<table>
<thead>
<tr>
<th>SDA in</th>
<th>SCL in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDA out</td>
<td>SDA out</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SDA in</th>
<th>SCL in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDA out</td>
<td>SDA out</td>
</tr>
</tbody>
</table>
```

Master and slave devices are interfaced to the bus in exactly the same way. The default state of the bus is to have both lines high via the pull-up resistors. Any device on the bus can lower either line, when allowed to do so by the protocol. Usually the SDA line only changes while SCL is low, but the start and stop conditions involve SDA changing while SCL is high. All devices have the ability to both read and write both lines. In reality not all bit-banged hardware works quite like this. Instead just two GPIO pins are used, and these are switched between input and output mode as required.

The bitbang function should support the following operations:
Porting to New Hardware

CYG_I2C_BITBANG_INIT

This will be called during system initialization, as a side effect of a prioritized C++ static constructor. The bitbang function should ensure that both SCL and SDA are driven high.

CYG_I2C_BITBANG_SCL_HIGH
CYG_I2C_BITBANG_SCL_LOW
CYG_I2C_BITBANG_SDA_HIGH
CYG_I2C_BITBANG_SDA_LOW

These operations simply set the appropriate lines high or low.

CYG_I2C_BITBANG_SCL_HIGH_CLOCKSTRETCH

In its simplest form this operation should simply set the SCL line high, indicating that the data on the SDA line is stable. However there is a complication: if a device is not ready yet then it can throttle back the master by keeping the SCL line low. This is known as clock-stretching. Hence for this operation the bitbang function should allow the SCL line to float high, then poll it until it really has become high. If a single pin is used for the SCL line then this pin should be turned back into a high output at the end of the call.

CYG_I2C_BITBANG_SCL_LOW_SDA_INPUT

This is used when there is a change of direction and the slave device is about to start driving the SDA line. This can be significant if a single pin is used to handle both input and output of SDA, to avoid a situation where both the master and the slave are driving the SDA line for an extended period of time. The operation combines dropping the SCL line and switching SDA to an input in an atomic or near-atomic operation.

CYG_I2C_BITBANG_SDA_READ

The SDA line is currently set as an input and the bitbang function should sample and return the current state.

The bitbang function returns a boolean. For most operations this return value is ignored. For CYG_I2C_BITBANG_SDA_READ it should be the current level of the SDA line.

Depending on the hardware some care may have to be taken when manipulating the GPIO pins. Although the I2C subsystem performs the required locking at the bus level, the device registers controlling the GPIO pins may get used by other subsystems or by the application. It is the responsibility of the bitbang function to perform appropriate locking, whether via a mutex or by briefly disabling interrupts around the register accesses.

Full Bus Driver

If the processor has dedicated I2C hardware then usually this will involve a separate device driver package in the devs/i2c hierarchy of the eCos component repository. That package should also be included in the appropriate ecos.db target entry or entries. The device driver may exist already, or it may have to be written from scratch.

A new I2C driver basically involves creating an cyg_i2c_bus structure. The device driver should supply the following fields:

i2c_init_fn

This function will be called during system initialization to set up the I2C hardware. The generic I2C code creates a static object with a prioritized constructor, and this constructor will invoke the init functions for the various I2C buses in the system.
Porting to New Hardware

i2c_tx_fn
i2c_rx_fn
i2c_stop_fn

These functions implement the core I2C functionality. The arguments and results are the same as for the transaction functions cyg_i2c_transaction_tx, cyg_i2c_transaction_rx and cyg_i2c_transaction_stop.

void* i2c_extra

This field holds any extra information that may be needed by the device driver. Typically it will be a pointer to some driver-specific data structure.

To assist with instantiating a cyg_i2c_bus object the header file cyg/io/i2c.h provides a macro. Typical usage would be:

```c
struct xyzzy_data {
    ...
} xyzzy_object;

static void
xyzzy_i2c_init(struct cyg_i2c_bus* bus) {
    ...
}

static cyg_uint32
xyzzy_i2c_tx(const cyg_i2c_device* dev, cyg_bool send_start, const cyg_uint8* tx_data, cyg_uint32 count, cyg_bool send_stop) {
    ...
}

static cyg_uint32
xyzzy_i2c_rx(const cyg_i2c_device* dev, cyg_bool send_start, cyg_uint8* rx_data, cyg_uint32 count, cyg_bool send_nack, cyg_bool send_stop) {
    ...
}

static void
xyzzy_i2c_stop(const cyg_i2c_device* dev) {
    ...
}

CYG_I2C_BUS(cyg_i2c_xyzzy_bus, &xyzzy_i2c_init, &xyzzy_i2c_tx, &xyzzy_i2c_rx, &xyzzy_i2c_stop, (void*) &xyzzy_object);
```
Porting to New Hardware

The generic I2C code contains these functions for a bit-banged I2C bus device. It can be used as a starting point for new drivers. Note that the bit-bang code uses the i2c_extra field to hold the hardware-specific bitbang function rather than a pointer to some data structure.
X. CAN Support
Chapter 30. Overview

Description

The Controller Area Network, CAN, is a multicast shared, differential serial bus standard especially suited for networking "intelligent" devices as well as sensors and actuators within a system or sub-system. The protocol was originally developed in the 1980s by Robert Bosch GmbH aiming at automotive applications. Nowadays CAN has gained widespread use and is used in industrial automation as well as in automotive, mobile machines and in many embedded control applications.

The CAN protocol is defined by the ISO 11898-1 standard. The physical layer uses differential transmission on a twisted pair wire. CAN uses a non-destructive bit-wise arbitration to control access to the bus.

There is no explicit address in the messages because in CAN networks there is no addressing of subscribers or stations, but instead, each message carries a prioritized identifier. A transmitter sends a message to all CAN nodes (broadcasting). The identifier may serve as an identification of the contents of the message and also determines the priority that the message enjoys in competition for bus access. A node decides on the basis of this identifier received whether it should process the message or not.

The CAN messages are small (at most eight data bytes) and are protected by a checksum. Each CAN message consists of an 11 bit message ID, up to 8 bytes of data and, a CRC checksum and a number of control bits. These short messages ensure a robust transfer of data in electromagnetically noisy environments. An extended version of the CAN frame supports 29 bit message identifiers.

Basically there are two different operational modes for CAN receivers - FullCAN and BasicCAN. The difference between these two modes is the Object Storage function. The BasicCAN architecture is quite similar to a simple UART. A BasicCAN device has typically one transmit buffer and two receive buffers. The CAN chip handles only the transmitting and receiving of the data (and the error handling) and so most of the manipulation of the data has to be done by the CPU. The CPU has to request the transmitting or acknowledge the receiving of the data through the interrupt flags. This will burden the CPU and take up much of the CPU time.

The FullCAN architecture is more suitable for high-speed performance. It has its own storage area on chip and works with a number of message buffers or message boxes. The CAN controller has its own Acceptance Filtering Mask on chip. It can thus determine which frames are to be received by examining the identifiers. The CPU in this case will only receive the valid (wanted) frames and hence improve the performance of the CPU.

You can find more information at the CAN in Automation (http://www.can-cia.org/) website.

eCos Support for CAN

The eCos CAN subsystem supports the BasicCAN and FullCAN mode. The architecture and the interface of the eCos CAN driver is quite similar to the eCos serial driver and supports the same interface.

The eCos CAN support for any given platform is spread over a number of different packages:

- This package, CYGPKG_IO_CAN, exports a generic device independent CAN I/O API for accessing devices attached to a CAN network. This API handles issues such as locking between threads. The package does not contain any hardware-specific code. Instead it will call into a CAN device driver to handle the hardware device access. This package also defines the interface that such hardware drivers should provide.

- Each CAN device will have its own device driver, which is implemented as a separate package, for example CYGPKG_DEVS_CAN_MCF52xx_FLEXCAN. For devices that may be attached to a variety of different boards the
device driver will be generic and a second platform specific package will be used to customize it to each platform. For devices that are associated with a specific chipset, only a single package may be present. Typically all appropriate packages will be loaded automatically when you configure eCos for a given platform. If the application does not use any of the CAN I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the platform, so the platform HAL documentation should be checked for further details.

There is an important exception to this: if the CAN devices are attached to an expansion connector, such as PCI, then the platform HAL will not know about these devices. Instead the necessary packages will need to be added explicitly during configuration.
The CAN driver uses the standard eCos I/O API functions. All functions except `cyg_io_lookup()` require an I/O "handle".

All functions return a value of the type `Cyg_ErrNo`. If an error condition is detected, this value will be negative and the absolute value indicates the actual error, as specified in `cyg/error/codes.h`. The only other legal return values will be `ENOERR`, `-EINTR` and `-EAGAIN`. All other function arguments are pointers (references). This allows the drivers to pass information efficiently, both into and out of the driver. The most striking example of this is the `len` value passed to the read and write functions. This parameter contains the desired length of data on input to the function and the actual transferred length on return.

```c
// Lookup a CAN device and return its handle
Cyg_ErrNo cyg_io_lookup(
    const char *name,
    cyg_io_handle_t *handle
)
```

This function maps a CAN device name onto an appropriate handle. If the named device is not in the system, then the error `-ENOENT` is returned. If the device is found, then the handle for the device is returned by way of the handle pointer `*handle`.

```c
// Send a CAN message
Cyg_ErrNo cyg_io_write(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len
)
```

This function sends one single CAN message (not a buffer of CAN messages) to a device. The size of data to send is contained in `*len` and the actual size sent will be returned in the same place.

```c
// Read one CAN event from device
Cyg_ErrNo cyg_io_read(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len
)
```

This function receives one single CAN event from a device. The desired size of data to receive is contained in `*len` and the actual size obtained will be returned in the same place.

```c
// Read configuration of a CAN device
Cyg_ErrNo cyg_io_get_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    void *buf,
    cyg_uint32 *len
)
```

This function is used to obtain run-time configuration about a device. The type of information retrieved is specified by the `key`. The data will be returned in the given buffer. The value of `*len` should contain the amount of data requested, which must be at least as large as the size appropriate to the selected key. The actual size of data retrieved is placed in `*len`. The appropriate key values are all listed in the file `<cyg/io/config_keys.h>`.

```c
// Change configuration of a CAN device
Cyg_ErrNo cyg_io_set_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
```
This function is used to manipulate or change the run-time configuration of a device. The type of information is specified by the key. The data will be obtained from the given buffer. The value of *len should contain the amount of data provided, which must match the size appropriate to the selected key. The appropriate key values are all listed in the file `<cyg/io/config_keys.h>`. 
Chapter 32. CAN driver details

Allow applications and other packages to access CAN devices.

Description

A raw CAN driver is provided as a standard part of the eCos system.
Use the include file `<cyg/io/canio.h>` for this driver.

The CAN driver is capable of sending single CAN messages to a device and receiving single CAN events from a CAN device. Controls are provided to configure the actual hardware, but there is no manipulation of the data by this driver.

There may be many instances of this driver in a given system, one for each CAN channel. Each channel corresponds to a physical device and there will typically be a device module created for this purpose. The device modules themselves are configurable, allowing specification of the actual hardware details.

API Details

**cyg_io_write**

cyg_io_write(handle, buf, len)

To transmit a message an application must fill a `cyg_can_message` buffer and call `cyg_io_write()`. This function sends one single CAN message (not a buffer of CAN messages) to a device. The size of data to send is contained in `*len` and the actual size sent will be returned in the same place. A pointer to a `cyg_can_message` is contained in `*buf`. The driver maintains a buffer to hold the data. The size of the intermediate buffer is configurable within the interface module. The data is not modified at all while it is being buffered. On return, `*len` contains the amount of characters actually consumed - that means `*len` always contains `sizeof(cyg_can_message)`.

It is possible to configure the write call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option `CYGOPT_IO_CAN_SUPPORT_NONBLOCKING` to be enabled, and the specific device to be set to non-blocking mode for writes (see `cyg_io_set_config()`). In blocking mode, the call will not return until there is space in the buffer and the content of the CAN message has been consumed. In non-blocking mode, if there is no space in buffer for the CAN message, `-EAGAIN` is returned and the caller must try again.

It is possible to configure the write call to be non-blocking with timeout. Non-blocking mode with timeout requires the configuration option `CYGOPT_IO_CAN_SUPPORT_NONBLOCKING` and `CYGOPT_IO_CAN_SUPPORT_TIMEOUTS` to be enabled, requires the eCos kernel package to be included and the specific device to be set to non-blocking mode for writes (see `cyg_io_set_config()`). In non-blocking mode with timeouts, if there is no space in buffer for the CAN message, the driver waits a certain amount of time (the timeout time) for space in the buffer. If there is still no space in buffer after expiration of the timeout time, `-EINTR` is returned and the caller must try again.

If a message was successfully sent, the function returns `ENOERR`. 
Chapter 32. CAN driver details

CAN Messages

The CAN driver uses cyg_can_message structures to pass messages between the application and the CAN driver. The type cyg_can_message provides a device independent type of CAN message. Before calling the write function this message should be setup properly.

typedef struct can_message
{
    cyg_uint32 id;
    cyg_uint8 data[8];
    cyg_can_id_type ext;
    cyg_can_frame_type rtr;
    cyg_uint8 dlc;
} cyg_can_message;

The structure contains the following fields:

cyg_uint32 id
Message ID. This is the ID to be transmitted with the message, or the ID received. If the ext field is set, then this will contain a 29 bit ID, otherwise it will contain an 11 bit ID.

cyg_uint32 data
Message data. Only the first dlc bytes of data are valid. If the rtr field is set, then the contents of this field are ignored.

cyg_can_id_type ext
Extended ID. If this field is CYGNUM_CAN_ID_EXT then the id field contains a 29 bit extended ID. If it contains CYGNUM_CAN_ID_STD then the ID is 11 bits.

cyg_can_frame_type rtr
Remote Transmission Request. If this field contains CYGNUM_CAN_FRAME_RTR then the RTR bit on the message will be set and the data field will be ignored. If the field contains CYGNUM_CAN_FRAME_DATA then a normal data frame will be send.

cyg_uint8 dlc
The length of the data carried in the message. This can range from zero to 8. In a message with the rtr field set, this indicates the size of data being requested.

Example code for sending one single CAN message:

cyg_can_message tx_msg;
cyg_uint32 len;
Cyg_ErrNo ret;

tx_msg.id = 0x100;
tx_msg.ext = CYGNUM_CAN_ID_EXT;
tx_msg.rtr = CYGNUM_CAN_FRAME_DATA;
tx_msg.dlc = 1;
tx_msg.data[0] = 0xF1;

len = sizeof(tx_msg);
ret = cyg_io_write(hDrvCAN, &tx_msg, &len);
cyg_io_read

cyg_io_read(handle, buf, len)

To receive a message the application calls cyg_can_recv(). This function receives one single event from a device. The desired size of data to receive is contained in *len and the actual size obtained will be returned in the same place. A pointer to a cyg_can_event is contained in *buf. No manipulation of the data is performed before being transferred. Again, this buffering is completely configurable. On return, *len contains sizeof(cyg_can_event).

It is possible to configure the read call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option CYGOPT_IO_CAN_SUPPORT_NONBLOCKING to be enabled, and the specific device to be set to non-blocking mode for reads (see cyg_io_set_config()). In blocking mode, the call will not return until one single CAN event has been read. In non-blocking mode, if there is no CAN event in buffer, the call returns immediately with -EAGAIN and the caller must try again.

It is possible to configure the write call to be non-blocking with timeout. None-blocking mode with timeout requires the configuration option CYGOPT_IO_CAN_SUPPORT_NONBLOCKING and CYGOPT_IO_CAN_SUPPORT_TIMEOUTS to be enabled, requires the eCos kernel package to be included and the specific device to be set to non-blocking mode for reads (see cyg_io_set_config()). In non-blocking mode with timeouts, if there is no CAN event in receive buffer, the driver waits a certain amount of time (the timeout time) for a CAN event to arrive. If there is still no CAN event in buffer after expiration of the timeout time, -EINTR is returned and the caller must try again.

If an event was successfully received, the function returns ENOERR.

CAN Events

The CAN driver uses cyg_can_event structures to pass events from hardware device driver to the generic CAN driver. A cyg_can_event provides a generic device independent type for handling CAN events that may occur.

typedef struct cyg_can_event_st
{
    cyg_uint32 timestamp;
    cyg_can_message msg;
    cyg_uint16 flags;
} cyg_can_event;

The structure contains the following fields:

cyg_uint32 timestamp

If the hardware CAN device driver supports timestamps then this field may contain a timestamp value for an event that occurred.

cyg_can_message msg

CAN message. The msg field contains a CAN message if an RX or TX event occurred. If another type of event occurred, the data field of the msg may contain additional event specific data.

cyg_uint16 flags

Event flags. The flags field contains 16 bits that indicate which kind of events occurred.

The following events are supported and after receiving an event the application should check the flag field against these values:

typedef enum
Chapter 32. CAN driver details

cyg_can_event_flags;

Often the flags field will contain only one single set flag. But it is possible that a number of flags is set and so the flag field should always be checked by a receiver. I.e. if the CYGNUM_CAN_EVENT_RX is set then also the CYGNUM_CAN_EVENT_OVERRUN_RX may be set if the received message caused an RX overrun.

The internal receive buffers of the CAN device driver are circular buffers. That means that even if the buffers are completely filled new messages will be received. In this case the newest message will always overwrite the oldest message in receive buffer. If this happens the CYGNUM_CAN_EVENT_OVERRUN_RX flag will be set for this new message that caused overwriting of the old one. The CYGNUM_CAN_EVENT_OVERRUN_RX flag will be set also if a overrun occurs in hardware message buffers of the CAN device.

Example code for receiving one single CAN event:

cyg_can_event rx_event;
cyg_uint32 len;
Cyg_ErrNo ret;

len = sizeof(rx_event);
ret = cyg_io_read(hDrvCAN, &rx_event, &len);

if (ENOERR == ret)
{
    if (rx_event.flags & CYGNUM_CAN_EVENT_RX)
    {
        // handle RX event
    }

    if (rx_event.flags & ~CYGNUM_CAN_EVENT_RX)
    {
        // handle other events
    }
}
else if (-EINTR == ret)
{
    // handle timeout
}
Chapter 32. CAN driver details

cyg_io_get_config

cyg_io_get_config(handle, key, buf, len)

This function is used to obtain run-time configuration about a device. The type of information retrieved is specified by the key. The data will be returned in the given buffer. The value of *len should contain the amount of data requested, which must be at least as large as the size appropriate to the selected key. The actual size of data retrieved is placed in *len. The appropriate key values are all listed in the file <cyg/io/config_keys.h>.

The following config keys are currently supported:

- CYG_IO_GET_CONFIG_READ_BLOCKING
- CYG_IO_GET_CONFIG_WRITE_BLOCKING
- CYG_IO_GET_CONFIG_CAN_INFO
- CYG_IO_GET_CONFIG_CAN_BUFFER_INFO
- CYG_IO_GET_CONFIG_CAN_MSGBUF_INFO
- CYG_IO_GET_CONFIG_CAN_TIMEOUT
- CYG_IO_GET_CONFIG_CAN_HDI
- CYG_IO_GET_CONFIG_CAN_STATE


cyg_io_set_config

cyg_io_set_config(handle, key, buf, len)

This function is used to manipulate or change the run-time configuration of a device. The type of information is specified by the key. The data will be obtained from the given buffer. The value of *len should contain the amount of data provided, which must match the size appropriate to the selected key. The appropriate key values are all listed in the file <cyg/io/config_keys.h>.

The following config keys are currently supported:

- CYG_IO_SET_CONFIG_READ_BLOCKING
- CYG_IO_SET_CONFIG_WRITE_BLOCKING
- CYG_IO_SET_CONFIG_CAN_INFO
- CYG_IO_SET_CONFIG_CAN_OUTPUT_DRAIN
- CYG_IO_SET_CONFIG_CAN_OUTPUT_FLUSH
- CYG_IO_SET_CONFIG_CAN_INPUT_FLUSH
- CYG_IO_SET_CONFIG_CAN_TIMEOUT
- CYG_IO_SET_CONFIG_CAN_MSGBUF
- CYG_IO_SET_CONFIG_CAN_MODE
- CYG_IO_SET_CONFIG_CAN_ABORT
- CYG_IO_SET_CONFIG_CAN_CALLBACK

Runtime Configuration

Runtime configuration is achieved by exchanging data structures with the driver via the cyg_io_set_config() and cyg_io_get_config() functions.

Device configuration

typedef struct cyg_can_info_st {
    cyg_can_baud_rate_t baud;
} cyg_can_info_t;
Device configuration is achieved by exchanging `cyg_can_info_t` data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions using the config keys `CYG_IO_GET_CONFIG_CAN_INFO` and `CYG_IO_SET_CONFIG_CAN_INFO`. The field `baud` contains a baud rate selection. This must be one of the following values:

- `CYGNUM_CAN_KBAUD_10`
- `CYGNUM_CAN_KBAUD_20`
- `CYGNUM_CAN_KBAUD_50`
- `CYGNUM_CAN_KBAUD_100`
- `CYGNUM_CAN_KBAUD_125`
- `CYGNUM_CAN_KBAUD_250`
- `CYGNUM_CAN_KBAUD_500`
- `CYGNUM_CAN_KBAUD_800`
- `CYGNUM_CAN_KBAUD_1000`

### Timeout configuration

```c
typedef struct cyg_can_timeout_info_st
{
    cyg_uint32 rx_timeout;
    cyg_uint32 tx_timeout;
} cyg_can_timeout_info_t;
```

Timeout configuration is achieved by exchanging `cyg_can_timeout_info_t` data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions using the config keys `CYG_IO_SET_CONFIG_CAN_TIMEOUT` and `CYG_IO_SET_CONFIG_CAN_TIMEOUT`.

- `cyg_uint32 rx_timeout`
  - Timeout for `cyg_io_read` calls.
- `cyg_uint32 tx_timeout`
  - Timeout for `cyg_io_write` calls.

Timeout runtime configuration is supported if the configuration options `CYGOPT_IO_CAN_SUPPORT_NONBLOCKING` and `CYGOPT_IO_CAN_SUPPORT_TIMEOUTS` are enabled.

### Reading buffer configuration

```c
typedef struct cyg_can_buf_info_st
{
    cyg_int32 rx_bufsize;
    cyg_int32 rx_count;
    cyg_int32 tx_bufsize;
    cyg_int32 tx_count;
} cyg_can_buf_info_t;
```

`CYG_IO_GET_CONFIG_CAN_BUFFER_INFO` - This function retrieves the current state of the software buffers in the CAN drivers. For the transmit buffer it returns the total number of `cyg_can_message` objects in buffer and the current number of `cyg_can_message` objects occupied in the buffer. For the receive buffer it returns the total number of `cyg_can_event` objects in receive buffer and the current number of `cyg_can_event` objects occupied in the buffer. It does not take into account any buffering such as FIFOs or holding registers that the CAN hardware device itself may have.
Chapter 32. CAN driver details

cyg_uint32 rx_bufsize
Total number of cyg_can_event buffers in receive queue.

cyg_uint32 rx_count
Current number of cyg_can_event buffers occupied in receive queue.

cyg_uint32 tx_bufsize
Total number of cyg_can_message buffers in transmit queue.

cyg_uint32 rtx_count
Current number of cyg_can_message buffers occupied in transmit queue.

Reading hardware description information

typedef struct cyg_can_hdi_st
{
  cyg_uint8 support_flags;
  cyg_uint8 controller_type;
} cyg_can_hdi;

CYG_IO_GET_CONFIG_CAN_HDI - This function retrieves information about the used hardware. The Hardware Description Interface provides a method to gather information about the CAN hardware and the functionality of the driver. For this purpose the structure cyg_can_hdi is defined.

cyg_uint8 support_flags
Contains information about the capabilities of the used CAN hardware.

cyg_uint8 controller_type
A number that identifies the CAN controller type.

The following flags are available in the field support_flags:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+-------------------------------+-------------------------------+
| res | res | res |timest.|SW-Filt|FullCAN| Frametype |

Frametype
Bit 0 and Bit 1 of the structure describe the possibilities of the CAN controller. The following values are defined:

CYGNUM_CAN_HDI_FRAMETYPE_STD // receives only standard frame
CYGNUM_CAN_HDI_FRAMETYPE_EXT_PASSIVE // can receive but not send extended frames
CYGNUM_CAN_HDI_FRAMETYPE_EXT_ACTIVE // can send and receive extended frames

FullCAN
If the Bit 2 - CYGNUM_CAN_HDI_FULLCAN - is set to one, the CAN controller supports more than one message buffer.
Chapter 32. CAN driver details

**SW-Filter**

If Bit 3 - CYGNUM_CAN_HDI_FILT_SW - is set to one then the CAN driver supports some kind of software message filtering.

**Timestamp**

If Bit 4 - CYGNUM_CAN_HDI_TIMESTAMP - is set to one then the CAN hardware supports timestamps for CAN messages

### Reading hardware message buffer configuration

```c
typedef struct cyg_can_msgbuf_info_st {
    cyg_uint8 count; // number of message buffers available for this device
    cyg_uint8 free; // number of free message buffers
} cyg_can_msgbuf_info;
```

CYG_IO_GET_CONFIG_CAN_MSGBUF_INFO - If the CAN hardware supports more than one message buffer for reception of CAN messages (flag CYGNUM_CAN_HDI_FULLCAN is set while reading hardware description interface with CYG_IO_GET_CONFIG_CAN_HDI) then this function reads the number of message buffers the CAN hardware supports and the number of free message buffers.

- **cyg_uint8 count**
  - Counts the number of message buffers supported by the device.

- **cyg_uint8 free**
  - Contains the number of free message buffers. The free message buffers are available for setting up remote buffers (CYG_IO_SET_CONFIG_CAN_REMOTE_BUF) and message filters (CYG_IO_SET_CONFIG_CAN_FILTER_MSG).

### Reading state of CAN hardware

```c
typedef enum {
    CYGNUM_CAN_STATE_ACTIVE, // CAN controller active, no errors
    CYGNUM_CAN_STATE_STOPPED, // CAN controller in stopped mode
    CYGNUM_CAN_STATE_STANDBY, // CAN controller in Sleep mode
    CYGNUM_CAN_STATE_BUS_WARN, // CAN controller active, warning level is reached
    CYGNUM_CAN_STATE_ERR_PASSIVE, // CAN controller went into error passive mode
    CYGNUM_CAN_STATE_BUS_OFF, // CAN controller went into bus off mode
    CYGNUM_CAN_STATE_PHY_FAULT, // General failure of physical layer
    CYGNUM_CAN_STATE_PHY_H, // Fault on CAN-H detected (Low Speed CAN)
    CYGNUM_CAN_STATE_PHY_L, // Fault on CAN-L detected (Low Speed CAN)
} cyg_can_state;
```

CYG_IO_GET_CONFIG_CAN_STATE - This function retrieves the present state of the CAN controller. Possible values are defined in the cyg_can_state enumeration.
Changing mode of CAN hardware

CYG_IO_SET_CONFIG_CAN_MODE - This function changes the operating mode of the CAN controller. The identifiers for the different operating modes are defined in the cyg_can_mode enumeration.

typedef enum
{
    CYGNUM_CAN_MODE_STOP,  // set controller into stop mode
    CYGNUM_CAN_MODE_START, // set controller into operational mode
    CYGNUM_CAN_MODE_STANDBY // set controller into standby / sleep mode
} cyg_can_mode;

CYGNUM_CAN_MODE_STOP

Set controller into stop mode

CYGNUM_CAN_MODE_START

Set controller into operational mode

CYGNUM_CAN_MODE_STANDBY

Set controller into standby / sleep mode.

Before the hardware configuration of the device is changed, that means if baud rate is changed or the message buffer and filter configuration is changed, the CAN hardware should be set into stop mode and if configuration is finished, then device should be set back into operational mode. Before the device is set into standby mode, the output buffers should be flushed or drained because transmission of a CAN message may wake up the CAN hardware. If a received message wakes up the CAN hardware from standby mode then a CYGNUM_CAN_EVENT_LEAVING_STANDBY event will be inserted into receive message buffer or the CYGNUM_CAN_EVENT_LEAVING_STANDBY flag will be set for the message that caused wake up of CAN hardware.

Flush or drain buffers

CYG_IO_SET_CONFIG_CAN_OUTPUT_DRAIN - This function waits for any buffered output to complete. This function only completes when there is no more data remaining to be sent to the device.

CYG_IO_SET_CONFIG_CAN_OUTPUT_FLUSH - This function discards any buffered output for the device.

CYG_IO_SET_CONFIG_CAN_INPUT_FLUSH - This function discards any buffered input for the device.

Configuring blocking/non-blocking calls

By default all calls to cyg_io_read() and cyg_io_write() are blocking calls. The config keys

CYG_IO_SET_CONFIG_READ_BLOCKING
CYG_IO_SET_CONFIG_WRITE_BLOCKING

enable switching between blocking and nonblocking calls separately for read and write calls. If blocking calls are configured then the read/write functions return only if a message was stored into TX buffer or a event was received from RX buffer. If non-blocking calls are enabled and there is no space in TX buffer or RX buffer is empty then the function returns immediately with -EAGAIN.
Chapter 32. CAN driver details

If non-blocking calls are enabled and additionally timeouts are supported by driver, then the read/write functions wait until timeout value is expired and then return with -EINTR. If the read/write operation succeeds during the timed wait then the functions return successfully with ENOERR.

To query if cyg_io_read() and cyg_io_write() are blocking or non-blocking you can use the config keys:

CYG_IO_GET_CONFIG_READ_BLOCKING
CYG_IO_GET_CONFIG_WRITE_BLOCKING

Message buffer management

Full CAN controllers often support more than one message buffer. These message buffers are often configurable for transmission or reception of certain CAN messages or as a remote buffers. If a CAN hardware supports more than one message buffer then it is possible to configure the CAN hardware to receive only CAN messages with certain identifiers or to configure hardware support for remote buffers. If message filtering is done by hardware, the number of received CAN messages decreases and so also the time for processing received CAN messages and the memory required for buffering received messages decreases. This saves valuable memory and processing time.

The eCos CAN driver supports a generic way of adding message filters or remote buffers. By default the CAN driver is configured for reception of any kind of CAN standard and extended frames. Configuration of message buffers is done by calling cyg_io_set_config() with the config key:

CYG_IO_SET_CONFIG_CAN_MSGBUF

and by exchanging cyg_can_msgbuf_cfg data structures.

typedef struct cyg_can_msgbuf_cfg_st
{
   cyg_can_msgbuf_cfg_id cfg_id; // configuration id
   cyg_can_msgbuf_handle handle; // handle to message buffer
   cyg_can_message msg; // CAN message - for configuration of buffer
} cyg_can_msgbuf_cfg;

cyg_can_msgbuf_cfg_id cfg_id

The cfg_id field contains the configuration ID that tells the driver what to do with a message buffer.

cyg_can_msgbuf_handle handle

Contains a reference to a certain message buffer.

cyg_can_message msg

Required for configuration of message buffer parameters.

The following configuration identifiers are supported:

CYGNUM_CAN_MSGBUF_RESET_ALL // clears all message buffers
CYGNUM_CAN_MSGBUF_RX_FILTER_ALL // cfg driver for reception of all can messages
CYGNUM_CAN_MSGBUF_RX_FILTER_ADD // add single message filter
CYGNUM_CAN_MSGBUF_REMOTE_BUF_ADD // add new remote response buffer
CYGNUM_CAN_MSGBUF_REMOTE_BUF_WRITE // stores data into existing remote buffer

CYGNUM_CAN_MSGBUF_RESET_ALL

Clears all message buffers - no message will be received and all remote buffers are deleted.
Chapter 32. CAN driver details

**CYGNUM_CAN_MSGBUF_RX_FILTER_ALL**

Configure driver for reception of all can messages

**CYGNUM_CAN_MSGBUF_RX_FILTER_ADD**

Add single message filter.

**CYGNUM_CAN_MSGBUF_REMOTE_BUF_ADD**

Add new remote response buffer.

**CYGNUM_CAN_MSGBUF_REMOTE_BUF_WRITE**

Stores data into existing remote buffer (remote buffer handle required).

Example code for resetting all message buffers:

```c
#cyg_can_msgbuf_cfg msgbox_cfg;
msgbox_cfg.cfg_id = CYGNUM_CAN_MSGBUF_RESET_ALL;
len = sizeof(msgbox_cfg);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN,
CYG_IO_SET_CONFIG_CAN_MSGBUF,
&msgbox_cfg, &len))
{
    // handle configuration error
}
```

**Remote frame response buffer configuration**

The remote frame is a message frame which is transmitted to request a data frame. Some CAN hardware generates receive interrupts when a remote transmission request arrives. Other CAN hardware, i.e. the Motorola FlexCAN module, does not generate any receive interrupt. These CAN hardware chips like the FlexCAN module can be configured to transmit a data frame automatically in response to a remote frame. In order to support any kind of CAN hardware the eCos CAN driver provides a generic handling of remote transmission requests.

The transmission of the data frame in response to a remote frame is completely handled by the CAN driver. If the hardware driver, like the driver for the FlexCAN module, supports hardware message buffers, then the response frame is automatically transmitted if a remote transmission request with a matching ID arrives. If a CAN hardware does not provide hardware support for sending data frames in response to a remote frame, then this need to be implemented in software by the hardware device driver.

It is always possible to add remote response buffers. It does not matter if the driver is configured for reception of all CAN messages or if message filtering is used. As long as there are free message buffers available, it is possible to add remote response buffers.

In order to respond to a remote frame, a remote frame response buffer need to be initialized before a data frame can be sent in response to a remote frame. This is achieved by by exchanging `cyg_can_remote_buf` data structures with the driver via the `cyg_io_set_config()` function using the config key `CYG_IO_SET_CONFIG_CAN_MSGBUF`. Once the buffer is initialized, the CAN data can be changed at any time by the application.

```c
typedef struct cyg_can_msgbuf_cfg_st
{
    cyg_can_msgbuf_cfg_id cfg_id; // configuration id
    cyg_can_msgbuf_handle handle; // handle to message buffer
    cyg_can_message msg; // CAN message - for configuration of buffer
}...
```
Chapter 32. CAN driver details

```c
} cyg_can_remote_buf;

cyg_can_msgbuf_cfg_id cfg_id

The cfg_id field contains the configuration ID that tells the driver what to do with a message buffer
(CYGNUM_CAN_MSGBUF_REMOTE_BUF_ADD or CYGNUM_CAN_MSGBUF_REMOTE_BUF_WRITE).

cyg_can_msgbuf_handle handle

If there is no buffer initialized for this data, the value of the handle field need to be set to
CYGNUM_CAN_MSGBUF_INIT. After the call to cyg_io_set_config() the handle field contains a valid
remote buffer handle (>= 0) or the value CYGNUM_CAN_MSGBUF_NA (< 0) if no free buffer is available.

cyg_can_message msg

The CAN frame that should be transmitted in response to a remote frame.

Example code for setting up a remote response buffer:

cyg_can_remote_buf rtr_buf;

// prepare the remote response buffer
rtr_buf.cfg_id = CYGNUM_CAN_MSGBUF_REMOTE_BUF_ADD;
rtr_buf.handle = CYGNUM_CAN_MSGBUF_INIT;
rtr_buf.msg.id = 0x7FF;
rtr_buf.msg.ext = CYGNUM_CAN_ID_STD;
rtr_buf.msg.rtr = CYGNUM_CAN_FRAME_DATA;
rtr_buf.msg.dlc = 1;
rtr_buf.msg.data[0] = 0xAB;

len = sizeof(rtr_buf);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN,
       CYG_IO_SET_CONFIG_CAN_MSGBUF,
       &rtr_buf, &len))
{
    // handle configuration error
}

if (rtr_buf.handle == CYGNUM_CAN_MSGBUF_NA)
{
    // no free message buffer available - handle this problem here
}

// change CAN data for a buffer that is already initialized
rtr_buf.cfg_id = CYGNUM_CAN_MSGBUF_REMOTE_BUF_WRITE;
rtr_buf.msg.data[0] = 0x11;

len = sizeof(rtr_buf);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN,
       CYG_IO_SET_CONFIG_CAN_MSGBUF,
       &rtr_buf, &len))
{
    // handle configuration error
}
```

232
Message filter configuration

If message filtering is done by hardware the number of received CAN messages decreases and so also the time for processing received CAN messages and the memory required for buffering received messages decreases. This saves valuable memory and processing time. The eCos CAN driver supports a generic way of adding message filters. By default the CAN driver is configured for reception of any kind of CAN standard and extended frames. As soon as a message filter is added, the CAN driver will only receive the CAN frames with the identifier of the CAN filter. By adding a number of message filters it is possible for the CAN hardware to receive an number of different CAN messages.

Adding message filters is only possible if driver is not configured for reception of all available CAN messages. If the driver is configured for reception of all CAN messages then message buffers need to be reset before adding single message filters.

In order to add a message filter, a message buffer need to be initialized. This is achieved by exchanging cyg_can_filter data structures with the driver via the cyg_io_set_config() function using the config key CYG_IO_SET_CONFIG_CAN_MSGBUF. Once the buffer is initialized, the CAN hardware can receive messages with the identifier of the filter.

```
typedef struct cyg_can_msgbox_cfg_st {
    cyg_can_msgbuf_cfg_id cfg_id;
    cyg_can_msgbuf_handle handle;
    cyg_can_message msg;
} cyg_can_filter;
```

cyg_can_msgbuf_cfg_id cfg_id

The cfg_id field contains the configuration ID that tells the driver what to do with a message buffer.

cyg_can_msgbuf_handle handle

After the call to cyg_io_set_config() the handle field contains a valid value ( >= 0) or the value CYGNUM_CAN_MSGBUF_NA (< 0) if no free buffer is available.

cyg_can_message msg

The fields id and ext of the msg configure the type of message to receive by a certain message filter.

Example code for setting up a message filter:

```
cyg_can_msgbuf_cfg msgbox_cfg;
cyg_can_filter rx_filter;

// reset all message buffers
msgbox_cfg.cfg_id = CYGNUM_CAN_MSGBUF_RESET_ALL;
len = sizeof(msgbox_cfg);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN, CYG_IO_SET_CONFIG_CAN_MSGBUF, &msgbox_cfg, &len))
{
    // handle configuration error
}

// prepare the message filter
```
Chapter 32. CAN driver details

rx_filter.cfg_id = CYGNUM_CAN_MSGBUF_RX_FILTER_ADD
rx_filter.msg.id = 0x800;
rx_filter.msg.ext = CYGNUM_CAN_ID_EXT;

len = sizeof(rx_filter);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN,
    CYG_IO_SET_CONFIG_CAN_MSGBUF,
    &rx_filter, &len))
{
    // handle configuration error;
}
else if (CYGNUM_CAN_MSGBUF_NA == rx_filter.handle)
{
    // no free message buffer available - handle this problem here
}

Message filter deactivation

After startup of your device the CAN driver is configured for reception of all available CAN messages. If you change this configuration by adding single message filters then you can reset this default state with the configuration ID:

CYGNUM_CAN_MSGBUF_RX_FILTER_ALL

This message buffer configuration id will clear all message filters and remote buffers and prepares the CAN hardware for reception of any kind of CAN standard and extended frames. It is not necessary to reset the message buffer configuration before this configuration step is executed because this should be done by device driver.

Example code for deactivation of message filtering:

cyg_can_filter rx_filter;

// now setup a RX all configuration
rx_filter.cfg_id = CYGNUM_CAN_MSGBUF_RX_FILTER_ALL;
len = sizeof(rx_filter);
if (ENOERR != cyg_io_set_config(hDrvFlexCAN,
    CYG_IO_SET_CONFIG_CAN_MSGBUF,
    &rx_filter, &len))
{
    CYG_TEST_FAIL_FINISH("Error writing config of /dev/can0");
}

Configuring a callback on events

By default application cannot get information about an event arriving in the RX buffer until it calls the cyg_io_read(). Usually this leads applications to use accessory threads to wait for new CAN events.

The CDL option CYGOPT_IO_CAN_SUPPORT_CALLBACK allows application to use a callback on event arrival. It is configured by passing a cyg_can_callback_cfg data structure to the driver via the cyg_io_set_config() function using the config key CYG_IO_SET_CONFIG_CAN_CALLBACK.
typedef void (*cyg_can_event_cb_t)(cyg_uint16, CYG_ADDRWORD);

typedef struct cyg_can_callback_cfg_st
{
    cyg_can_event_cb_t callback_func; // callback function
    cyg_uint16 flag_mask; // flags mask
    CYG_ADDRWORD data; // data passed to callback
} cyg_can_callback_cfg;

cyg_can_event_cb_t callback_func
   Pointer to the callback function. The function will be called from DSR context so you should be careful to
   only call API functions that are safe in DSR context. The First parameter is a combination of event flags
   for events that have occurred. Second parameter is a user defined data pointer or value.

CYG_ADDRWORD data
   Additional user data that will be passed to callback function as a second parameter.

cyg_uint16 flag_mask
   Should be set with a combination of CYGNUM_CAN_EVENT_* flags. If one of these events happens, the
   callback function will be called, with the actually event flags passed as a parameter. To disable the callback
   function from being called set flag_mask to 0.
Chapter 32. CAN driver details
Chapter 33. Configuration

The CAN subsystem has a number of configuration options.

cdl_interface CYGINT_IO_CAN_TIMESTAMP

A hardware device driver that supports timestamps should implement this interface.

cdl_option CYGOPT_IO_CAN_SUPPORT_TIMESTAMP

If the CAN hardware driver supports some kind of timestamps then this option enables propagation of timestamps to higher layers. This may add some extra code to hardware drivers.

cdl_option CYGOPT_IO_CAN_TX_EVENT_SUPPORT

This option enables support for TX events. If a CAN message is transmitted successfully a TX event will be inserted into the receive event queue and propagated to higher layers. If this option is enabled the RX event queue will be filled faster.

cdl_option CYGOPT_IO_CAN_SUPPORT_NONBLOCKING

This option enables extra code in the generic CAN driver which allows clients to switch read() and write() call semantics from blocking to non-blocking.

cdl_option CYGOPT_IO_CAN_SUPPORT_CALLBACK

This option enables extra code in the generic CAN driver which allows an application to register a callback for events. The callback function is called from DSR context so you should be careful to only call API functions that are safe in DSR context.

cdl_option CYGNUM_IO_CAN_DEFAULT_TIMEOUT_READ

The initial timeout value in clock ticks for cyg_io_read() calls.

cdl_option CYGNUM_IO_CAN_DEFAULT_TIMEOUT_WRITE

The initial timeout value in clock ticks for cyg_io_write() calls.
Chapter 34. Writing a CAN hardware device driver

A CAN driver is nothing more than a named entity that supports the basic I/O functions - read, write, get config, and set config. The device driver uses and manages interrupts from the device. While the interface is generic and device driver independent, the actual driver implementation is completely up to the device driver designer.

That said, the reason for using a device driver is to provide access to a CAN device from application code in as general purpose a fashion as reasonable. Most driver writers are also concerned with making this access as simple as possible while being as efficient as possible.

Like other device drivers the CAN device driver is concerned with the movement of information - the CAN messages. In order to make the most efficient use of system resources, interrupts are used. This will allow other application processing to take place while the data transfers are under way, with interrupts used to indicate when various events have occurred. For example, a CAN device typically generates an interrupt after a CAN message has been sent or a CAN message has been received by a CAN hardware message buffer. It makes sense to allow further application processing while the data is being sent since this can take quite a long time. The interrupt can be used to allow the driver to send a CAN message as soon as the current one is complete, without any active participation by the application code.

The main building blocks for CAN device drivers are found in the include files `<cyg/io/devtab.h>` and `<cyg/io/can.h>`.

Like many other device drivers in eCos, CAN device drivers are described by a device table entry, using the `cyg_devtab_entry_t` type. The entry should be created using the `DEVTAB_ENTRY()` macro.

How to Write a CAN Hardware Interface Driver

The standard CAN driver supplied with eCos is structured as a hardware independent portion and a hardware dependent interface module. To add support for a new CAN device, the user should be able to use the existing hardware independent portion and just add their own interface driver which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the CAN driver and CAN implementation modules are contained in the file `<cyg/io/can.h>`.

DevTab Entry

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```c
DEVTAB_ENTRY(<<module_name>>,
             <<device_name>>,
             0,
             &can_devio,
             <<module_init>>,  
             <<module_lookup>>,
             &<<can_channel>>
         );
```
Chapter 34. Writing a CAN hardware device driver

Arguments

module_name

The "C" label for this devtab entry

device_name

The "C" string for the device. E.g. /dev/can0.

can_devio

The table of I/O functions. This set is defined in the hardware independent CAN driver and should be used.

module_init

The hardware module initialization function.

module_lookup

The device lookup function. This function typically sets up the CAN device for actual use, turning on interrupts, configuring the message buffers, etc.

can_channel

This table (defined below) contains the interface between the interface module and the CAN driver proper.

Example devtab entry for Motorola FlexCAN device driver:

DEVTAB_ENTRY(flexcan_devtab,
CYGDAT_DEVS_CAN_MCF52xx_FLEXCAN0_NAME,
0, // Does not depend on a lower level interface
&cyg_io_can_devio,
flexcan_init,
flexcan_lookup, // CAN driver may need initializing
&flexcan_can0_chan
);

CAN Channel Structure

Each CAN device must have a “CAN channel”. This is a set of data which describes all operations on the device. It also contains buffers, etc. The CAN channel is created by the macro:

CAN_CHANNEL_USING_INTERRUPTS(l, funs, dev_priv, baud,
out_buf, out_buflen,
in_buf, in_buflen)

Arguments

l

The "C" label for this structure.

funs

The set of interface functions (see below).
dev_priv

A placeholder for any device specific data for this channel.

baud

The initial baud rate value (cyg_can_baud_rate_t).

out_buf

Pointer to the output buffer

out buflen

The length of the output buffer.

in_buf

pointer to the input buffer.

in buflen

The length of the input buffer.

Example CAN channel implementation for Motorola FlexCAN device driver:

CAN_CHANNEL_USING_INTERRUPTS(
    flexcan_can0_chan,
    flexcan_lowlevel_funs,
    flexcan_can0_info,
    CYG_CAN_BAUD_RATE(CYGNUM_DEVS_CAN_MCF52xx_FLEXCAN0_KBAUD),
    flexcan_can0_txbuf, CYGNUM_DEVS_CAN_MCF52xx_FLEXCAN0_QUEUESIZE_TX,
    flexcan_can0_rxbuf, CYGNUM_DEVS_CAN_MCF52xx_FLEXCAN0_QUEUESIZE_RX
);

The interface from the hardware independent driver into the hardware interface module is contained in the
funs table. This is defined by the macro:

CAN Lowlevel Functions Structure

CAN_LOWLEVEL_FUNS(l, putmsg, getevent, get_config, set_config, start_xmit, stop_xmit)

Arguments

l

The "C" label for this structure.

putmsg

bool (*putmsg)(can_channel *priv, cyg_can_message *pmsg, void *pdata)

This function sends one CAN message to the interface. It should return true if the message is actually
consumed. It should return false if there is no space in the interface

ggetevent

bool (*getevent)(can_channel *priv, cyg_can_event *pevent, void *pdata)

This function fetches one event from the interface.
Chapter 34. Writing a CAN hardware device driver

get_config

Cyg_ErrNo (*get_config)(can_channel *priv, cyg_uint32 key, const void *xbuf, 
cyg_uint32 *len)

This function is used to query the configuration of a CAN channel.

set_config

Cyg_ErrNo (*set_config)(can_channel *priv, cyg_uint32 key, const void *xbuf, 
cyg_uint32 *len)

This function is used to change configuration of a CAN channel.

start_xmit

void (*start_xmit)(can_channel *priv)

Enable the transmit channel and turn on transmit interrupts.

stop_xmit

void (*stop_xmit)(can_channel *priv)

Disable the transmit channel and turn transmit interrupts off.

Example implementation of low level function structure for Motorola FlexCAN device driver:

```c
CAN_LOWLEVEL_FUNS(flexcan_lowlevel_funs, 
    flexcan_putmsg, 
    flexcan_getevent, 
    flexcan_get_config, 
    flexcan_set_config, 
    flexcan_start_xmit, 
    flexcan_stop_xmit
)
```

Callbacks

The device interface module can execute functions in the hardware independent driver via chan->callbacks. These functions are available:

void (*can_init)(can_channel *chan)

This function is used to initialize the CAN channel.

void (*xmt_msg)(can_channel *chan, void *pdata)

This function would be called from an interrupt handler after a transmit interrupt indicating that additional messages may be sent. The upper driver will call the putmsg function as appropriate to send more data to the device.

void (*rcv_event)(can_channel *chan, void *pdata)

This function is used to tell the driver that a message has arrived at the interface or that an event has occurred. This function is typically called from the interrupt handler.
XI. ADC Support
Overview

Name
Overview — eCos Support for Analog/Digital Converters

Introduction
ADC support in eCos is based around the standard character device interface. Hence all device IO function, or file IO functions may be used to access ADC devices.

ADC devices are presented as read-only serial channels that generate samples at a given rate. The size of each sample is hardware specific and is defined by the cyg_adc_sample_t type. The sample rate may be set at runtime by the application. Most ADC devices support several channels which are all sampled at the same rate. Therefore setting the rate for one channel will usually change the rate for all channels on that device.

Examples
The use of the ADC devices is best shown by example. The following is a simple example of using the eCos device interface to access the ADC:

```c
int res;
cyg_io_handle_t handle;

// Get a handle for ADC device 0 channel 0
res = cyg_io_lookup( "./dev/adc00", &handle );
if( res != ENOERR )
    handle_error(err);

for(;;)
{
    cyg_adc_sample_t sample;
    cyg_uint32 len = sizeof(sample);

    // read a sample from the channel
    res = cyg_io_read(handle, &sample, &len );
    if( res != ENOERR )
        handle_error(err);

    use_sample( sample );
}
```

In this example, the required channel is looked up and a handle on it acquired. Conventionally ADC devices are named "/dev/adcXY" where X is the device number and Y the channel within that device. Following this, samples are read from the device sequentially.

ADC devices may also be accessed using FILEIO operations. These allow more sophisticated usage. The following example shows select() being used to gather samples from several devices.

```c
int fd1, fd2;

// open channels, non-blocking
fd1 = open( "/dev/adc01", O_RDONLY|O_NONBLOCK );
```
Overview

fd2 = open( "/dev/adc02", O_RDONLY|O_NONBLOCK );

if( fd1 < 0 || fd2 < 0 )
    handle_error( errno );

for(;;)
{
    fd_set rd;
    int maxfd = 0;
    int err;
    cyg_adc_sample_t samples[128];
    int len;

    FD_ZERO( &rd );
    FD_SET( fd1, &rd );
    FD_SET( fd2, &rd );
    maxfd = max(fd1,fd2);

    // select on available data on each channel.
    err = select( maxfd+1, &rd, NULL, NULL, NULL );

    if( err < 0 )
        handle_error(errno);

    // If channel 1 has data, handle it
    if( FD_ISSET( fd1, &rd ) )
    {
        len = read( fd1, &samples, sizeof(samples) );

        if( len > 0 )
            handle_samples_chan1( &samples, len/sizeof(sample[0]) );
    }

    // If channel 2 has data, handle it
    if( FD_ISSET( fd2, &rd ) )
    {
        len = read( fd2, &samples, sizeof(samples) );

        if( len > 0 )
            handle_samples_chan2( &samples, len/sizeof(sample[0]) );
    }
}

This test uses FILEIO operations to access ADC channels. It starts by opening two channels for reading only and with blocking disabled. It then falls into a loop using select to wake up whenever either channel has samples available.

Details

As indicated, the main interface to ADC devices is via the standard character device interface. However, there are a number of aspects that are ADC specific.
Sample Type

Samples can vary in size depending on the underlying hardware and is often a non-standard number of bits. The actual number of bits is defined by the hardware driver package, and the generic ADC package uses this to define a type `cyg_adc_sample_t` which can contain at least the required number of bits. All reads from an ADC channel should be expressed in multiples of this type, and actual bytes read will also always be a multiple.

Sample Rate

The sample rate of an ADC device can be varied by calling a `set_config` function, either at the device IO API level or at the FILEIO level. The following two functions show how this is done at each:

```c
int set_rate_io( cyg_io_handle_t handle, int rate )
{
    cyg_adc_info_t info;
    cyg_uint32 len = sizeof(info);

    info.rate = rate;
    return cyg_io_set_config( handle,
                            CYG_IO_SET_CONFIG_ADC_RATE,
                            &info,
                            &len);
}

int set_rate_fileio( int fd, int rate )
{
    cyg_adc_info_t info;

    info.rate = rate;
    return cyg_fs_fsetinfo( fd,
                           CYG_IO_SET_CONFIG_ADC_RATE,
                           &info,
                           sizeof(info) );
}
```

Enabling a Channel

Channels are initialized in a disabled state and generate no samples. When a channel is first looked up or opened, then it is automatically enabled and samples start to accumulate. A channel may then be disable or re-enabled via a `set_config` function:

```c
int disable_io( cyg_io_handle_t handle )
{
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_DISABLE,
                              NULL,
                              NULL);
}

int enable_io( cyg_io_handle_t handle )
{
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_DISABLE,
                              NULL,
                              NULL);
}
```
Configuration

The ADC package defines a number of generic configuration options that apply to all ADC implementations:

cdl_component CYGPKG_IO_ADC_DEVICES
  This option enables the hardware device drivers for the current platform. ADC devices will only be enabled if this option is itself enabled.

cdl_option CYGNUM_IO_ADC_SAMPLE_SIZE
  This option defines the sample size for the ADC devices. Given in bits, it will be rounded up to 8, 16 or 32 to define the cyg_adc_sample_t type. This option is usually set by the hardware device driver.

cdl_option CYGPKG_IO_ADC_SELECT_SUPPORT
  This option enables support for the select() API function on all ADC devices. This option can be disabled if the select() is not used, saving some code and data space.

In addition to the generic options, each hardware device driver defines some parameters for each device and channel. The exact names of the following option depends on the hardware device driver, but options of this form should be available in all drivers.

cdl_option CYGDAT_IO_ADC_EXAMPLE_CHANNELN_NAME
  This option specifies the name of the device for an ADC channel. Channel names should be of the form "/dev/adcXY" where X is the device number and Y the channel within that device.

cdl_option CYGNUM_IO_ADC_EXAMPLE_CHANNELN_BUFSIZE
  This option specifies the buffer size for an ADC channel. The value is expressed in multiples of cyg_adc_sample_t rather than bytes. The default value is 128.

cdl_option CYGNUM_IO_ADC_EXAMPLE_DEFAULT_RATE
  This option defines the initial default sample rate for all channels. The hardware driver may place constraints on the range of values this option may take.
ADC Device Drivers

Name
Overview — ADC Device Drivers

Introduction
This section describes how to write an ADC hardware device. While users of ADC devices do not need to read it, it may provide added insight into how the devices work.

Data Structures
An ADC hardware driver is represented by a number of data structures. These are generic device and channel data structures, a driver private device data structure, a generic character device table entry and a driver function table. Most of these structures are instantiated using macros, which will be described here.

The data structure instantiation for a typical single device, four channel ADC would look like this:

```c
//==========================================================================
// Instantiate data structures
// ------------------------------------------------------------------------
// Driver functions:
CYG_ADC_FUNCTIONS( example_adc_funs,
    example_adc_enable,
    example_adc_disable,
    example_adc_set_rate );
// ------------------------------------------------------------------------
// Device instance:
static example_adc_info example_adc_info0 =
{ .base = CYGARC_HAL_EXAMPLE_ADC_BASE,
  .vector = CYGNUM_HAL_INTERRUPT_ADC }
CYG_ADC_DEVICE( example_adc_device,
    &example_adc_funs,
    &example_adc_info0,
    CYGNUM_IO_ADC_EXAMPLE_DEFAULT_RATE );
// ------------------------------------------------------------------------
// Channel instances:
# define EXAMPLE_ADC_CHANNEL( __chan ) \ 
CYG_ADC_CHANNEL( example_adc_channel##__chan,
    __chan,
    CYGNUM_IO_ADC_EXAMPLE_CHANNEL##__chan##_BUFSIZE,
    &example_adc_device );
// DEVTAB_ENTRY( example_adc_channel##__chan##_device,
CYGDAT_IO_ADC_EXAMPLE_CHANNEL##__chan##_NAME,
```
The macro CYG_ADC_FUNCTIONS() instantiates a function table called example_adc_funs and populates it with the ADC driver functions (see later for details).

Then an instance of the driver private device data structure is instantiated. In addition to the device base address and interrupt vector shown here, this structure should contain the interrupt object and handle for attaching to the vector. It may also contain any other variables needed to manage the device.

The macro CYG_ADC_DEVICE() instantiates a cyg_adc_device structure, named example_adc_device which will contain pointers to the function table and private data structure. The initial sample rate is also supplied here.

For each channel, an ADC channel structure and a device table entry must be created. The macro EXAMPLE_ADC_CHANNEL() is defined to simplify this process. The macro CYG_ADC_CHANNEL defines a cyg_adc_channel structure, which contains the channel number, the buffer size, and a pointer to the device object defined earlier. The call to DEVTAB_ENTRY() generates a device table entry containing the configured channel name, a pointer to a device function table defined in the generic ADC driver, pointers to init and lookup functions implemented here, and a pointer to the channel data structure just defined.

Finally, four channels, numbered 0 to 3 are created.

**Functions**

There are several classes of function that need to be defined in an ADC driver. These are those function that go into the channel’s device table, those that go into the ADC device’s function table, calls that the driver makes into the generic ADC package, and interrupt handling functions.

**Device Table Functions**

These functions are placed in the standard device table entry for each channel and handle initialization and location of the device within the generic driver infrastructure.

static bool example_adc_init(struct cyg_devtab_entry *tab) This function is called from the device IO infrastructure to initialize the device. It should perform any work needed to start up the device, short of actually starting the generation of samples. This function will be called for each channel, so if there is initialization that only needs to be done once, such as creating an interrupt object, then care should be taken to do this. This function should also call cyg_adc_device_init() to initialize the generic parts of the driver.

static Cyg_ErrNo example_adc_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *sub_tab, const char *name) This function is called when a client looks up or opens a channel. It should call cyg_adc_channel_init() to initialize the generic part of the channel. It should also perform any operations needed to start the channel generating samples.
**Driver Functions**

These are the functions installed into the driver function table by the `CYG_ADC_FUNCTIONS()` macro.

```c
static void example_adc_enable( cyg_adc_channel *chan )
```

This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_ENABLE` config operation. It should take any steps needed to start the channel generating samples.

```c
static void example_adc_disable( cyg_adc_channel *chan )
```

This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_DISABLE` config operation. It should take any steps needed to stop the channel generating samples.

```c
static void example_adc_set_rate( cyg_adc_channel *chan, cyg_uint32 rate )
```

This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_RATE` config operation. It should take any steps needed to change the sample rate of the channel, or of the entire device.

**Generic Package Functions**

These functions are called by a hardware ADC device driver to perform operations in the generic ADC package.

```c
__externC void cyg_adc_device_init( cyg_adc_device *device )
```

This function is called from the driver’s `init` function and is used to initialize the `cyg_adc_device` object.

```c
__externC void cyg_adc_channel_init(cyg_adc_channel *chan)
```

This function is called from the driver’s `lookup` function and is used to initialize the `cyg_adc_channel` object.

```c
__externC cyg_uint32 cyg_adc_receive_sample(cyg_adc_channel *chan, cyg_adc_sample_t sample)
```

This function is called from the driver’s ISR to add a new sample to the buffer. The return value will be either zero, or `CYG_ISR_CALL_DSR` and should be ORed with the return value of the ISR.

```c
__externC void cyg_adc_wakeup(cyg_adc_channel *chan )
```

This function is called from the driver’s DSR to cause any threads waiting for data to wake up when a new sample is available. It should only be called if the `wakeup` field of the channel object is `true`.

**Interrupt Functions**

These functions are internal to the driver, but make calls on generic package functions. Typically an ADC device will have a single interrupt vector with which it signals available samples on the channels and any error conditions such as overruns.

```c
static cyg_uint32 example_adc_isr(cyg_vector_t vector, cyg_addrword_t data)
```

This function is the ISR attached to the ADC device’s interrupt vector. It is responsible for reading samples from the channels and passing them on to the generic layer. It needs to check each channel for data, and call `cyg_adc_receive_sample()` for each new sample available, and then ready the device for the next interrupt. Its activities are best explained by example:

```c
static cyg_uint32 example_adc_isr(cyg_vector_t vector, cyg_addrword_t data)
{
    cyg_adc_device *example_device = (cyg_adc_device *) data;
    example_adc_info *example_info = example_device->dev_priv;
    cyg_uint32 res = 0;
    int i;

    // Deal with errors if necessary
    DEVICE_CHECK_ERRORS( example_info );

    // Look for all channels with data available
```
for( i = 0; i < CHANNEL_COUNT; i++ )
{
    if( CHANNEL_SAMPLE_AVAILABLE(i) )
    {
        // Fetch data from this channel and pass up to higher
        // level.
        cyg_adc_sample_t data = CHANNEL_GET_SAMPLE(i);
        res |= CYG_ISR_HANDLED | cyg_adc_receive_sample( example_info->channel[i], data );
    }
}

// Clear any interrupt conditions
DEVICE_CLEAR_INTERRUPTS( example_info );

cygDrv_interrupt_acknowledge(example_info->vector);

return res;

static void example_adc_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t
data)
This function is the DSR attached to the ADC device’s interrupt vector. It is called by the kernel if the
ISR return value contains the CYG_ISRHandled bit. It needs to call cyg_adc_wakeup() for each channel that
has its wakeup field set. Again, and example should make it all clear:

static void example_adc_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data)
{
    cyg_adc_device *example_device = (cyg_adc_device *) data;
    example_adc_info *example_info = example_device->dev_priv;
    int i;

    // Look for all channels with pending wakeups
    for( i = 0; i < CHANNEL_COUNT; i++ )
    {
        if( example_info->channel[i]->wakeup )
            cyg_adc_wakeup( example_info->channel[i] );
    }
}
XII. Framebuffer Support
Overview

Name

Overview — eCos Support for Framebuffer Devices

Description

Framebuffer devices are the most common way for a computer system to display graphical output to users. There are immense variations in the implementations of such devices. CYGPKG_IO_FRAMEBUFFER provides an abstraction layer for use by application code and other packages. It defines an API for manipulating framebuffers, mapping this API on to functionality provided by the appropriate device driver. It also defines the interface which such device drivers should implement. For simple hardware it provides default implementations of much of this interface, greatly reducing the effort needed to write a device driver.

This package does not constitute a graphics library. It does not implement functionality like drawing text or arbitrary lines, let alone any kind of windowing system. Instead it operates at the lower level of individual pixels and blocks of pixels, in addition to control operations such as hardware initialization. Some applications may use the framebuffer API directly. Others will instead use a higher-level graphics library, and it is that library which uses the framebuffer API.

It is assumed that users are already familiar with the fundamentals of computer graphics, and no attempt is made here to explain terms like display depth, palette or pixel.

Note: This package is work-in-progress. The support for 1bpp, 2bpp and 4bpp display depths is incomplete. For double-buffered displays the code does not yet maintain a bounding box of the updated parts of the display. The package has also been designed to allow for expansion with new functionality.

Configuration

CYGPKG_IO_FRAMEBUFFER only contains hardware-independent code. It should be complemented by one or more framebuffer device drivers appropriate for the target platform. These drivers may be specific to the platform, or they may be more generic with platform-specific details such as the framebuffer memory base address provided by the platform HAL. When creating a configuration for a given target the device driver(s) will always be included automatically (assuming one has been written or ported). However by default this driver will be inactive and will not get built, so does not add any unnecessary size overhead for applications which do not require graphics. To activate the device driver CYGPKG_IO_FRAMEBUFFER must be added explicitly to the configuration, for example using ecosconfig add framebuffer. After this the full framebuffer API will be available to other packages and to application code.

This package contains very few configuration options. Instead it is left to device drivers or higher-level code to provide appropriate configurability. One option, CYGFUN_IO_FRAMEBUFFER_INSTALL_DEFAULT_PALETTE, relates to the initialization of paletted displays.

There are a number of calculated and inferred configuration options and a number of interfaces. These provide information such as whether or not there is a backlight. The most important one is CYGDAT_IO_FRAMEBUFFER_DEVICES, which holds a list of framebuffer identifiers for use with the macro-based API. If there is a single framebuffer device driver which supports one display in either landscape or portrait mode, the configuration option may hold a value like 240x320x8 320x240x8r90.
Application Programmer Interfaces

Framebuffer devices require a difficult choice between flexibility and performance. On the one hand the API should be able to support multiple devices driving separate displays, or a single device operating in different modes at different times. On the other hand graphics tends to involve very large amounts of I/O: even something as simple as drawing a background image can involve setting many thousands of pixels. Efficiency requires avoiding all possible overheads including function calls. Instead the API should make extensive use of macros or inline functions. Ideally details of the framebuffer device such as the stride would be known constants at compile-time, giving the compiler as much opportunity as possible to optimize the code. Clearly this is difficult if multiple framebuffer devices are in use or if the device mode may get changed at run-time.

To meet the conflicting requirements the generic framebuffer package provides two APIs: a fast macro API which requires selecting a single framebuffer device at compile or configure time; and a slower function API without this limitation. The two are very similar, for example:

```c
#include <cyg/io/framebuf.h>

void clear_screen(cyg_fb* fb, cyg_fb_colour colour)
{
    cyg_fb_fill_block(fb, 0, 0,
        fb->fb_width, fb->fb_height,
        colour);
}
```

or the equivalent macro version:

```c
#include <cyg/io/framebuf.h>

#define FRAMEBUF 240x320x8

void clear_screen(cyg_fb_colour colour)
{
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
        CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
        colour);
}
```

The function-based API works in terms of cyg_fb structures, containing all the information needed to manipulate the device. Each framebuffer device driver will export one or more of these structures, for example cyg_alaia_fb_240x320x8, and the driver documentation should list the variable names. The macro API works in terms of identifiers such as 240x320x8, and by a series of substitutions the main macro gets expanded to the appropriate device-specific code, usually inline. Again the device driver documentation should list the supported identifiers. In addition the configuration option CYGDAT_IO_FRAMEBUF_DEVICES will contain the full list. By convention the identifier will be specified by a `#define`d symbol such as FRAMEBUF, or in the case of graphics libraries by a configuration option.

If a platform has multiple framebuffer devices connected to different displays then there will be separate cyg_fb structures and macro identifiers for each one. In addition some devices can operate in multiple modes. For example a PC VGA card can operate in a monochrome 640x480 mode, an 8bpp 320x200 mode, and many other modes, but only one of these can be active at a time. The different modes are also represented by different cyg_fb structures and identifiers, effectively treating the modes as separate devices. It is the responsibility of higher-level code to ensure that only one mode is in use at a time.
Overview

It is possible to use the macro API with more than one device, basically by compiling the code twice with different values of `FRAMEBUF`, taking appropriate care to avoid identifier name clashes. This gives the higher performance of the macros at the cost of increased code size.

All of the framebuffer API, including exports of the device-specific `cyg_fb` structures, is available through a single header file `<cyg/io/framebuf.h>`. The API follows a number of conventions. Coordinates (0,0) correspond to the top-left corner of the display. All functions and macros which take a pair of coordinates have x first, y second. For block operations these coordinates are followed by width, then height. Coordinates and dimensions use `cyg_ucount16` variables, which for any processor should be the most efficient unsigned data type with at least 16 bits - usually plain unsigned integers. Colours are identified by `cyg_fb_colour` variables, again usually unsigned integers.

To allow for the different variants of the English language, the API allows for a number of alternate spellings. Colour and color can be used interchangeably, so there are data types `cyg_fb_colour` and `cyg_fb_color`, and functions `cyg_fb_make_colour` and `cyg_fb_make_color`. Similarly gray is accepted as a variant of grey so the predefined colours `CYG_FB_DEFAULT_PALETTE_LIGHTGREY` and `CYG_FB_DEFAULT_PALETTE_LIGHTGRAY` are equivalent.

The API is split into the following categories:

- **parameters**
  - getting information about a given framebuffer device such as width, height and depth. Colours management is complicated so has its own category.

- **control**
  - operations such as switching the display on and off, and more device-specific ones such as manipulating the backlight.

- **colours**
  - determining the colour format (monochrome, paletted, …), manipulating the palette, or constructing true colours.

- **drawing**
  - primitives for manipulating pixels and blocks of pixels.

- **iteration**
  - efficiently iterating over blocks of pixels.

**Thread Safety**

The framebuffer API never performs any locking so is not thread-safe. Instead it assumes that higher-level code such as a graphics library performs any locking that may be needed. Adding a mutex lock and unlock around every drawing primitive, including pixel writes, would be prohibitively expensive.

It is also assumed that the framebuffer will only be updated from thread context. With most hardware it will also be possible to access a framebuffer from DSR or ISR context, but this should be avoided in portable code.
Framebuffer Parameters

**Name**

Parameters — determining framebuffer capabilities

**Synopsis**

```c
#include <cyg/io/framebuf.h>

typedef struct cyg_fb {
    cyg_ucount16 fb_depth;
    cyg_ucount16 fb_format;
    cyg_ucount16 fb_width;
    cyg_ucount16 fb_height;
    #ifdef CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT
    cyg_ucount16 fb_viewport_width;
    cyg_ucount16 fb_viewport_height;
    #endif
    void* fb_base;
    cyg_ucount16 fb_stride;
    cyg_uint32 fb_flags0;
    ...
} cyg_fb;

cyg_fb* CYG_FB_STRUCT(FRAMEBUF);
cyg_ucount16 CYG_FB_DEPTH(FRAMEBUF);
cyg_ucount16 CYG_FB_FORMAT(FRAMEBUF);
cyg_ucount16 CYG_FB_WIDTH(FRAMEBUF);
cyg_ucount16 CYG_FB_HEIGHT(FRAMEBUF);
cyg_ucount16 CYG_FB_VIEWPORT_WIDTH(FRAMEBUF);
cyg_ucount16 CYG_FB_VIEWPORT_HEIGHT(FRAMEBUF);
void* CYG_FB_BASE(FRAMEBUF);
cyg_ucount16 CYG_FB_STRIDE(FRAMEBUF);
cyg_uint32 CYG_FB_FLAGS0(FRAMEBUF);
```

**Description**

When developing an application for a specific platform the various framebuffer parameters such as width and height are known, and the code can be written accordingly. However when writing code that should work on many platforms with different framebuffer devices, for example a graphics library, the code must be able to get these parameters and adapt.

Code using the function API can extract the parameters from the cyg_fb structures at run-time. The macro API provides dedicated macros for each parameter. These do not follow the usual eCos convention where the result is provided via an extra argument. Instead the result is returned as normal, and is guaranteed to be a compile-time constant. This allows code like the following:

```c
#if CYG_FB_DEPTH(FRAMEBUF) < 8
    ...
#else
    ...
```
Framebuffer Parameters

#else

or alternatively:

    if (CYG_FB_DEPTH(FRAMEBUF) < 8) {
        ...
    } else {
        ...
    }

or:

    switch (CYG_FB_DEPTH(FRAMEBUF)) {
        case 1 : ... break;
        case 2 : ... break;
        case 4 : ... break;
        case 8 : ... break;
        case 16 : ... break;
        case 32 : ... break;
    }

In terms of the code actually generated by the compiler these approaches have much the same effect. The macros expand to a compile-time constant so unnecessary code can be easily eliminated.

The available parameters are as follows:

depth

    The number of bits per pixel or bpp. The common depths are 1, 2, 4, 8, 16 and 32.

format

    How the pixel values are mapped on to visible colours, for example true colour or paletted or greyscale.

width

height

    The number of framebuffer pixels horizontally and vertically.

viewport width

viewport height

    With some devices the framebuffer height and/or width are greater than what the display can actually show. The display is said to offer a viewport into the larger framebuffer. The number of visible pixels is determined from the viewport width and height. The position of the viewport is controlled via an ioctl. Within a cyg_fb structure these fields are only present if CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT is defined, to avoid wasting data space on fields that are unnecessary for the current platform. For the macro API the viewport macros should only be used if CYG_FB_FLAGS0_VIEWPORT is set for the framebuffer:

    #if (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_VIEWPORT)
        ...
    #endif
Framebuffer Parameters

base
stride
For linear framebuffers these parameters provide the information needed to access framebuffer memory.
The stride is in bytes.

flags0
This gives further information about the hardware capabilities. Some of this overlaps with other parameters,
especially when it comes to colour, because it is often easier to test for a single flag than for a range
of colour modes. The current flags are:

CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER
Framebuffer memory is organized in a conventional fashion and can be accessed directly by higher-
level code using the base and stride parameters.

CYG_FB_FLAGS0_LE
This flag is only relevant for 1bpp, 2bpp and 4bpp devices and controls how the pixels are organized
within each byte. If the flag is set then the layout is little-endian: for a 1bpp device pixel (0,0)
occupies bit 0 of the first byte of framebuffer memory. The more common layout is big-endian
where pixel (0,0) occupies bit 7 of the first byte.

CYG_FB_FLAGS0_TRUE_COLOUR
The framebuffer uses a true colour format where the value of each pixel directly encodes the red,
green and blue intensities. This is common for 16bpp and 32bpp devices, and is occasionally used
for 8bpp devices.

CYG_FB_FLAGS0_PALETTE
The framebuffer uses a palette. A pixel value does not directly encode the colours, but instead acts
as an index into a separate table of colour values. That table may be read-only or read-write. Paletted
displays are common for 8bpp and some 4bpp displays.

CYG_FB_FLAGS0_WRITEABLE_PALETTE
The palette is read-write.

CYG_FB_FLAGS0_DELAYED_PALETTE_UPDATE
Palette updates can be synchronized to a vertical blank, in other words a brief time period when the
display is not being updated, by using CYG_FB_UPDATE_VERTICAL_RETRACE as the last argument to
cyg_fb_write_palette or CYG_FB_WRITE_PALETTE. With some hardware updating the palette in
the middle of a screen update may result in visual noise.

CYG_FB_FLAGS0_VIEWPORT
The framebuffer contains more pixels than can be shown on the display. Instead the display provides
a viewport into the framebuffer. An ioctl can be used to move the viewport.

CYG_FB_FLAGS0_DOUBLE_BUFFER
The display does not show the current contents of the framebuffer, so the results of drawing into
the framebuffer are not immediately visible. Instead higher-level code needs to perform an explicit
synch operation to update the display.
Framebuffer Parameters

**CYG_FB_FLAGS0_PAGE_FLIPPING**

The hardware supports two or more pages, each of width*height pixels, only one of which is visible on the display. This allows higher-level code to update one page without disturbing what is currently visible. An ioctl is used to switch the visible page.

**CYG_FB_FLAGS0_BLANK**

The display can be blanked without affecting the framebuffer contents or settings.

**CYG_FB_FLAGS0_BACKLIGHT**

There is a backlight which can be switched on or off. Some hardware provides finer-grained control over the backlight intensity.

**CYG_FB_FLAGS0_MUST_BE_ON**

Often it is desirable to perform some initialization such as clearing the screen or setting the palette before the display is switched on, to avoid visual noise. However not all hardware allows this. If this flag is set then it is possible to access framebuffer memory and the palette before the cyg_fb_on or CYG_FB_ON operation. It may also be possible to perform some other operations such as activating the backlight, but that is implementation-defined.

To allow for future expansion there are also flags1, flags2, and flags3 fields. These may get used for encoding additional ioctl functionality, support for hardware acceleration, and similar features.

**Linear Framebuffers**

There are drawing primitives for writing and reading individual pixels. However these involve a certain amount of arithmetic each time to get from a position to an address within the frame buffer, plus function call overhead if the function API is used, and this will slow down graphics operations.

When the framebuffer device is known at compile-time and the macro API is used then there are additional macros specifically for iterating over parts of the frame buffer. These should prove very efficient for many graphics operations. However if the device is selected at run-time then the macros are not appropriate and code may want to manipulate framebuffer memory directly. This is possible if two conditions are satisfied:

1. The CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER flag must be set. Otherwise framebuffer memory is either not directly accessible or has a non-linear layout.
2. The CYG_FB_FLAGS0_DOUBLE_BUFFER flag must be clear. An efficient double buffer synch operation requires knowing what part of the framebuffer have been updated, and the various drawing primitives will keep track of this. If higher-level code then starts manipulating the framebuffer directly the synch operation may perform only a partial update.

The base, stride, depth, width and height parameters, plus the CYG_FB_FLAGS0_LE flag for 1bpp, 2bpp and 4bpp devices, provide all the information needed to access framebuffer memory. A linear framebuffer has pixel (0,0) at the base address. Incrementing y means adding stride bytes to the pointer.

The base and stride parameters may be set even if CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER is clear. This can be useful if for example the display is rotated in software from landscape to portrait mode. However the meaning of these parameters for non-linear framebuffers is implementation-defined.
Framebuffer Control Operations

Name
Control Operations — managing a framebuffer

Synopsis

```c
#include <cyg/io/framebuf.h>

int cyg_fb_on(cyg_fb* fbdev);
int cyg_fb_off(cyg_fb* fbdev);
int cyg_fb_ioctl(cyg_fb* fbdev, cyg_uint16 key, void* data, size_t* len);
int CYG_FB_ON(FRAMEBUF);
int CYG_FB_OFF(FRAMEBUF);
int CYG_FB_IOCTL(FRAMEBUF, cyg_uint16 key, void* data, size_t* len);
```

Description

The main operations on a framebuffer are drawing and colour management. However on most hardware it is also necessary to switch the display on before the user can see anything, and application code should be able to control when this happens. There are also miscellaneous operations such as manipulating the backlight or moving the viewpoint. These do not warrant dedicated functions, especially since the functionality will only be available on some hardware, so an ioctl interface is used.

Switching the Display On or Off

With most hardware nothing will be visible until there is a call to cyg_fb_on or an invocation of the CYG_FB_ON macro. This will initialize the framebuffer control circuitry, start sending the data signals to the display unit, and switch on the display if necessary. The exact initialization semantics are left to the framebuffer device driver. In some cases the hardware may already be partially or fully initialized by a static constructor or by boot code that ran before eCos.

There are some circumstances in which initialization can fail, and this is indicated by a POSIX error code such as ENODEV. An example would be plug and play hardware where the framebuffer device is not detected at run-time. Another example is hardware which can operate in several modes, with separate cyg_fb structures for each mode, if the hardware is already in use for a different mode. A return value of 0 indicates success.

Some but not all hardware allows the framebuffer memory and, if present, the palette to be manipulated before the device is switched on. That way the user does not see random noise on the screen during system startup. The flag CYG_FB_FLAGS0_MUST_BE_ON should be checked:

```c
static void
init_screen(cyg_fb_colour background) {
    int result;

    #if (! (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_MUST_BE_ON))
        CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
```
Framebuffer Control Operations

Obviously if the application has already manipulated framebuffer memory or the palette but then the cyg_fb_on operation fails, the system is left in an undefined state.

It is also possible to switch a framebuffer device off, using the function cyg_fb_off or the macro CYG_FB_OFF, although this functionality is rarely used in embedded systems. The exact semantics of switching a device off are implementation-defined, but typically it involves shutting down the display, stopping the data signals to the display, and halting the control circuitry. The framebuffer memory and the palette are left in an undefined state, and application code should assume that both need full reinitializing when the device is switched back on. Some hardware may also provide a blank operation which typically just manipulates the display, not the whole framebuffer device. Normally cyg_fb_on returns 0. The API allows for a POSIX error code as with cyg_fb_on, but switching a device off is not an operation that is likely to fail.

If a framebuffer device can operate in several modes, represented by several cyg_fb structures and macro identifiers, then switching modes requires turning the current device off before turning the next one on.

Miscellaneous Control Operations

Some hardware functionality such as an LCD panel backlight is common but not universal. Supporting these does not warrant dedicated functions. Instead a catch-all ioctl interface is provided, with the arguments just passed straight to the device driver. This approach also allows for future expansion and for device-specific operations. cyg_fb_ioctl and CYG_FB_IOCTL take four arguments: a cyg_fb structure or framebuffer identifier; a key that specifies the operation to be performed; an arbitrary pointer, which should usually be a pointer to a data structure specific to the key; and a length field. Key values from 0 to 0x7fff are generic. Key values from 0x8000 onwards are reserved for the individual framebuffer device drivers, for device-specific functionality.

The length field should be set to the size of the data structure, and may get updated by the device driver.

With most ioctl operations the device can indicate whether or not it supports the functionality by one of the flags, for example:

```c
#include <stdio.h>
#include <unistd.h>

void backlight_off(cyg_fb* fb)
{
    if (fb->fb_flags0 & CYG_FB_FLAGS0_BACKLIGHT) {
        cyg_fb_ioctl_start
        size_t len = sizeof(cyg_fb_ioctl_backlight);
        int result;
        new_setting.fbbl_current = 0;
        result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_BACKLIGHT_SET,
```
Framebuffer Control Operations

The operation returns zero for success or a POSIX error code on failure, for example ENOSYS if the device driver does not implement the requested functionality.

Viewport

# define CYG_FB_IOCTL_VIEWPORT_GET_POSITION 0x0100
# define CYG_FB_IOCTL_VIEWPORT_SET_POSITION 0x0101

typedef struct cyg_fb_ioctl_viewport {
    cyg_ucount16 fbvp_x; // position of top-left corner of the viewport within
    cyg_ucount16 fbvp_y; // the framebuffer
    cyg_ucount16 fbvp_when; // set-only, now or vert retrace
} cyg_fb_ioctl_viewport;

On some targets the framebuffer device has a higher resolution than the display. Only a subset of the pixels, the viewport, is currently visible. Application code can exploit this functionality to achieve certain effects, for example smooth scrolling. Framebuffers which support this functionality will have the CYG_FB_FLAGS0_VIEWPORT flag set. The viewport dimensions are available as additional parameters to the normal framebuffer width and height.

The current position of the viewport can be obtained using an CYG_FB_IOCTL_VIEWPORT_GET_POSITION ioctl operation. The data argument should be a pointer to a cyg_fb_ioctl_viewport structure. On return the fbvp_x and fbvp_y fields will be filled in. To move the viewport use CYG_FB_IOCTL_VIEWPORT_SET_POSITION with fbvp_x and fbvp_y set to the top left corner of the new viewport within the framebuffer, and fbvp_when set to either CYG_FB_UPDATE_NOW or CYG_FB_UPDATE_VERTICAL RETRACE. If the device driver cannot easily synchronize to a vertical retrace period then this last field is ignored.

void
move_viewport(cyg_fb* fb, int dx, int dy)
{
    #ifdef CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT
    cyg_fb_ioctl_viewport viewport;
    int len = sizeof(cyg_fb_ioctl_viewport);
    int result;

    result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_VIEWPORT_GET_POSITION,
                          &viewport, &len);
    if (result != 0) { ...
    }
    if (((int)viewport.fbvp_x + dx) < 0) {
        viewport.fbvp_x = 0;
    } else if (((int)viewport.fbvp_x + dx + fb->fb_viewport_width) > fb->fb_width) {
        viewport.fbvp_x = fb->fb_width - fb->fb_viewport_width;
    } else {
        viewport.fbvp_x += dx;
    }
    if (((int)viewport.fbvp_y + dy) < 0) {

Framebuffer Control Operations

    viewport.fbvp_y = 0;
} else if ((viewport.fbvp_y + dy + fb->fb_viewport_height) > fb->fb_height) {
    viewport.fbvp_y = fb->fb_height - fb->fb_viewport_height;
} else {
    viewport.fbvp_y += dy;
}

result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_VIEWPORT_SET_POSITION, &viewport, &len);
if (result != 0) {
    ...
} else
    CYG_UNUSED_PARAM(cyg_fb*, fb);
    CYG_UNUSED_PARAM(int, dx);
    CYG_UNUSED_PARAM(int, dy);
#endif

If an attempt is made to move the viewport beyond the boundaries of the framebuffer then the resulting behaviour is undefined. Some hardware may behave reasonably, wrapping around as appropriate, but portable code cannot assume this. The above code fragment is careful to clip the viewport to the framebuffer dimensions.

Page Flipping

    #define CYG_FB_IOCTL_PAGE_FLIPPING_GET_PAGES 0x0200
    #define CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES 0x0201

typedef struct cyg_fb_ioctl_page_flip {
    cyg_uint32 fbpf_number_pages;
    cyg_uint32 fbpf_visible_page;
    cyg_uint32 fbpf_drawable_page;
    cyg_ucount16 fbpf_when; // set-only, now or vert retrace
} cyg_fb_ioctl_page_flip;

On some targets the framebuffer has enough memory for several pages, only one of which is visible at a time. This allows the application to draw into one page while displaying another. Once drawing is complete the display is flipped to the newly drawn page, and the previously displayed page is now available for updating. This technique is used for smooth animation, especially in games. The flag CYG_FB_FLAGS0_PAGE_FLIPPING indicates support for this functionality.

CYG_FB_IOCTL_PAGE_FLIPPING_GET_PAGES can be used to get the current settings of the page flipping support. The data argument should be a pointer to a cyg_fb_ioctl_page_flip structure. The resulting fbpf_number_pages field indicates the total number of pages available: 2 is common, but more pages are possible. fbpf_visible_page gives the page that is currently visible to the user, and will be between 0 and (fbpf_number_pages - 1). Similarly fbpf_drawable_page gives the page that is currently visible. It is implementation-defined whether or not the visible and drawable page can be the same one.

CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES can be used to change the visible and drawable page. The fbpf_number_pages field is ignored. fbpf_visible_page and fbpf_drawable_page give the new settings. fbpf_when should be one of CYG_FB_UPDATE_NOW or CYG_FB_UPDATE_VERTICAL_RETRACE, but may be ignored by some device drivers.

#if !(CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_PAGE_FLIPPING)
A page flip typically just changes a couple of pointers within the hardware and device driver. No attempt is made to synchronize the contents of the pages, that is left to higher-level code.

### Blanking the Screen

```c
#define CYG_FB_IOCTL_BLANK_GET 0x0300
#define CYG_FB_IOCTL_BLANK_SET 0x0301

typedef struct cyg_fb_ioctl_blank {
    cyg_bool fbbl_on;
} cyg_fb_ioctl_blank;
```
Framebuffer Control Operations

Some hardware allows the display to be switched off or blanked without shutting down the entire framebuffer device, greatly reducing power consumption. The current blanking state can be obtained using `CYG_FB_IOCTL_BLANK_GET` and the state can be updated using `CYG_FB_IOCTL_BLANK_SET`. The data argument should be a pointer to a `cyg_fb_ioctl_blank` structure. Support for this functionality is indicated by the `CYG_FB_FLAGS0_BLANK` flag.

```c
static cyg_bool
display_blanked(cyg_fb_* fb)
{
    cyg_fb_ioctl_blank blank;
    size_t len = sizeof(cyg_fb_ioctl_blank);
    if (! (fb->fb_flags0 & CYG_FB_FLAGS0_BLANK)) {
        return false;
    }
    (void) cyg_fb_ioctl(fb, CYG_FB_IOCTL_BLANK_GET, &blank, &len);
    return !blank.fbbl_on;
}
```

Controlling the Backlight

```c
#define CYG_FB_IOCTL_BACKLIGHT_GET 0x0400
#define CYG_FB_IOCTL_BACKLIGHT_SET 0x0401

typedef struct cyg_fb_ioctl_backlight {
    cyg_ucount32 fbbl_current;
    cyg_ucount32 fbbl_max;
} cyg_fb_ioctl_backlight;

Many LCD panels provide some sort of backlight, making the display easier to read at the cost of increased power consumption. Support for this is indicated by the `CYG_FB_FLAGS0_BACKLIGHT` flag. `CYG_FB_IOCTL_BACKLIGHT_GET` can be used to get both the current setting and the maximum value. If the maximum is 1 then the backlight can only be switched on or off. Otherwise it is possible to control the intensity.

```c
static void
set_backlight_50_percent(void)
{
    #if (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_BACKLIGHT)
        cyg_fb_ioctl_backlight backlight;
        size_t len = sizeof(cyg_fb_ioctl_backlight);
        CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_BACKLIGHT_GET, &backlight, &len);
        backlight.fbbl_current = (backlight.fbbl_max + 1) >> 1;
        CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_BACKLIGHT_SET, &backlight, &len);
    #endif
}
```
Framebuffer Colours

Name
Colours — formats and palette management

Synopsis

#include <cyg/io/framebuf.h>

typedef struct cyg_fb {
    cyg_ucount16 fb_depth;
    cyg_ucount16 fb_format;
    cyg_uint32 fb_flags0;
    ...
} cyg_fb;

extern const cyg_uint8 cyg_fb_palette_ega[16 * 3];
extern const cyg_uint8 cyg_fb_palette_vga[256 * 3];

#define CYG_FB_DEFAULT_PALETTE_BLACK 0x00
#define CYG_FB_DEFAULT_PALETTE_BLUE 0x01
#define CYG_FB_DEFAULT_PALETTE_GREEN 0x02
#define CYG_FB_DEFAULT_PALETTE_CYAN 0x03
#define CYG_FB_DEFAULT_PALETTE_RED 0x04
#define CYG_FB_DEFAULT_PALETTE_MAGENTA 0x05
#define CYG_FB_DEFAULT_PALETTE_BROWN 0x06
#define CYG_FB_DEFAULT_PALETTE_LIGHTGREY 0x07
#define CYG_FB_DEFAULT_PALETTE_LIGHITGRAY 0x07
#define CYG_FB_DEFAULT_PALETTE_DARKGREY 0x08
#define CYG_FB_DEFAULT_PALETTE_DARKGRAY 0x08
#define CYG_FB_DEFAULT_PALETTE_LIGHTBLUE 0x09
#define CYG_FB_DEFAULT_PALETTE_LIGHTGREEN 0x0A
#define CYG_FB_DEFAULT_PALETTE_LIGHTCYAN 0x0B
#define CYG_FB_DEFAULT_PALETTE_LIGHTRED 0x0C
#define CYG_FB_DEFAULT_PALETTE_LIGHTMAGENTA 0x0D
#define CYG_FB_DEFAULT_PALETTE_YELLOW 0x0E
#define CYG_FB_DEFAULT_PALETTE_WHITE 0x0F

cyg_ucount16 CYG_FB_FORMAT(framebuf);
void cyg_fb_read_palette(cyg_fb* fb, cyg_ucount32 first, cyg_ucount32 count, void* data);
void cyg_fb_write_palette(cyg_fb* fb, cyg_ucount32 first, cyg_ucount32 count, const void* data, cyg_ucount16 when);
cyg_fb_colour cyg_fb_make_colour(cyg_fb* fb, cyg_ucount8 r, cyg_ucount8 g, cyg_ucount8 b);
void cyg_fb_break_colour(cyg_fb* fb, cyg_fb_colour colour, cyg_ucount8 r, cyg_ucount8 g, cyg_ucount8 b);
void CYG_FB_READ_PALETTE(FRAMEBUF, cyg_ucount32 first, cyg_ucount32 count, void* data);
void CYG_FB_WRITE_PALETTE(FRAMEBUF, cyg_ucount32 first, cyg_ucount32 count, const void* data, cyg_ucount16 when);
cyg_fb_colour CYG_FB_MAKE_COLOUR(FRAMEBUF, cyg_ucount8 r, cyg_ucount8 g, cyg_ucount8 b);
void CYG_FB_BREAK_COLOUR(FRAMEBUF, cyg_fb_colour colour, cyg_ucount8 r, cyg_ucount8 g, cyg_ucount8 b);
Framebuffer Colours

Description

Managing colours can be one of the most difficult aspects of writing graphics code, especially if that code is intended to be portable to many different platforms. Displays can vary from 1bpp monochrome, via 2bpp and 4bpp greyscale, through 4bpp and 8bpp paletted, and up to 16bpp and 32bpp true colour - and those are just the more common scenarios. The various drawing primitives like cyg_fb_write_pixel work in terms of cyg_fb_colour values, usually an unsigned integer. Exactly how the hardware interprets a cyg_fb_colour depends on the format.

Colour Formats

There are a number of ways of finding out how these values will be interpreted by the hardware:

1. The CYG_FB_FLAGS0_TRUE_COLOUR flag is set for all true colour displays. The format parameter can be examined for more details but this is not usually necessary. Instead code can use cyg_fb_make_colour or CYG_FB_MAKE_COLOUR to construct a cyg_fb_colour value from red, green and blue components.

2. If the CYG_FB_FLAGS0_WRITEABLE_PALETTE flag is set then a cyg_fb_colour value is an index into a lookup table known as the palette, and this table contains red, green and blue components. The size of the palette is determined by the display depth, so 16 entries for a 4bpp display and 256 entries for an 8bpp display. Application code or a graphics library can install its own palette so can control exactly what colour each cyg_fb_colour value corresponds to. Alternatively there is support for installing a default palette.

3. If CYG_FB_FLAGS0_PALETTE is set but CYG_FB_FLAGS0_WRITEABLE_PALETTE is clear then the hardware uses a fixed palette. There is no easy way for portable software to handle this case. The palette can be read at run-time, allowing the application’s desired colours to be mapped to whichever palette entry provides the best match. However normally it will be necessary to write code specifically for the fixed palette.

4. Otherwise the display is monochrome or greyscale, depending on the depth. There are still variations, for example on a monochrome display colour 0 can be either white or black.

As an alternative or to provide additional information, the exact colour format is provided by the fb_format field of the cyg_fb structure or by the CYG_FB_FORMAT macro. It can be one of the following (more entries may be added in future):

CYG_FB_FORMAT_1BPP_MONO_0_BLACK

simple 1bpp monochrome display, with 0 as black or the darker of the two colours, and 1 as white or the lighter colour.

CYG_FB_FORMAT_1BPP_MONO_0_WHITE

simple 1bpp monochrome display, with 0 as white or the lighter of the two colours, and 1 as black or the darker colour.

CYG_FB_FORMAT_1BPP_PAL888

a 1bpp display which cannot easily be described as monochrome. This is unusual and not readily supported by portable code. It can happen if the framebuffer normally runs at a higher depth, for example 4bpp or 8bpp paletted, but is run at only 1bpp to save memory. Hence only two of the palette entries are used, but can be set to arbitrary colours. The palette may be read-only or read-write.

CYG_FB_FORMAT_2BPP_GREYSCALE_0_BLACK

a 2bpp display offering four shades of grey, with 0 as black or the darkest of the four shades, and 3 as white or the lightest.
Framebuffer Colours

**CYG_FB_FORMAT_2BPP_GREYSCALE_0_WHITE**

A 2bpp display offering four shades of grey, with 0 as white or the lightest of the four shades, and 3 as black or the darkest.

**CYG_FB_FORMAT_2BPP_PAL888**

A 2bpp display which cannot easily be described as greyscale, for example providing black, red, blue and white as the four colours. This is unusual and not readily supported by portable code. It can happen if the framebuffer normally runs at a higher depth, for example 4bpp or 8bpp palleted, but is run at only 2bpp to save memory. Hence only four of the palette entries are used, but can be set to arbitrary colours. The palette may be read-only or read-write.

**CYG_FB_FORMAT_4BPP_GREYSCALE_0_BLACK**

A 4bpp display offering sixteen shades of grey, with 0 as black or the darkest of the 16 shades, and 15 as white or the lightest.

**CYG_FB_FORMAT_4BPP_GREYSCALE_0_WHITE**

A 4bpp display offering sixteen shades of grey, with 0 as white or the lightest of the 16 shades, and 15 as black or the darkest.

**CYG_FB_FORMAT_4BPP_PAL888**

A 4bpp paletted display, allowing for 16 different colours on screen at the same time. The palette may be read-only or read-write.

**CYG_FB_FORMAT_8BPP_PAL888**

An 8bpp paletted display, allowing for 256 different colours on screen at the same time. The palette may be read-only or read-write.

**CYG_FB_FORMAT_8BPP_TRUE_332**

An 8bpp true colour display, with three bits (eight levels) of red and green intensity and two bits (four levels) of blue intensity.

**CYG_FB_FORMAT_16BPP_TRUE_565**

A 16bpp true colour display with 5 bits each for red and blue and 6 bits for green.

**CYG_FB_FORMAT_16BPP_TRUE_555**

A 16bpp true colour display with five bits each for red, green and blue, and one unused bit.

**CYG_FB_FORMAT_32BPP_TRUE_0888**

A 32bpp true colour display with eight bits each for red, green and blue and eight bits unused.

For the true colour formats the format does not define exactly which bits in the pixel are used for which colour. Instead the `cyg_fb_make_colour` and `cyg_fb_break_colour` functions or the equivalent macros should be used to construct or decompose pixel values.

Paletted Displays

Palettes are the common way of implementing low-end colour displays. There are two variants. A read-only palette provides a fixed set of colours and it is up to application code to use these colours appropriately. A read-
write palette allows the application to select its own set of colours. Displays providing a read-write palette will have the CYG_FB_FLAGS0_WRITEABLE_PALETTE flag set in addition to CYG_FB_FLAGS0_PALETTE.

Even if application code can install its own palette, many applications do not exploit this functionality and instead stick with a default. There are two standard palettes: the 16-entry PC EGA for 4bpp displays; and the 256-entry PC VGA, a superset of the EGA one, for 8bpp displays. This package provides the data for both, in the form of arrays cyg_fb_palette_ega and cyg_fb_palette_vga, and 16 #define’s such as CYG_FB_DEFAULT_PALETTE_BLACK for the EGA colours and the first 16 VGA colours. By default device drivers for read-write paletted displays will install the appropriate default palette, but this can be suppressed using configuration option CYGFUN_IO_FRAMEBUF_INSTALL_DEFAULT_PALETTE. If a custom palette will be used then installing the default palette involves wasting 48 or 768 bytes of memory.

It should be emphasized that displays vary widely. A colour such as CYG_FB_DEFAULT_PALETTE_YELLOW may appear rather differently on two different displays, although it should always be recognizable as yellow. Developers may wish to fine-tune the palette for specific hardware.

The current palette can be retrieved using cyg_fb_read_palette or CYG_FB_READ_PALETTE. The first and count arguments control which palette entries should be retrieved. For example, to retrieve just palette entry 12 first should be set to 12 and count should be set to 1. To retrieve all 256 entries for an 8bpp display, first should be set to 0 and count should be set to 256. The data argument should point at an array of bytes, allowing three bytes for every entry. Byte 0 will contain the red intensity for the first entry, byte 1 green and byte 2 blue.

For read-write palettes the palette can be updated using cyg_fb_write_palette or CYG_FB_WRITE_PALETTE. The first and count arguments are the same as for cyg_fb_read_palette, and the data argument should point at a suitable byte array packed in the same way. The when argument should be one of CYG_FB_UPDATE_NOW or CYG_FB_UPDATE_VERTICAL_RETRACE. With some displays updating the palette in the middle of an update may result in visual noise, so synchronizing to the vertical retrace avoids this. However not all device drivers will support this.

There is an assumption that palette entries use 8 bits for each of the red, green and blue colour intensities. This is not always the case, but the device drivers will perform appropriate adjustments. Some hardware may use only 6 bits per colour, and the device driver will ignore the bottom two bits of the supplied intensity values. Occasionally hardware may use more than 8 bits, in which case the supplied 8 bits are shifted left appropriately and zero-padded. Device drivers for such hardware may also provide device-specific routines to manipulate the palette in a non-portable fashion.

**True Colour displays**

True colour displays are often easier to manage than paletted displays. However this comes at the cost of extra memory. A 16bpp true colour display requires twice as much memory as an 8bpp paletted display, yet can offer only 32 or 64 levels of intensity for each colour as opposed to the 256 levels provided by a palette. It also requires twice as much video memory bandwidth to send all the pixel data to the display for every refresh, which may impact the performance of the rest of the system. A 32bpp true colour display offers the same colour intensities but requires four times the memory and four times the bandwidth.

Exactly how the colour bits are organized in a cyg_fb_colour pixel value is not defined by the colour format. Instead code should use the cyg_fb_make_colour or CYG_FB_MAKE_COLOUR primitives. These take 8-bit intensity levels for red, green and blue, and return the corresponding cyg_fb_colour. When using the macro interface the arithmetic happens at compile-time, for example:

```c
#define BLACK CYG_FB_MAKE_COLOUR(FRAMEBUF, 0, 0, 0)
#define WHITE CYG_FB_MAKE_COLOUR(FRAMEBUF, 255, 255, 255)
#define RED CYG_FB_MAKE_COLOUR(FRAMEBUF, 255, 0, 0)
#define GREEN CYG_FB_MAKE_COLOUR(FRAMEBUF, 0, 255, 0)
```
Displays vary widely so the numbers may need to be adjusted to give the exact desired colours.

For symmetry there are also `cyg_fb_break_colour` and `CYG_FB_BREAK_COLOUR` primitives. These take a `cyg_fb_colour` value and decompose it into its red, green and blue components.
Framebuffer Colours
Framebuffer Drawing Primitives

Name
Drawing Primitives — updating the display

Synopsis

```c
#include <cyg/io/framebuf.h>

void cyg_fb_write_pixel(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_fb_colour colour);
cyg_fb_colour cyg_fb_read_pixel(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y);
void cyg_fb_write_hline(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len, cyg_fb_colour colour);
void cyg_fb_write_vline(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len, cyg_fb_colour colour);
void cyg_fb_fill_block(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, cyg_fb_colour colour);
void cyg_fb_write_block(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, const void* data, cyg_ucount16 offset, cyg_ucount16 stride);
void cyg_fb_read_block(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, void* data, cyg_ucount16 offset, cyg_ucount16 stride);
void cyg_fb_move_block(cyg_fb* fbdev, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, const void* data, cyg_ucount16 offset, cyg_ucount16 stride);
void cyg_fb_synch(cyg_fb* fbdev, cyg_ucount16 when);
void CYG_FB_WRITE_PIXEL(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_fb_colour colour);
cyg_fb_colour CYG_FB_READ_PIXEL(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y);
void CYG_FB_WRITE_HLINE(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len, cyg_fb_colour colour);
void CYG_FB_WRITE_VLINE(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len, cyg_fb_colour colour);
void CYG_FB_FILL_BLOCK(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, cyg_fb_colour colour);
void CYG_FB_WRITE_BLOCK(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, const void* data, cyg_ucount16 offset, cyg_ucount16 stride);
void CYG_FB_READ_BLOCK(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, void* data, cyg_ucount16 offset, cyg_ucount16 stride);
void CYG_FB_MOVE_BLOCK(FRAMEBUF, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 width, cyg_ucount16 height, cyg_ucount16 new_x, cyg_ucount16 new_y);
void CYG_FB_SYNCH(FRAMEBUF, cyg_ucount16 when);
```

Description

The eCos framebuffer infrastructure defines a small number of drawing primitives. These are not intended to provide full graphical functionality like multiple windows, drawing text in arbitrary fonts, or anything like that. Instead they provide building blocks for higher-level graphical toolkits. The available primitives are:

1. Manipulating individual pixels.
2. Drawing horizontal and vertical lines.
Framebuffer Drawing Primitives

3. Block fills.

4. Moving blocks between the framebuffer and main memory.

5. Moving blocks within the framebuffer.

6. For double-buffered devices, synchronizing the framebuffer contents with the actual display.

There are two versions for each primitive: a macro and a function. The macro can be used if the desired
framebuffer device is known at compile-time. Its first argument should be a framebuffer identifier, for example
320x240x16, and must be one of the entries in the configuration option CYGDAT_IO_FRAMEBUFFER_DEVICES. In
the examples below it is assumed that FRAMEBUF has been #define’d to a suitable identifier. The function can
be used if the desired framebuffer device is selected at run-time. Its first argument should be a pointer to the
appropriate cyg_fb structure.

The pixel, line, and block fill primitives take a cyg_fb_colour argument. For details of colour handling see
Framebuffer Colours. This argument should have no more bits set than are appropriate for the display depth.
For example on a 4bpp only the bottom four bits of the colour may be set, otherwise the behaviour is undefined.
None of the primitives will perform any run-time error checking, except possibly for some assertions in a debug
build. If higher-level code provides invalid arguments, for example trying to write a block which extends past
the right hand side of the screen, then the system’s behaviour is undefined. It is the responsibility of higher-level
code to perform clipping to the screen boundaries.

Manipulating Individual Pixels

The primitives for manipulating individual pixels are very simple: a pixel can be written or read back. The
following example shows one way of drawing a diagonal line:

```c
void
draw_diagonal(cyg_fb* fb, cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len, cyg_fb_colour colour)
{
    while ( len-- ) {
        cyg_fb_write_pixel(fb, x++, y++, colour);
    }
}
```

The next example shows how to draw a horizontal XOR line on a 1bpp display.

```c
void
draw_horz_xor(cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len)
{
    cyg_fb_colour colour;
    while ( len-- ) {
        colour = CYG_FB_READ_PIXEL(FRAMEBUF, x, y);
        CYG_FB_WRITE_PIXEL(FRAMEBUF, x++, y, colour ^ 0x01);
    }
}
```

The pixel macros should normally be avoided. Determining the correct location within framebuffer memory
corresponding to a set of coordinates for each pixel is a comparatively expensive operation. Instead there is
direct support for iterating over parts of the display, avoiding unnecessary overheads.
Drawing Simple Lines

Higher-level graphics code often needs to draw single-pixel horizontal and vertical lines. If the application involves multiple windows then these will usually have thin borders around them. Widgets such as buttons and scrollbars also often have thin borders.

cyg_fb_draw_hline and CYG_FB_DRAW_HLINE draw a horizontal line of the specified colour, starting at the $x$ and $y$ coordinates and extending to the right (increasing $x$) for a total of $len$ pixels. A 50 pixel line starting at $(100,100)$ will end at $(149,100)$.

cyg_fb_draw_vline and CYG_FB_DRAW_VLINE take the same arguments, but the line extends down (increasing $y$).

These primitives do not directly support drawing lines more than one pixel thick, but block fills can be used to achieve those. There is no generic support for drawing arbitrary lines, instead that is left to higher-level graphics toolkits.

Block Fills

Filling a rectangular part of the screen with a solid colour is another common requirement for higher-level code. The simplest example is during initialization, to set the display’s whole background to a known value. Block fills are also often used when creating new windows or drawing the bulk of a simple button or scrollbar widget. cyg_fb_fill_block and CYG_FB_FILL_BLOCK provide this functionality.

The $x$ and $y$ arguments specify the top-left corner of the block to be filled. The $width$ and $height$ arguments specify the number of pixels affected, a total of $width \times height$. The following example illustrates part of the process for initializing a framebuffer, assumed here to have a writeable palette with default settings.

```c
int display_init(void)
{
    int result = CYG_FB_ON(FRAMEBUF);
    if ( result ) {
        return result;
    }
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      CYG_FB_DEFAULT_PALETTE_WHITE);
    ...
}
```

Copying Blocks between the Framebuffer and Main Memory

The block transfer primitives serve two main purposes: drawing images, and saving parts of the current display to be restored later. For simple linear framebuffers the primitives just implement copy operations, with no data conversion of any sort. For non-linear ones the primitives act as if the framebuffer memory was linear.

For example, consider a 2bpp display where the two bits for a single pixel are split over two separate bytes in framebuffer memory, or two planes. For a block write operation the source data should still be organized with four full pixels per byte, as for a linear framebuffer of the same depth, and the block write primitive will distribute the bits over the framebuffer memory as required. Similarly a block read will combine the appropriate bits from different locations in framebuffer memory and the resulting memory block will have four full pixels per byte.
Framebuffer Drawing Primitives

Because the block transfer primitives perform no data conversion, if they are to be used for rendering images then those images should be pre-formatted appropriately for the framebuffer device. For small images this would normally happen on the host-side as part of the application build process. For larger images it will usually be better to store them in a compressed format and decompress them at run-time, trading off memory for cpu cycles.

The $x$ and $y$ arguments specify the top-left corner of the block to be transferred, and the width and height arguments determine the size. The $data$, $offset$ and $stride$ arguments determine the location and layout of the block in main memory:

$\textbf{data}$

The source or destination for the transfer. For 1bpp, 2bpp and 4bpp devices the data will be packed in accordance with the framebuffer device’s endianness as per the CYG_FB_FLAGS0_LE flag. Each row starts in a new byte so there may be some padding on the right. For 16bpp and 32bpp the data should be aligned to the appropriate boundary.

$\textbf{offset}$

Sometimes only part of an image should be written to the screen. A vertical offset can be achieved simply by adjusting $data$ to point at the appropriate row within the image instead of the top row. For 8bpp, 16bpp and 32bpp displays an additional horizontal offset can also be achieved by adjusting $data$. However for 1bpp, 2bpp and 4bpp displays the starting position within the image may be in the middle of a byte. Hence the horizontal pixel offset can instead be specified with the $offset$ argument.

$\textbf{stride}$

This indicates the number of bytes between rows. Usually it will be related to the width, but there are exceptions such as when drawing only part of an image.

The following example fills a 4bpp display with an image held in memory and already in the right format. If the image is smaller than the display it will be centered. If the image is larger then the center portion will fill the entire display.

```c
void draw_image(const void* data, int width, int height)
{
    cyg_ucount16 stride;
    cyg_ucount16 x, y, offset;

    #if (4 != CYG_FB_DEPTH(FRAMEBUF))
        #error This code assumes a 4bpp display
    #endif

    stride = (width + 1) >> 1; // 4bpp to byte stride

    if (width < CYG_FB_WIDTH(FRAMEBUF)) {
        x = (CYG_FB_WIDTH(FRAMEBUF) - width) >> 1;
        offset = 0;
    } else {
        x = 0;
        offset = (width - CYG_FB_WIDTH(FRAMEBUF)) >> 1;
        width = CYG_FB_WIDTH(FRAMEBUF);
    }

    if (height < CYG_FB_HEIGHT(FRAMEBUF)) {
        y = (CYG_FB_HEIGHT(FRAMEBUF) - height) >> 1;
    } else {
        y = 0;
    }
```
Moving Blocks with the Framebuffer

Sometimes it is necessary to move a block of data around the screen, especially when using a higher-level graphics toolkit that supports multiple windows. Block moves can be implemented by a read into main memory followed by a write block, but this is expensive and imposes an additional memory requirement. Instead the framebuffer infrastructure provides a generic block move primitive. It will handle all cases where the source and destination positions overlap. The \texttt{x} and \texttt{y} arguments specify the top-left corner of the block to be moved, and \texttt{width} and \texttt{height} determine the block size. \texttt{new\_x} and \texttt{new\_y} specify the destination. The source data will remain unchanged except in areas where it overlaps the destination.

Synchronizing Double-Buffered Displays

Some framebuffer devices are double-buffered: the framebuffer memory that gets manipulated by the drawing primitives is separate from what is actually displayed, and a synch operation is needed to update the display. In some cases this may be because the actual display memory is not directly accessible by the processor, for example it may instead be attached via an SPI bus. Instead drawing happens in a buffer in main memory, and then this gets transferred over the SPI bus to the actual display hardware during a synch. In other cases it may be a software artefact. Some drawing operations, especially ones involving complex curves, can take a very long time and it may be considered undesirable to have the user see this happening a few pixels at a time. Instead the drawing happens in a separate buffer in main memory and then a double buffer synch just involves a block move to framebuffer memory. Typically that block move is much faster than the drawing operation. Obviously there is a cost: an extra area of memory, and the synch operation itself can consume many cycles and much of the available memory bandwidth.

It is the responsibility of the framebuffer device driver to provide the extra main memory. As far as higher-level code is concerned the only difference between an ordinary and a double-buffered display is that with the latter changes do not become visible until a synch operation has been performed. The framebuffer infrastructure provides support for a bounding box, keeping track of what has been updated since the last synch. This means only the updated part of the screen has to be transferred to the display hardware.

The synch primitives take two arguments. The first identifies the framebuffer device. The second should be one of \texttt{CYG\_FB\_UPDATE\_NOW} for an immediate update, or \texttt{CYG\_FB\_UPDATE\_VERTICAL\_RETRACE}. Some display hardware involves a lengthy vertical retrace period every 10-20 milliseconds during which nothing gets drawn to the screen, and performing the synch during this time means that the end user is unaware of the operation (assuming the synch can be completed in the time available). When the hardware supports it, specifying \texttt{CYG\_FB\_UPDATE\_VERTICAL\_RETRACE} means that the synch operation will block until the next vertical retrace takes place and then perform the update. This may be an expensive operation, for example it may involve polling a bit in a register. In a multi-threaded environment it may also be unreliable because the thread performing the synch may get interrupted or rescheduled in the middle of the operation. When the hardware does not involve vertical retraces, or when there is no easy way to detect them, the second argument to the synch operation will just be ignored and the update will always happen immediately.

It is up to higher level code to determine when a synch operation is appropriate. One approach for typical event-driven code is to perform the synch at the start of the event loop, just before waiting for an input or timer...
Framebuffer Drawing Primitives

event. This may not be optimal. For example if there two small updates to opposite corners of the screen then it would be better to make two synch calls with small bounding boxes, rather than a single synch call with a large bounding box that requires most of the framebuffer memory to be updated.

Leaving out the synch operations leads to portability problems. On hardware which does not involve double-buffering the synch operation is a no-op, usually eliminated at compile-time, so invoking synch does not add any code size or cpu cycle overhead. On double-buffered hardware, leaving out the synch means the user cannot see what has been drawn into the framebuffer.
Framebuffer Pixel Manipulation

Name
Pixel Manipulation — iterating over the display

Synopsis

#include <cyg/io/framebuf.h>

CYG_FB_PIXEL0_VAR(FRAMEBUF);
void CYG_FB_PIXEL0_SET(FRAMEBUF, cyg_u16 x, cyg_u16 y);
void CYG_FB_PIXEL0_GET(FRAMEBUF, cyg_u16 x, cyg_u16 y);
void CYG_FB_PIXEL0_ADDX(FRAMEBUF, cyg_u16 incr);
void CYG_FB_PIXEL0_ADDY(FRAMEBUF, cyg_u16 incr);
void CYG_FB_PIXEL0_WRITE(FRAMEBUF, cyg_fb_colour colour);
cyg_fb_colour CYG_FB_PIXEL0_READ(FRAMEBUF);
void CYG_FB_PIXEL0_FLUSHABS(FRAMEBUF, cyg_u16 x0, cyg_u16 y0, cyg_u16 width, cyg_u16 height);
void CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, cyg_u16 x0, cyg_u16 y0, cyg_u16 dx, cyg_u16 dy);

Description

A common requirement for graphics code is to iterate over parts of the framebuffer. Drawing text typically involves iterating over a block of pixels for each character, say 8 by 8, setting each pixel to either a foreground or background colour. Drawing arbitrary lines typically involves moving to the start position and then adjusting the x and y coordinates until the end position is reached, setting a single pixel each time around the loop. Drawing images which are not in the frame buffer’s native format typically involves iterating over a block of pixels, from top to bottom and left to right, setting pixels as the image is decoded.

Functionality like this can be implemented in several ways. One approach is to use the pixel write primitive. Typically this involves some arithmetic to get from the x and y coordinates to a location within framebuffer memory so it is fairly expensive compared with a loop which just increments a pointer. Another approach is to write the data first to a separate buffer in memory and then use a block write primitive to move it to the framebuffer, but again this involves overhead. The eCos framebuffer support provides a third approach: a set of macros specifically for iterating over the frame buffer. Depending on the operation being performed and the details of the framebuffer implementation, these macros may be optimal or near-optimal. Obviously there are limitations. Most importantly the framebuffer device must be known at compile-time: the compiler can do a better job optimizing the code if information such as the frame buffer width are constant. Also each iteration must be performed within a single variable scope: it is not possible to do some of the iteration in one function, some in another.

The Pixel Macros

All the pixel macros take a framebuffer identifier as their first argument. This is the same identifier that can be used with the other macros like CYG_FB_WRITE_HLINE and CYG_FB_ON, one of the entries in the configuration option CYGDAT_IO_FRAMEBUF_DEVICES. Using an invalid identifier will result in numerous compile-time error
Framebuffer Pixel Manipulation

messages which may bear little resemblance to the original code. In the examples below it is assumed that
FRAMEBUF has been #define’d to a suitable identifier.

Typical use of the pixel macros will look like this:

```
CYG_FB_PIXEL0_VAR(FRAMEBUF);
...
CYG_FB_PIXEL0.FlushABS(FRAMEBUF, x, y, width, height);
```

The \texttt{VAR} macro will define one or more local variables to keep track of the current pixel position, as appropriate
to the framebuffer device. The other pixel macros will then use these variables. For a simple 8bpp linear
framebuffer there will be just a byte pointer. For a 1bpp display there may be several variables: a byte pointer,
a bit index within that byte, and possibly a cached byte; using a cached value means that the framebuffer
may only get read and written once for every 8 pixels, and the compiler may well allocate a register for the
cached value; on some platforms framebuffer access will bypass the processor’s main cache, so reading from
or writing to framebuffer memory will be slow; reducing the number of framebuffer accesses may greatly
improve performance.

Because the \texttt{VAR} macro defines one or more local variables it is normally placed at the start of a function or
block, alongside other local variable definitions.

One the iteration has been completed there should be a \texttt{FLUSHABS} or \texttt{FLUSHREL} macro. This serves two pur-
poses. First, if the local variables involve a dirty cached value or similar state then this will be written back.
Second, for double-buffered displays the macro sets a bounding box for the part of the screen that has been
updated. This allows the double buffer synch operation to update only the part of the display that has been
modified, without having to keep track of the current bounding box for every updated pixel. For \texttt{FLUSHABS}
the \texttt{x0} and \texttt{y0} arguments specify the top-left corner of the bounding box, which extends for \texttt{width} by \texttt{height}
pixels. For \texttt{FLUSHREL} \texttt{x0} and \texttt{y0} still specify the top-left corner, but the bottom-right corner is now determined
from the current pixel position offset by \texttt{dx} and \texttt{dy}. More specifically, \texttt{dx} should move the current horizontal
position one pixel to the right of the right-most pixel modified, such that \((x + \texttt{dx}) - x0\) gives the width of the
bounding box. Similarly \texttt{dy} should move the current vertical position one pixel below the bottom-most pixel
modified. In typical code the current pixel position will already correspond in part or in whole to the bounding
box corner, as a consequence of iterating over the block of memory.

If a pixel variable has been used only for reading framebuffer memory, not for modifying it, then it should
still be flushed. A \texttt{FLUSHABS} with a width and height of 0 can be used to indicate that the bounding box is
empty. If it is known that the framebuffer device being used does not support double-buffering then again it is
possible to specify an empty bounding box. Otherwise portable code should specify a correct bounding box.
If the framebuffer device that ends up being used does not support double buffering then the relevant macro
arguments are eliminated at compile-time and do not result in any unnecessary code. In addition if there is no
cached value or other state then the whole flush operation will be a no-op and no code will be generated.

Failure to perform the flush may result in strange drawing artefacts on some displays which can be very hard
to debug. A \texttt{FLUSHABS} or \texttt{FLUSHREL} macro only needs to be invoked once, at the end of the iteration.

The \texttt{SET} macro sets the current position within the framebuffer. It can be used many times within an iteration.
However it tends to be somewhat more expensive than \texttt{ADDX} or \texttt{ADDY}, so usually \texttt{SET} is only executed once at
the start of an iteration.

```
CYG_FB_PIXEL0_VAR(FRAMEBUF);
CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
...
CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, 0, 0);
```

The \texttt{GET} macro retrieves the \texttt{x} and \texttt{y} coordinates corresponding to the current position. It is provided mainly
for symmetry, but can prove useful for debugging.
Framebuffer Pixel Manipulation

```c
CYG_FB_PIXEL0_VAR(FRAMEBUF);
CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);

#ifdef DEBUG
CYG_FB_PIXEL0_GET(FRAMEBUF, new_x, new_y);
diag_printf("Halfway through: x now %d, y now %d\n", new_x, new_y);
#endif

CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, 0, 0);
```

The `ADDX` and `ADDY` macros adjust the current position. The most common increments are 1 and -1, moving to the next or previous pixel horizontally or vertically, but any increment can be used.

```c
CYG_FB_PIXEL0_VAR(FRAMEBUF);
CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
for (rows = height; rows; rows--) {
    for (columns = width; columns; columns--) {
        <perform operation>
        CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
    }
    CYG_FB_PIXEL0_ADDX(FRAMEBUF, -1 * width);
    CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
}
CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, width, 0);
```

Here the current position is moved one pixel to the right each time around the inner loop. In the outer loop the position is first moved back to the start of the current row, then moved one pixel down. For the final flush the current x position is off by `width`, but the current y position is already correct.

The final two macros `READ` and `WRITE` can be used to examine or update the current pixel value.

```c
CYG_FB_PIXEL0_VAR(FRAMEBUF);
CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
for (rows = height; rows; rows--) {
    for (columns = width; columns; columns--) {
        cyg_fb_colour colour = CYG_FB_PIXEL0_READ(FRAMEBUF);
        if (colour == colour_to_replace) {
            CYG_FB_PIXEL0_WRITE(FRAMEBUF, replacement);
        }
    }
    CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
}
CYG_FB_PIXEL0_ADDX(FRAMEBUF, -1 * width);
CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, width, 0);
```

Concurrent Iterations

Although uncommon, in some cases application code may need to iterate over two or more blocks. An example might be an advanced block move where each copied pixel requires some processing. To support this there are `PIXEL1`, `PIXEL2` and `PIXEL3` variants of all the `PIXEL0` macros. For example:

```c
CYG_FB_PIXEL0_VAR(FRAMEBUF);
CYG_FB_PIXEL1_VAR(FRAMEBUF);

CYG_FB_PIXEL0_SET(FRAMEBUF, dest_x, dest_y_);
CYG_FB_PIXEL1_SET(FRAMEBUF, source_x, source_y);
```
Framebuffer Pixel Manipulation

```c
for (rows = height; rows; rows--) {
    for (columns = width; columns; columns--) {
        colour = CYG_FB_PIXEL1_READ(FRAMEBUF);
        <do some processing on colour>
        CYG_FB_PIXEL0_WRITE(FRAMEBUF, colour);
        CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
        CYG_FB_PIXEL1_ADDX(FRAMEBUF, 1);
    }
    CYG_FB_PIXEL0_ADDX(FRAMEBUF, -100);
    CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
    CYG_FB_PIXEL1_ADDX(FRAMEBUF, -100);
    CYG_FB_PIXEL1_ADDY(FRAMEBUF, 1);
}

CYG_FB_PIXEL0_FLUSHABS(FRAMEBUF, source_x, source_y, width, height);
CYG_FB_PIXEL1_FLUSHABS(FRAMEBUF, 0, 0, 0, 0); // Only used for reading
```

The `PIXEL0`, `PIXEL1`, `PIXEL2` and `PIXEL3` macros all use different local variables so there are no conflicts. The variable names also depend on the framebuffer device. If the target has two displays and two active framebuffer devices then the pixel macros can be used with the two devices without conflict:

```c
CYG_FB_PIXEL0_VAR(FRAMEBUF0);
CYG_FB_PIXEL0_VAR(FRAMEBUF1);
...```
Writing a Framebuffer Device Driver

Name
Porting — writing a new framebuffer device driver

Description
As with most device drivers, the easiest way to write a new framebuffer package is to start with an existing one. Suitable ones include the PC VGA mode13 driver, an 8bpp paletted display, and the ARM iPAQ driver, a 16bpp true colour display. This document only outlines the process.

Before writing any code it is necessary to decide how many framebuffer devices should be provided by the device driver. Each such device requires a cyg_fb structure and appropriate functions, and an identifier for use with the macro API plus associated macros. There are no hard rules here. Some device drivers may support just a single device, others may support many devices which drive the hardware in different modes or orientations. Optional functionality such as viewports and page flipping may be supported by having different cyg_fb devices, or by a number of configuration options which affect a single cyg_fb device. Usually providing multiple cyg_fb structures is harmless because the unused ones will get eliminated at link-time.

Configuration
The CDL for a framebuffer package is usually straightforward. A framebuffer package should be a hardware package and reside in the devs/framebuf hierarchy, further organized by architecture. Generic framebuffer packages, if any, can go into a generic subdirectory, and will normally rely on the platform HAL to provide some platform-specific information such as base addresses. The package should be part of the target definition and hence loaded automatically, but should be active_if CYGPKG_IO_FRAMEBUF so that the driver only gets built if the generic framebuffer support is explicitly added to the configuration.

The configuration option CYGDAT_IO_FRAMEBUF_DEVICES should hold all the valid identifiers which can be used as the first argument for the macro API. This helps application developers to select the appropriate identifier, and allows higher-level graphics library packages to check that they have been configured correctly. This is achieved using something like the following, where mode13_320x200x8 is a valid identifier for the PC VGA driver:

```c
requires { is_substr(CYGDAT_IO_FRAMEBUF_DEVICES, " mode13_320x200x8 ") }
```

The spaces ensure that the CDL inference engine keeps the identifiers separate.

CYGPKG_IO_FRAMEBUF contains a number of interfaces which should be implemented by individual device drivers when appropriate. This is used to eliminate some code or data structure fields at compile-time, keeping down memory requirements. The interfaces are CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_32BPP, CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_TRUE_COLOUR, CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_PALETTE, CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_WRITEABLE_PALETTE, CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_DOUBLE_BUFFER, and CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT. For example if a device driver provides a true colour display but fails to implement the relevant interface then functions like cyg_fb_make_colour will be no-ops.

Device drivers for paletted displays should observe the generic configuration option CYGFUN_IO_FRAMEBUF_INSTALL_DEFAULT_PALETTE and install either cyg_fb_palette_ega or cyg_fb_palette_vga as part of their cyg_fb_on implementation.
Exported Header File(s)

Each framebuffer device driver should export one or more header files to cyg/io/framebufs. A custom build step in CYGPKG_IO_FRAMEBUF ensures that application code can just include cyg/io/framebuf.h and this will automatically include the device-specific headers. Drivers may export one header per cyg_fb device or a single header for all devices, without affecting any code outside the device driver.

Each exported header serves two purposes. First it defines the parameters, drawing primitive macros, and iteration macros for each device. Second it declares the cyg_fb structure.

Parameters

The parameter section should resemble the following:

```c
#define CYG_FB_320x240x16_STRUCT cyg_ipaq_fb_320x240x16
#define CYG_FB_320x240x16_DEPTH 16
#define CYG_FB_320x240x16_FORMAT CYG_FB_FORMAT_16BPP_TRUE_565
#define CYG_FB_320x240x16_WIDTH 320
#define CYG_FB_320x240x16_HEIGHT 240
#define CYG_FB_320x240x16_VIEWPORT_WIDTH 320
#define CYG_FB_320x240x16_VIEWPORT_HEIGHT 240
#define CYG_FB_320x240x16_FLAGS0 (CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER | CYG_FB_FLAGS0_TRUE_COLOUR | CYG_FB_FLAGS0_BLANK | CYG_FB_FLAGS0_BACKLIGHT)
#define CYG_FB_320x240x16_FLAGS1 0
#define CYG_FB_320x240x16_FLAGS2 0
#define CYG_FB_320x240x16_FLAGS3 0
#define CYG_FB_320x240x16_BASE ((void *)0x01FC0020)
#define CYG_FB_320x240x16_STRIDE 640
```

Here 320x240x16 is the framebuffer identifier for use with the macro API. Application code like:

```c
#define FRAMEBUF 320x240x16
cyg_u16 width = CYG_FB_WIDTH(FRAMEBUF);
```

will end up using the `CYG_FB_320x240x16_WIDTH` definition. To allow for efficient portable code all parameters must be compile-time constants. If the hardware may allow some of the parameters to be varied, for example different resolutions, then this should be handled either by defining separate devices for each resolution or by configuration options.

The viewport width and height should always be defined. If the device driver does not support a viewport then these will be the same as the standard width and height.

To allow for future expansion there are `FLAGS1`, `FLAGS2` and `FLAGS3` parameters. No flags are defined for these at present, but device drivers should still define the parameters.

Drawing Primitives

For each device the exported header file should define macros for the drawing primitives, using the same naming convention as for parameters. In the case of true colour displays there should also be macros for the make-colour and break-colour primitives:

```c
#define CYG_FB_320x240x16_WRITE_PIXEL(_x_, _y_, _colour_) ...
#define CYG_FB_320x240x16_READ_PIXEL(_x_, _y_) ...
```
Writing a Framebuffer Device Driver

#define CYG_FB_320x240x16_WRITE_HLINE(_x_, _y_, _len_, _colour_) ...  
#define CYG_FB_320x240x16_WRITE_VLINE(_x_, _y_, _len_, _colour_) ...  
#define CYG_FB_320x240x16_FILL_BLOCK(_x_, _y_, _w_, _h_, _colour_) ...  
#define CYG_FB_320x240x16_WRITE_BLOCK(_x_, _y_, _w_, _h_, _data_, _off_, _s_) ...  
#define CYG_FB_320x240x16_READ_BLOCK(_x_, _y_, _w_, _h_, _data_, _off_, _s_) ...  
#define CYG_FB_320x240x16_MOVE_BLOCK(_x_, _y_, _w_, _h_, _new_x_, _new_y_) ...  
#define CYG_FB_320x240x16_MAKE_COLOUR(_r_, _g_, _b_) ...  
#define CYG_FB_320x240x16_BREAK_COLOUR(_colour_, _r_, _g_, _b_) ...  

For typical linear framebuffers there are default implementations of all of these primitives in the generic framebuffer package, held in the exported header cyg/io/framebuf.inl. Hence the definitions will typically look something like:

#include <cyg/io/framebuf.inl>
... 
#define CYG_FB_320x240x16_WRITE_PIXEL(_x_, _y_, _colour_)  
 CYG_MACRO_START  
 cyg_fb_linear_write_pixel_16_inl(CYG_FB_320x240x16_BASE,  
 CYG_FB_320x240x16_STRIDE,  
 _x_, _y_, _colour_);  
 CYG_MACRO_END  
#define CYG_FB_320x240x16_READ_PIXEL(_x_, _y_)  
 ({ cyg_fb_linear_read_pixel_16_inl(CYG_FB_320x240x16_BASE,  
 CYG_FB_320x240x16_STRIDE,  
 _x_, _y_); }) 
...  

All of the drawing primitives have variants for the common display depths and layouts: 1le, 1be, 2le, 2be, 4le, 4be, 8, 16 and 32. The inlines take the framebuffer memory base address as the first argument, and the stride in bytes as the second. Similarly there are default definitions of the true colour primitives for 8BPP_TRUE_332, 16BPP_TRUE_565, 16BPP_TRUE_555, and 32BPP_TRUE_0888:

#define CYG_FB_320x240x16_MAKE_COLOUR(_r_, _g_, _b_)  
 ({ CYG_FB_MAKE_COLOUR_16BPP_TRUE_565(_r_, _g_, _b_); })  
#define CYG_FB_320x240x16_BREAK_COLOUR(_colour_, _r_, _g_, _b_)  
 CYG_MACRO_START  
 CYG_FB_BREAK_COLOUR_16BPP_TRUE_565(_colour_, _r_, _g_, _b_);  
 CYG_MACRO_END  

These default definitions assume the most common layout of colours within a pixel value, so for example CYG_FB_MAKE_COLOUR_16BPP_TRUE_565 assumes bits 0 to 4 hold the blue intensity, bits 5 to 10 the green, and bits 11 to 15 the red.

If the hardware does not implement a linear framebuffer then obviously writing the device driver will be significantly more work. The macros will have to perform the operations themselves instead of relying on generic implementations. The required functionality should be obvious, and the generic implementations can still be consulted as a reference. For complicated hardware it may be appropriate to map the macros onto function calls, rather than try to implement everything inline.

Note: At the time of writing the support for linear framebuffers is incomplete. Only 8bpp, 16bpp and 32bpp depths have full support. There may also be future extensions, for example r90, r180 and r270 variants to support rotation in software, and db variants to support double-buffered displays.

287
**Iteration Macros**

In addition to the drawing primitives the exported header file should define iteration macros:

```c
#define CYG_FB_320x240x16_PIXELx_VAR( _fb_, _id_) ...
#define CYG_FB_320x240x16_PIXELx_SET( _fb_, _id_, _x_, _y_) ...
#define CYG_FB_320x240x16_PIXELx_GET( _fb_, _id_, _x_, _y_) ...
#define CYG_FB_320x240x16_PIXELx_ADDX( _fb_, _id_, _incr_) ...
#define CYG_FB_320x240x16_PIXELx_ADDY( _fb_, _id_, _incr_) ...
#define CYG_FB_320x240x16_PIXELx_WRITE(_fb_, _id_, _colour_) ...
#define CYG_FB_320x240x16_PIXELx_READ( _fb_, _id_)...
#define CYG_FB_320x240x16_PIXELx_FLUSHABS( _fb_, _id_, _x0_, _y0_, _w_, _h_) ...
#define CYG_FB_320x240x16_PIXELx_FLUSHREL( _fb_, _id_, _x0_, _y0_, _dx_, _dy_) ...
```

The `fb` argument will be the identifier, in this case 320x240x16, and the `id` will be a small number, 0 for a PIXEL0 iteration, 1 for PIXEL1, and so on. Together these two should allow unique local variable names to be constructed. Again there are default definitions of the macros in `cyg/io/framebuf.inl` for linear framebuffers:

```c
#define CYG_FB_320x240x16_PIXELx_VAR( _fb_, _id_) 
   CYG_FB_PIXELx_VAR_16( _fb_, _id_)
#define CYG_FB_320x240x16_PIXELx_SET( _fb_, _id_, _x_, _y_) 
   CYG_MACRO_START 
   CYG_FB_PIXELx_SET_16( _fb_, _id_, 
   CYG_FB_320x240x16_BASE, 
   320, _x_, _y_); 
   CYG_MACRO_END
```

The linear `set` and `get` macros take base and stride information. The `addx` and `addy` macros only need the stride. By convention most of the macros are wrapped in `CYG_MACRO_START/CYG_MACRO_END` or `{/}` pairs, allowing debug code to be inserted if necessary. However the `_VAR` macro must not be wrapped in this way: its purpose is to define one or more local variables; wrapping the macro would declare the variables in a new scope, inaccessible to the other macros.

Again for non-linear framebuffers it will be necessary to implement these macros fully rather than rely on generic implementations, but the generic versions can be consulted as a reference.

**The cyg_fb declaration**

Finally there should be an export of the cyg_fb structure or structures. Typically this uses the `_STRUCT` parameter, reducing the possibility of an accidental mismatch between the macro and function APIs:

```c
extern cyg_fb CYG_FB_320x240x16_STRUCT;
```

**Driver-Specific Source Code**

Exporting parameters and macros in a header file is not enough. It is also necessary to actually define the cyg_fb structure or structures, and to provide hardware-specific versions of the control operations. For non-linear framebuffers it will also be necessary to provide the drawing functions. There is a utility macro `CYG_FB_FRAMEBUFFER` for instantiating a cyg_fb structure. Drivers may ignore this macro and do the work themselves, but at an increased risk of compatibility problems with future versions of the generic code.
The first 13 arguments to the macro correspond to the device parameters. The next four are arbitrary CYG_ADDRWORD values for use by the device driver. Typically these are used to share on/off/ioctl functions between multiple cyg_fb structure. They are followed by function pointers: on/off/ioctl control; double buffer synch; palette management; true colour support; and the drawing primitives. nop versions of the on, off, ioctl, synch, palette management and true colour functions are provided by the generic framebuffer package, and often these arguments to the CYG_FB_FRAMEBUFFER macro will be discarded at compile-time because the relevant CDL interface is not implemented. The final four arguments are currently unused and should be 0. They are intended for future expansion, with a value of 0 indicating that a device driver does not implement non-core functionality.

As with the macros there are default implementations of the true colour primitives for 8bpp_true_332, 16bpp_true_565, 16bpp_true_555 and 32bpp_true_0888, assuming the most common layout for these colour modes. There are also default implementations of the drawing primitives for linear framebuffers, with variants for the common display depths and layouts. Obviously non-linear framebuffers will need rather more work.

Typically a true colour or grey scale framebuffer device driver will have to implement just three hardware-specific functions:

```c
int cyg_ipaq_fb_on(cyg_fb* fb)
{
    ... 
}
```
int
cyg_ipaq_fb_off(cyg_fb* fb)
{
    ...
}

int
cyg_ipaq_fb_ioctl(cyg_fb* fb, cyg_ucount16 key, void* data, size_t* len)
{
    int result;

    switch(key) {
        case CYG_FB_IOCTL_BLANK_GET: ...
        ...
        default: result = ENOSYS; break;
    }
    return result;
}

These control operations are entirely hardware-specific and cannot be implemented by generic code. Paletted
displays will need two more functions, again hardware-specific:

void
cyg_pcvga_fb_read_palette(cyg_fb* fb, cyg_ucount32 first, cyg_ucount32 len,
    void* data)
{
    ...
}

void
cyg_pcvga_fb_write_palette(cyg_fb* fb, cyg_ucount32 first, cyg_ucount32 len,
    const void* data, cyg_ucount16 when)
{
    ...
}

Future Expansion

As has been mentioned before framebuffer hardware varies widely. The design of a generic framebuffer API
requires complicated trade-offs between efficiency, ease of use, ease of porting, and still supporting a very
wide range of hardware. To some extent this requires a lowest common denominator approach, but the design
allows for some future expansion and optional support for more advanced features like hardware acceleration.
The most obvious route for expansion is the ioctl interface. Device drivers can define their own keys, values
0x8000 and higher, for any operation. Alternatively a device driver does not have to implement just the interface
provided by the generic framebuffer package: additional functions and macros can be exported as required.
Currently there are only a small number of ioctl operations. Additional ones may get added in future, for
example to support a hardware mouse cursor, but only in cases where the functionality is likely to be provided
by a significant number of framebuffer devices. Adding new generic functionality adds to the maintenance
overhead of both code and documentation. When a new generic ioctl operation is added there will usually
also be one or more new flags, so that device drivers can indicate they support the functionality. At the time of
writing only 12 of the 32 FLAGS0 flags are used, and a further 96 are available in FLAGS1, FLAGS2 and FLAGS3.
Another route for future expansion is the four spare arguments to the CYG_FB_FRAMEBUFFER macro. As an example of how these may get used in future, consider support for 3D hardware acceleration. One of the spare fields would become another table of function pointers to the various accelerators, or possibly a structure. A FLAGS0 flag would indicate that the device driver implements such functionality.

Other forms of expansion such as defining a new standard drawing primitive would be more difficult, since this would normally involve changing the CYG_FB_FRAMEBUFFER macro. Such expansion should not be necessary because the existing primitives provide all reasonable core functionality. Instead other packages such as graphics libraries can work on top of the existing primitives.
XIII. eCos POSIX compatibility layer
Chapter 35. POSIX Standard Support

eCos contains support for the POSIX Specification (ISO/IEC 9945-1)[POSIX].

POSIX support is divided between the POSIX and the FILEIO packages. The POSIX package provides support for threads, signals, synchronization, timers and message queues. The FILEIO package provides support for file and device I/O. The two packages may be used together or separately, depending on configuration.

This document takes a functional approach to the POSIX library. Support for a function implies that the data types and definitions necessary to support that function, and the objects it manipulates, are also defined. Any exceptions to this are noted, and unless otherwise noted, implemented functions behave as specified in the POSIX standard.

This document only covers the differences between the eCos implementation and the standard; it does not provide complete documentation. For full information, see the POSIX standard [POSIX]. Online, the Open Group Single Unix Specification [SUS2] provides complete documentation of a superset of POSIX. If you have access to a Unix system with POSIX compatibility, then the manual pages for this will be of use. There are also a number of books available. [Lewine] covers the process, signal, file and I/O functions, while [Lewis1], [Lewis2], [Nichols] and [Norton] cover Pthreads and related topics (see Bibliography, xref). However, many of these books are oriented toward using POSIX in non-embedded systems, so care should be taken in applying them to programming under eCos.

The remainder of this chapter broadly follows the structure of the POSIX Specification. References to the appropriate section of the Standard are included.

Omitted functions marked with “// TBA” are potential candidates for later implementation.

Process Primitives [POSIX Section 3]

Functions Implemented

int kill(pid_t pid, int sig);
int pthread_kill(pthread_t thread, int sig);
int sigaction(int sig, const struct sigaction *act,
               struct sigaction *oact);
int sigqueue(pid_t pid, int sig, const union sigval value);
int sigprocmask(int how, const sigset_t *set,
                 sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *set);
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
unsigned int alarm(unsigned int seconds);
int pause(void);
unsigned int sleep(unsigned int seconds);
Functions Omitted

pid_t fork(void);
int execl( const char *path, const char *arg, ... );
int execv( const char *path, char *const argv[] );
int execlp( const char *path, const char *arg, ... );
int execve( const char *path, char *const argv[],
            char *const envp[] );
int pthread_atfork( void (*)(prepare)(void),
                    void (*)(parent)(void),
                    void (*)(child)() );
pid_t wait( int *stat_loc );
pid_t waitpid( pid_t pid, int *stat_loc,
              int options );
void _exit( int status );

Notes

- Signal handling may be enabled or disabled with the CYGPKG_POSIX_SIGNALS option. Since signals
  are used by other POSIX components, such as timers, disabling signals will disable those components too.
- `kill()` and `sigqueue()` may only take a `pid` argument of zero, which maps to the current process.
- The `SIGEV_THREAD` notification type is not currently implemented.
- Job Control and Memory Protection signals are not supported.
- An extra implementation defined `si_code` value, `SI_EXCEPT`, is defined to distinguish hardware generated
  exceptions from others.
- Extra signals are defined: `_SIGTRAP_`, `_SIGIOT_`, `_SIGEMT_`, and `_SIGSYS_`. These are largely to main-
  tain compatibility with the signal numbers used by GDB.
- Signal delivery may currently occur at unexpected places in some API functions. Using `longjmp()` to transfer
  control out of a signal handler may result in the interrupted function not being able to complete properly. This
  may result in later function calls failing or deadlocking.

Process Environment [POSIX Section 4]

Functions Implemented

int uname( struct utsname *name );
time_t time( time_t *tloc );
char *getenv( const char *name );
int isatty( int fd );
long sysconf( int name );
Functions Omitted

pid_t getpid( void );
pid_t getppid( void );
uid_t getuid( void );
uid_t geteuid( void );
gid_t getgid( void );
gid_t getegid( void );
int setuid( uid_t uid );
int setgid( gid_t gid );
int getgroups( int gidsetsize, gid_t grouplist[] );
char *getlogin( void );
int getlogin_r( char *name, size_t namesize );
pid_t getpgrp( void );
pid_t setsid( void );
int setpgid( pid_t pid, pid_t pgid );
char *ctermid( char *s );
char *ttyname( int fd ); // TBA
int ttyname_r( int fd, char *name, size_t namesize ); // TBA
clock_t times( struct tms *buffer ); // TBA

Notes

• The fields of the utsname structure are initialized as follows:

  sysname “eCos”
nodename “” (gethostname() is currently not available)

  release Major version number of the kernel
  version Minor version number of the kernel
  machine “” (Requires some config tool changes)

  The sizes of these strings are defined by CYG_POSIX_UTSNAME_LENGTH and
  CYG_POSIX_UTSNAME_NODENAME_LENGTH. The latter defaults to the value of the former, but
  may also be set independently to accommodate a longer node name.

• The time() function is currently implemented in the C library.

• A set of environment strings may be defined at configuration time with the
  CYGDAT_LIBC_DEFAULT_ENVIRONMENT option. The application may also define an environment
  by direct assignment to the environ variable.

• At present isatty() assumes that any character device is a tty and that all other devices are not ttys. Since the
  only kind of device that eCos currently supports is serial character devices, this is an adequate distinction.

• All system variables supported by sysconf will yield a value. However, those that are irrelevant to eCos will
  either return the default minimum defined in <limits.h>, or zero.

Files and Directories [POSIX Section 5]

Functions Implemented

DIR *opendir( const char *dirname );
Chapter 35. POSIX Standard Support

```
struct dirent *readdir( DIR *dirp );
int readdir_r( DIR *dirp, struct dirent *entry,
              struct dirent ***result );
void rewinddir( DIR *dirp );
int closedir( DIR *dirp );
int chdir( const char *path );
char *getcwd( char *buf, size_t size );
int open( const char *path, int oflag, ... );
int creat( const char *path, mode_t mode );
int link( const char *existing, const char *new );
int mkdir( const char *path, mode_t mode );
int unlink( const char *path );
int rmdir( const char *path );
int rename( const char *old, const char *new );
int stat( const char *path, struct stat *buf );
int fstat( int fd, struct stat *buf );
int access( const char *path, int amode );
long pathconf(const char *path, int name);
long fpathconf(int fd, int name);

Functions Omitted

mode_t umask( mode_t cmask );
int mkfifo( const char *path, mode_t mode );
int chmod( const char *path, mode_t mode ); // TBA
int fchmod( int fd, mode_t mode ); // TBA
int chown( const char *path, uid_t owner, gid_t group );
int utime( const char *path, const struct utimbuf *times ); // TBA
int ftruncate( int fd, off_t length ); // TBA
```

Notes

- If a call to open() or creat() supplies the third _mode_ parameter, it will currently be ignored.
- Most of the functionality of these functions depends on the underlying filesystem.
- Currently access() only checks the F_OK mode explicitly, the others are all assumed to be true by default.
- The maximum number of open files allowed is supplied by the CYGNUM_FILEIO_NFILE option. The maximum number of file descriptors is supplied by the CYGNUM_FILEIO_NFD option.

Input and Output [POSIX Section 6]

```
Functions Implemented

int dup( int fd );
int dup2( int fd, int fd2 );
int close( int fd );
ssize_t read( int fd, void *buf, size_t nbyte );
ssize_t write( int fd, const void *buf, size_t nbyte );
int fcntl( int fd, int cmd, ... );
```
off_t lseek(int fd, off_t offset, int whence);
int fsync(int fd);
int fdatasync(int fd);

**Functions Omitted**

int pipe(int fildes[2]);
int aio_read(struct aiocb *aiocbp);  // TBA
int aio_write(struct aiocb *aiocbp);  // TBA
int lio_listio(int mode, struct aiocb *const list[],
  int nent, struct sigevent *sig);  // TBA
int aio_error(struct aiocb *aiocbp);  // TBA
int aio_return(struct aiocb *aiocbp);  // TBA
int aio_cancel(int fd, struct aiocb *aiocbp);  // TBA
int aio_suspend(const struct aiocb *const list[],
  int nent, const struct timespec *timeout);  // TBA
int aio_fsync(int op, struct aiocb *aiocbp);  // TBA

**Notes**

- Only the F_DUPFD command of fcntl() is currently implemented.
- Most of the functionality of these functions depends on the underlying filesystem.

**Device and Class Specific Functions [POSIX Section 7]**

**Functions Implemented**

speed_t cfgetospeed(const struct termios *termios_p);
int cfsetospeed(struct termios *termios_p, speed_t speed);
speed_t cfsetisp( struct termios *termios_p, speed_t speed );
int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions,
const struct termios *termios_p);
int tcgetpgrp(int fd, int duration);
int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcsendbreak(int fd, int queue_selector);

**Functions Omitted**

pid_t tcgetpgrp(int fd);
int tcsetpgrp(int fd, pid_t pgrp);
Chapter 35. POSIX Standard Support

Notes

- Only the functionality relevant to basic serial device control is implemented. Only very limited support for canonical input is provided, and then only via the “tty” devices, not the “serial” devices. None of the functionality relevant to job control, controlling terminals and sessions is implemented.
- Only \( MIN = 0 \) and \( TIME = 0 \) functionality is provided.
- Hardware flow control is supported if the underlying device driver and serial port support it.
- Support for break, framing and parity errors depends on the functionality of the hardware and device driver.

C Language Services [POSIX Section 8]

Functions Implemented

```c
char *setlocale( int category, const char *locale );
int fileno( FILE *stream );
FILE *fdopen( int fd, const char *type );
int getc_unlocked( FILE *stream );
int getchar_unlocked( void );
int putc_unlocked( FILE *stream );
int putchar_unlocked( void );
char *strtok_r( char *s, const char *sep, char ***lasts );
char *asctime_r( const struct tm *tm, char *buf );
char *ctime_r( const time_t *clock, char *buf );
struct tm *gmtime_r( const time_t *clock, struct tm *result );
struct tm *localtime_r( const time_t *clock, struct tm *result );
int rand_r( unsigned int *seed );
```

Functions Omitted

```c
void flockfile( FILE *file );
int ftrylockfile( FILE *file );
void funlockfile( FILE *file );
int sigsetjmp( sigjmp_buf env, int savemask ); // TBA
void siglongjmp( sigjmp_buf env, int val ); // TBA
void tzset(void); // TBA
```

Notes

- `setlocale()` is implemented in the C library Internationalization package.
- Functions `fileno()` and `fdopen()` are implemented in the C library STDIO package.
- Functions `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()` and `putchar_unlocked()` are defined but are currently identical to their non-unlocked equivalents.
• `strtok_r()`, `asctime_r()`, `ctime_r()`, `gmtime_r()`, `localtime_r()` and `rand_r()` are all currently in the C library, alongside their non-reentrant versions.

System Databases [POSIX Section 9]

Functions Implemented

• <none>

Functions Omitted

```c
struct group *getgrgid( gid_t gid );
int getgrgid( gid_t gid, struct group *grp, char *buffer,
             size_t bufsize, struct group **result );
struct group *getgrname( const char *name );
int getgrname( const char *name, struct group *grp,
              char *buffer, size_t bufsize, struct group **result );
struct passwd *getpwnam( const char *name );
int getpwnam_r( const char *name, struct passwd **result );
struct passwd *getpwnam( const char *name );
int getpwnam_r( const char *name, struct passwd **pwd,
                char *buffer, size_t bufsize, struct passwd **result );
```

Notes

• None of the functions in this section are implemented.

Data Interchange Format [POSIX Section 10]

This section details `tar` and `cpio` formats. Neither of these is supported by eCos.

Synchronization [POSIX Section 11]

Functions Implemented

```c
int sem_init( sem_t *sem, int pshared, unsigned int value );
int sem_destroy( sem_t *sem );
int sem_wait( sem_t *sem );
int sem_trywait( sem_t *sem );
int sem_post( sem_t *sem );
int sem_getvalue( sem_t *sem, int *sval );
int pthread_mutexattr_init( pthread_mutexattr_t *attr );
```
Chapter 35. POSIX Standard Support

```c
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutex_init(pthread_mutex_t *mutex,
   const pthread_mutexattr_t *mutex_attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_cond_init(pthread_cond_t *cond,
   const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
   pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
   pthread_mutex_t *mutex,
   const struct timespec *abstime);
```

Functions Omitted

```c
sem_t *sem_open(const char *name, int oflag, ...); // TBA
int sem_close(sem_t *sem); // TBA
int sem_unlink(const char *name); // TBA
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
   int *pshared);
int pthread_mutexattr_setpshared(const pthread_mutexattr_t *attr,
   int pshared);
int pthread_condattr_getpshared(const pthread_condattr_t *attr,
   int *pshared);
int pthread_condattr_setpshared(const pthread_condattr_t *attr,
   int pshared);
```

Notes

- The presence of semaphores is controlled by the CYGPKG_POSIX_SEMAPHORES option. This in turn causes the _POSIX_SEMAPHORES feature test macro to be defined and the semaphore API to be made available.
- The `pshared` argument to `sem_init()` is not implemented, its value is ignored.
- Functions `sem_open()`, `sem_close()` and `sem_unlink()` are present but always return an error (ENOSYS).
- The exact priority inversion protocols supported may be controlled with the `_POSIX_THREAD_PRIO_INHERIT` and `_POSIX_THREAD_PRIO_PROTECT` configuration options.
- `{_POSIX_THREAD_PROCESS_SHARED}` is not defined, so the process-shared mutex and condition variable attributes are not supported, and neither are the functions `pthread_mutexattr_getpshared()`, `pthread_mutexattr_setpshared()`, `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()`.
- Condition variables do not become bound to a particular mutex when `pthread_cond_wait()` is called. Hence different threads may wait on a condition variable with different mutexes. This is at variance with the standard, which requires a condition variable to become (dynamically) bound by the first waiter, and unbound
when the last finishes. However, this difference is largely benign, and the cost of policing this feature is non-trivial.

Memory Management [POSIX Section 12]

Functions Implemented

<none>

Functions Omitted

int mlockall( int flags );
int munlockall( void );
int mlock( const void *addr, size_t len );
int munlock( const void *addr, size_t len );
void mmap( void *addr, size_t len, int prot, int flags,
int fd, off_t off );
int munmap( void *addr, size_t len );
int mprotect( const void *addr, size_t len, int prot );
int msync( void *addr, size_t len, int flags );
int shm_open( const char *name, int oflag, mode_t mode );
int shm_unlink( const char *name );

Notes

None of these functions are currently provided. Some may be implemented in a restricted form in the future.

Execution Scheduling [POSIX Section 13]

Functions Implemented

int sched_yield(void);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int pthread_get priority_max(int policy);
int pthread_get priority_min(int policy);
int pthread_get priority_max(int policy);
int pthread_get priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
int pthread_get_priority_max(int policy);
int pthread_get_priority_min(int policy);
Chapter 35. POSIX Standard Support

```c
struct sched_param *param);
int pthread_mutexattr_setprotocol( pthread_mutexattr_t *attr,
    int protocol);
int pthread_mutexattr_getprotocol( pthread_mutexattr_t *attr,
    int *protocol);
int pthread_mutexattr_setprioceiling( pthread_mutexattr_t *attr,
    int prioceiling);
int pthread_mutexattr_getprioceiling( pthread_mutexattr_t *attr,
    int *prioceiling);
int pthread_mutex_setprioceiling( pthread_mutex_t *mutex,
    int prioceiling,
    int *old_ceiling);
int pthread_mutex_getprioceiling( pthread_mutex_t *mutex,
    int *prioceiling);
```

### Functions Omitted

```c
int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

### Notes

- The functions `sched_setparam()`, `sched_getparam()`, `sched_setscheduler()` and `sched_getscheduler()` are present but always return an error.
- The scheduler policy `SCHED_OTHER` is equivalent to `SCHED_RR`.
- Only `PTHREAD_SCOPE_SYSTEM` is supported as a `contentionscope` attribute.
- The default thread scheduling attributes are:

  ```c
  contentionscope PTHREAD_SCOPE_SYSTEM
  inheritsched PTHREAD_INHERIT_SCHED
  schedpolicy SCHED_OTHER
  schedparam.sched 0
  ```

- Mutex priority inversion protection is controlled by a number of kernel configuration options. If `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT` is defined then `{ _POSIX_THREAD_PRIO_INHERIT }` will be defined and `PTHREAD_PRIO_INHERIT` may be set as the protocol in a `pthread_mutexattr_t` object. If `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING` is defined then `{ _POSIX_THREAD_PRIO_PROTECT }` will be defined and `PTHREAD_PRIO_PROTECT` may be set as the protocol in a `pthread_mutexattr_t` object.
- The default attribute values set by `pthread_mutexattr_init()` is to set the protocol attribute to `PTHREAD_PRIO_NONE` and the prioceiling attribute to zero.
Clocks and Timers [POSIX Section 14]

Functions Implemented

- `int clock_settime(clockid_t clock_id, const struct timespec *tp);`
- `int clock_gettime(clockid_t clock_id, struct timespec *tp);`
- `int clock_getres(clockid_t clock_id, struct timespec *tp);`
- `int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timer_id);`
- `int timer_delete(timer_t timer_id);`
- `int timer_settime(timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue);`
- `int timer_gettime(timer_t timerid, struct itimerspec *value);`
- `int timer_getoverrun(timer_t timerid);`
- `int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);`
- `int gettimeofday(struct timeval *tv, struct timezone* tz);`

Functions Omitted

- `<none>`

Notes

- Currently `timer_getoverrun()` only reports timer notifications that are delayed in the timer subsystem. If they are delayed in the signal subsystem, due to signal masks for example, this is not counted as an overrun.
- The option CYGPKG_POSIX_TIMERS allows the timer support to be enabled or disabled, and causes _POSIX_TIMERS to be defined appropriately. This will cause other parts of the POSIX system to have limited functionality.

Message Passing [POSIX Section 15]

Functions Implemented

- `mqd_t mq_open(const char *name, int oflag, ... );`
- `int mq_close(mqd_t mqdes);`
- `int mq_unlink(const char *name);`
- `int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);`
- `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);`
- `int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat);`
- `int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);`
- `int mq_notify(mqd_t mqdes, const struct sigevent *notification);`
Chapter 35. POSIX Standard Support

From POSIX 1003.1d draft:

```c
int mq_send( mqd_t mqdes, const char *msg_ptr,  
    size_t msg_len, unsigned int msg_prio,  
    const struct timespec *abs_timeout );
ssize_t mq_receive( mqd_t mqdes, char *msg_ptr,  
    size_t msg_len, unsigned int *msg_prio,  
    const struct timespec *abs_timeout );
```

Functions Omitted

<none>

Notes

- The presence of message queues is controlled by the CYGPKG_POSIX_MQUEUES option. Setting this will cause [_POSIX_MESSAGE_PASSING] to be defined and the message queue API to be made available.
- Message queues are not currently filesystem objects. They live in their own name and descriptor spaces.

Thread Management [POSIX Section 16]

Functions Implemented

```c
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setdetachstate(pthread_attr_t *attr,  
    int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,  
    int *detachstate);
int pthread_attr_setstackaddr(pthread_attr_t *attr,  
    void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr,  
    void **stackaddr);
int pthread_attr_setstacksize(pthread_attr_t *attr,  
    size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,  
    size_t *stacksize);
int pthread_create( pthread_t *thread,  
    const pthread_attr_t *attr,  
    void *(*start_routine)(void *),  
    void *arg);
pthread_t pthread_self( void );
int pthread_equal(pthread_t thread1, pthread_t thread2);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **thread_return);
int pthread_detach(pthread_t thread);
int pthread_once(pthread_once_t *once_control,  
    void (*init_routine)(void));
```
Chapter 35. POSIX Standard Support

Functions Omitted

<none>

Notes

• The presence of thread support as a whole is controlled by the the CYGPKG_POSIX_PTHREAD configuration option. Note that disabling this will also disable many other features of the POSIX package, since these are intimately bound up with the thread mechanism.

• The default (non-scheduling) thread attributes are:

  - detachstate: PTHREAD_CREATE_JOINABLE
  - stackaddr: unset
  - stacksize: unset

• Dynamic thread stack allocation is only provided if there is an implementation of malloc() configured (i.e. a package implements the CYGINT_MEMALLOC_MALLOCALLOCATORS interface). If there is no malloc() available, then the thread creator must supply a stack. If only a stack address is supplied then the stack is assumed to be PTHREAD_STACK_MIN bytes long. This size is seldom useful for any but the most trivial of threads. If a different sized stack is used, both the stack address and stack size must be supplied.

• The value of PTHREAD_THREADS_MAX is supplied by the CYGNUM_POSIX_PTHREAD_THREADS_MAX option. This defines the maximum number of threads allowed. The POSIX standard requires this value to be at least 64, and this is the default value set.

• When the POSIX package is installed, the thread that calls main() is initialized as a POSIX thread. The priority of that thread is controlled by the CYGNUM_POSIX_MAIN_DEFAULT_PRIORITY option.

Thread-Specific Data [POSIX Section 17]

Functions Implemented

```c
int pthread_key_create(pthread_key_t *key, 
    void *(*destructor)(void *));
int pthread_setspecific(pthread_key_t key, const void *pointer);
void *pthread_getspecific(pthread_key_t key);
int pthread_key_delete(pthread_key_t key);
```

Functions Omitted

<none>
Notes

- The value of `PTHREAD_DESTRUCTOR_ITERATIONS` is provided by the `CYGNUM_POSIX_PTHREAD_DESTRUCTOR_ITERATIONS` option. This controls the number of times that a key destructor will be called while the data item remains non-NULL.
- The value of `PTHREAD_KEYS_MAX` is provided by the `CYGNUM_POSIX_PTHREAD_KEYS_MAX` option. This defines the maximum number of per-thread data items supported. The POSIX standard calls for this to be a minimum of 128, which is rather large for an embedded system. The default value for this option is set to 128 for compatibility but it should be reduced to a more usable value.

Thread Cancellation [POSIX Section 18]

Functions Implemented

```c
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
void pthread_cleanup_push( void (*routine)(void *), void *arg);
void pthread_cleanup_pop( int execute);
```

Functions Omitted

```c
<none>
```

Notes

Asynchronous cancellation is only partially implemented. In particular, cancellation may occur in unexpected places in some functions. It is strongly recommended that only synchronous cancellation be used.

Non-POSIX Functions

In addition to the standard POSIX functions defined above, the following non-POSIX functions are defined in the FILEIO package.

General I/O Functions

```c
int ioctl( int fd, CYG_ADDRWORD com, CYG_ADDRWORD data );
int select( int nfd, fd_set *in, fd_set *out, fd_set *ex, struct timeval *tv);
```
Socket Functions

int socket(int domain, int type, int protocol);
int bind(int s, const struct sockaddr *sa, unsigned int len);
int listen(int s, int len);
int accept(int s, struct sockaddr *sa, socklen_t *addrlen);
int connect(int s, const struct sockaddr *sa, socklen_t len);
int getpeername(int s, struct sockaddr *sa, socklen_t *len);
int getsockname(int s, struct sockaddr *sa, socklen_t *len);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                  struct sockaddr *from, socklen_t *fromlen);
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t sendto(int s, const void *buf, size_t len, int flags,
                const struct sockaddr *to, socklen_t tolen);
ssize_t send(int s, const void *buf, size_t len, int flags);
int shutdown(int s, int how);

Notes

• The precise behaviour of these functions depends mainly on the functionality of the underlying filesystem or network stack to which they are applied.
References and Bibliography


XIV. µITRON
Chapter 36. \(\mu ITRON\) API

Introduction to \(\mu ITRON\)

The \(\mu ITRON\) specification defines a highly flexible operating system architecture designed specifically for application in embedded systems. The specification addresses features which are common to the majority of processor architectures and deliberately avoids virtualization which would adversely impact real-time performance. The \(\mu ITRON\) specification may be implemented on many hardware platforms and provides significant advantages by reducing the effort involved in understanding and porting application software to new processor architectures.

Several revisions of the \(\mu ITRON\) specification exist. In this release, \(eCos\) supports the \(\mu ITRON\) version 3.02 specification, with complete “Standard functionality” (level S), plus many “Extended” (level E) functions. The definitive reference on \(\mu ITRON\) is Dr. Sakamura’s book: \(\mu ITRON\) 3.0, An Open and Portable Real-Time Operating System for Embedded Systems. The book can be purchased from the IEEE Press, and an ASCII version of the standard can be found online at http://www.itron.gr.jp/. The old address http://tron.unu-tokyo.ac.jp/TRON/ITRON/ still exists as a mirror site.

\(\mu ITRON\) and \(eCos\)

The \(eCos\) kernel implements the functionality used by the \(\mu ITRON\) compatibility subsystem. The configuration of the kernel influences the behavior of \(\mu ITRON\) programs.

In particular, the default configuration has time slicing (also known as round-robin scheduling) switched on; this means that a task can be moved from \textit{RUN} state to \textit{READY} state at any time, in order that one of its peers may run. This is not strictly conformant to the \(\mu ITRON\) specification, which states that timeslicing may be implemented by periodically issuing a \texttt{rot_rdq(0)} call from within a periodic task or cyclic handler; otherwise it is expected that a task runs until it is pre-empted in consequence of synchronization or communication calls it makes, or the effects of an interrupt or other external event on a higher priority task cause that task to become \textit{READY}. To disable timeslicing functionality in the kernel and \(\mu ITRON\) compatibility environment, please disable the \texttt{CYGSEMKERNEL_SCHED_TIMESLICE} configuration option in the kernel package. A description of kernel scheduling is in \textit{Kernel Overview}.

For another example, the semantics of task queueing when waiting on a synchronization object depend solely on the way the underlying kernel is configured. As discussed above, the multi-level queue scheduler is the only one which is \(\mu ITRON\) compliant, and it queues waiting tasks in FIFO order. Future releases of that scheduler might be configurable to support priority ordering of task queues. Other schedulers might be different again: for example the bitmap scheduler can be used with the \(\mu ITRON\) compatibility layer, even though it only allows one task at each priority and as such is not \(\mu ITRON\) compliant, but it supports only priority ordering of task queues. So which queueing scheme is supported is not really a property of the \(\mu ITRON\) compatibility layer; it depends on the kernel.

In this version of the \(\mu ITRON\) compatibility layer, the calls to disable and enable scheduling and interrupts (\texttt{dis_dsp()}, \texttt{ena_dsp()}, \texttt{loc_cpu()} and \texttt{unl_cpu()}) call underlying kernel functions; in particular, the \texttt{xxx_dsp()} functions lock the scheduler entirely, which prevents dispatching of DSRs; functions implemented by DSRs include clock counters and alarm timers. Thus time “stops” while dispatching is disabled with \texttt{dis_dsp()}.

Like all parts of the \(eCos\) system, the detailed semantics of the \(\mu ITRON\) layer are dependent on its configuration and the configuration of other components that it uses. The \(\mu ITRON\) configuration options are all defined in
the file pkgconf/uitron.h, and can be set using the configuration tool or editing the .ecc file in your build directory by hand.

An important configuration option for the µITRON compatibility layer is "Option: Return Error Codes for Bad Params" (CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS), which allows a lot of the error checking code in the µITRON compatibility layer to be removed. Of course this leaves a program open to undetected errors, so it should only be used once an application is fully debugged and tested. Its benefits include reduced code size and faster execution. However, it affects the API significantly, in that with this option enabled, bad calls do not return errors, but cause an assert failure (if that is itself enabled) or malfunction internally. There is discussion in more detail about this in each section below.

We now give a brief description of the µITRON functions which are implemented in this release. Note that all C and C++ source files should have the following #include statement:

```c
#include <cyg/comapt/uitron/uit_func.h>
```

## Task Management Functions

The following functions are fully supported in this release:

```c
ER sta_tsk(  
    ID tskid,  
    INT stacd )

void ext_tsk( void )

void exd_tsk( void )

ER dis_dsp( void )

ER ena_dsp( void )

ER chg_pri(  
    ID tskid,  
    PRI tskpri )

ER rot_rdq(  
    PRI tskpri )

ER get_tid(  
    ID *p_tskid )

ER ref_tsk(  
    T_RTSK *pk_rtsk,  
    ID tskid )

ER ter_tsk(  
    ID tskid )

ER rel_wai(  
    ID tskid )
```

The following two functions are supported in this release, when enabled with the configuration option CYGPKG_UITRON_TASKS_CREATE_DELETE with some restrictions:

```c
ER cre_tsk(  
    ID tskid,  
    T_CTSK *pk_ctsk )
```
These functions are restricted as follows:

Because of the static initialization facilities provided for system objects, a task is allocated stack space statically in the configuration. So while tasks can be created and deleted, the same stack space is used for that task (task ID number) each time. Thus the stack size (pk_cstk->stksz) requested in \texttt{cre_tsk()} is checked for being less than that which was statically allocated, and otherwise ignored. This ensures that the new task will have enough stack to run. For this reason \texttt{del_tsk()} does not in any sense free the memory that was in use for the task’s stack.

The task attributes (pk_cstk->tskatr) are ignored; current versions of eCos do not need to know whether a task is written in assembler or C/C++ so long as the procedure call standard appropriate to the CPU is followed.

Extended information (pk_cstk->exinf) is ignored.

### Error checking

For all these calls, an invalid task id (tskid) (less than 1 or greater than the number of configured tasks) only returns E_ID when bad params return errors (CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled, see above).

Similarly, the following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled:

- \texttt{pk_crtk} in \texttt{cre_tsk()} is a valid pointer, otherwise return E_PAR
- \texttt{ter_tsk()} or \texttt{rel_wai()} on the calling task returns E_OBJ
- the CPU is not locked already in \texttt{dis_dsp()} and \texttt{ena_dsp()} : returns E_CTX
- priority level in \texttt{chg_pri()} and \texttt{rot_rdq()} is checked for validity, E_PAR
- return value pointer in \texttt{get_tid()} and \texttt{ref_tsk()} is a valid pointer, or E_PAR

The following conditions are checked for, and return error codes if appropriate, regardless of the setting of CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS:

- When create and delete functions \texttt{cre_tsk()} and \texttt{del_tsk()} are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
- When supported, \texttt{cre_tsk()} : the task must not already exist; otherwise E_OBJ
- When supported, \texttt{cre_tsk()} : the requested stack size must not be larger than that statically configured for the task; see the configuration options “Static initializers”, and “Default stack size”. Else E_NOMEM
- When supported, \texttt{del_tsk()} : the underlying eCos thread must not be running - this would imply either a bug or some program bypassing the \texttt{µITRON} compatibility layer and manipulating the thread directly. E_OBJ
- \texttt{sta_tsk()} : the task must be dormant, else E_OBJ
- \texttt{ter_tsk()} : the task must not be dormant, else E_OBJ
- \texttt{chg_pri()} : the task must not be dormant, else E_OBJ
- \texttt{rel_wai()} : the task must be in \texttt{WAIT} or \texttt{WAIT-SUSPEND} state, else E_OBJ
Chapter 36. µITRON API

Task-Dependent Synchronization Functions

These functions are fully supported in this release:

ER sus_tsk{
    ID tskid }
ER rsm_tsk{
    ID tskid }
ER frsm_tsk{
    ID tskid }
ER slp_tsk{ void }
ER tslp_tsk{
    TMO tmout }
ER wup_tsk{
    ID tskid }
ER can_wup{
    INT *p_wupcnt, ID tskid }

Error checking

The following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled (see the configuration option “Return Error Codes for Bad Params”):

• invalid tskid; less than 1 or greater than CYGNUM_UITRON_TASKS returns E_ID
• wup_tsk(), sus_tsk(), rsm_tsk(), frsm_tsk() on the calling task returns E_OBJ
• dispatching is enabled in tslp_tsk() and slp_tsk(), or E_CTX
• tmout must be positive, otherwise E_PAR
• return value pointer in can_wup() is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS:

• When create and delete functions cre_tsk() and del_tsk() are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
• sus_tsk(): the task must not be dormant, else E_OBJ
• frsm/rsm_tsk(): the task must be suspended, else E_OBJ
• tslp/slp_tsk(): return codes E_TMOOUT, E_RLWAI and E_DLT are returned depending on the reason for terminating the sleep
• wup_tsk() and can_wup(): the task must not be dormant, or E_OBJ is returned
Synchronization and Communication Functions

These functions are fully supported in this release:

ER sig_sem(
    ID semid )

ER wai_sem(
    ID semid )

ER preq_sem(
    ID semid )

ER twai_sem(
    ID semid,     TMO tmout )

ER ref_sem(
    T_RSEM *pk_rsem ,     ID semid )

ER set_flg(
    ID flgid,     UINT setptn )

ER cle_flg(
    ID flgid,     UINT clrptn )

ER wai_flg(
    UINT *p_flgptn,     ID flgid ,
    UINT waipntn,     UINT wfmode )

ER pol_flg(
    UINT *p_flgptn,     ID flgid ,
    UINT waipntn,     UINT wfmode )

ER twai_flg(
    UINT *p_flgptn ID flgid ,
    UINT waipntn,     UINT wfmode,    TMO tmout )

ER ref_flg(
    T_RFLG *pk_rflg,     ID flgid )

ER snd_msg(
    ID mbxid,     T_MSG *pk_msg )

ER rcv_msg(
    T_MSG **ppk_msg,     ID mbxid )

ER prcv_msg(
    T_MSG **ppk_msg,     ID mbxid )

ER trcv_msg(
    T_MSG **ppk_msg,     ID mbxid ,     TMO tmout )

ER ref_mbx(
    T_RMBX *pk_rmbx,     ID mbxid )

The following functions are supported in this release (with some restrictions) if enabled with the appropriate configuration option for the object type (for example CYGPKG_UITRON_SEMAS_CREATE_DELETE):

ER cre_sem(
    ID semid,     T_CSEM *pk_csem )

ER del_sem{
In general the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict \( \mu \)ITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (\( pk\_cxxx\_atrat = TA\_TPRI \)) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (\( pk\_cxxx\_exinf \)) is ignored.

For semaphores, the initial semaphore count (\( pk\_csem\_isemcnt \)) is supported; the new semaphore’s count is set. The maximum count is not supported, and is not in fact defined in type \( pk\_csem \).

For flags, multiple tasks are allowed to wait. Because single task waiting is a subset of this, the W bit (TA_WMUL) of the flag attributes is ignored; all other bits must be zero. The initial flag value is supported.

For mailboxes, the buffer count is defined statically by kernel configuration option \( CYGNUM\_KERNEL\_SYNCH\_MBOX\_QUEUE\_SIZE \); therefore the buffer count field is not supported and is not in fact defined in type \( pk\_cmbx \). Queueing of messages is FIFO ordered only, so TA_MPRI (in \( pk\_cmbx\_mbxatr \)) is not supported.

## Error checking

The following conditions are only checked for, and only return errors if \( CYGSEM\_UITRON\_BAD\_PARAMS\_RETURN\_ERRORS \) is enabled:

- invalid object id; less than 1 or greater than \( CYGNUM\_UITRON\_TASKS/SEMAS/MBOXES \) as appropriate
- E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- \( pk\_cxxx \) pointers in \( cre\_xxx() \) must be valid pointers, or E_PAR
- return value pointer in \( ref\_xxx() \) is valid pointer, or E_PAR
- flag wait pattern must be non-zero, and mode must be valid, or E_PAR
- return value pointer in flag wait calls is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of \( CYGSEM\_UITRON\_BAD\_PARAMS\_RETURN\_ERRORS \):

- When create and delete functions \( cre\_xxx() \) and \( del\_xxx() \) are supported, all calls which use a valid object ID number check that the object exists. If not, E_NOEXS is returned.
- In create functions \( cre\_xxx() \), when supported, if the object already exists, then E_OBJ
In any call which can sleep, such as `twai_sem()` return codes `E_TMOUT`, `E_RLWAI`, `E_DLT` or of course `E_OK` are returned depending on the reason for terminating the sleep.

In polling functions such as `preq_sem()` return codes `E_TMOUT` or `E_OK` are returned depending on the state of the synchronization object.

In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in `cre_sem()` `pk_csem->sematr` must be equal to `TA_TFIFO` else `E_RSATR`.

In `cre_flg()` `pk_cflg->flgatr` must be either `TA_WMUL` or `TA_WSGL` else `E_RSATR`.

In `cre_mbx()` `pk_cmbx->mbxatr` must be `TA_TFIFO` + `TA_MFIFO` else `E_RSATR`.

### Extended Synchronization and Communication Functions

None of these functions are supported in this release.

### Interrupt management functions

These functions are fully supported in this release:

```c
void ret_int( void )
ER loc_cpu( void )
ER unl_cpu( void )
ER dis_int( UINT eintno )
ER ena_int( UINT eintno )
void ret_wup( ID tskid )
ER iwup_tsk( ID tskid )
ER isig_sem( ID semid )
ER iset_flg( ID flgid, UID setptn )
ER isend_msg( ID mbxid, T_MSG *pk_msg )
```

Note that `ret_int()` and the `ret_wup()` are implemented as macros, containing a “return” statement.

Also note that `ret_wup()` and the `ixxx_yyy()` style functions will only work when called from an ISR whose associated DSR is `cyg_uitron_dsr()`, as specified in include file `<cyg/compat/uitron/uit_ifnc.h>`, which defines the `ixxx_yyy()` style functions also.
If you are writing interrupt handlers more in the eCos style, with separate ISR and DSR routines both of your own devising, do not use these special functions from a DSR: use plain `xxx_yyy()` style functions (with no ‘i’ prefix) instead, and do not call any µITRON functions from the ISR at all.

The following functions are not supported in this release:

- `ER def_int(int dintno, T_DINT *pk_dint)`
- `ER chg_iXX(UINT iXXXX)`
- `ER ref_iXX(UINT *p_iXXXX)`

These unsupported functions are all Level C (CPU dependent). Equivalent functionality is available via other eCos-specific APIs.

### Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- `loc/unl_cpu()`: these must only be called in a µITRON task context, else E_CTX.
- `dis/ena_int()`: the interrupt number must be in range as specified by the platform HAL in question, else E_PAR.

### Memory pool Management Functions

These functions are fully supported in this release:

- `ER get_blf(VP *p_blf, ID mpfid)`
- `ER pget_blf(VP *p_blf, ID mpfid)`
- `ER tget_blf(VP *p_blf, ID mpfid, TMO tmout)`
- `ER rel_blf(ID mpfid, VP blf)`
- `ER ref_mpf(T_RMPF *pk_rmpf, ID mpfid)`
- `ER get_bik(VP *p_bik, ID mplid, INT biksz)`
- `ER pget_bik(VP *p_bik, ID mplid, INT biksz)`
- `ER tget_bik(VP *p_bik, ID mplid, INT biksz)`
Chapter 36. µITRON API

VP *p_blk, ID mplid, INT blksz, TMO tmout )

ER rel_blk{
    ID mplid, VP blk }

ER ref_mpl{
    T_RMPL *pk_rmpl, ID mplid }

Note that of the memory provided for a particular pool to manage in the static initialization of the memory pool objects, some memory will be used to manage the pool itself. Therefore the number of blocks * the blocksize will be less than the total memory size.

The following functions are supported in this release, when enabled with CYGPKG_UITRON_MEMPOOLVAR_CREATE_DELETE or CYGPKG_UITRON_MEMPOOLFIXED_CREATE_DELETE as appropriate, with some restrictions:

ER cre_mpl{
    ID mplid, T_CMPL *pk_cmpl }

ER del_mpl{
    ID mplid }

ER cre_mpf{
    ID mpfid, T_CMPF *pk_cmpf }

ER del_mpf{
    ID mpfid }

Because of the static initialization facilities provided for system objects, a memory pool is allocated a region of memory to manage statically in the configuration. So while memory pools can be created and deleted, the same area of memory is used for that memory pool (memory pool ID number) each time. The requested variable pool size (pk_cmpl->mplsz) or the number of fixed-size blocks (pk_cmpf->mpfcnt) times the block size (pk_cmpf->blfsz) are checked for fitting within the statically allocated memory area, so if a create call succeeds, the resulting pool will be at least as large as that requested. For this reason del_mpl() and del_mpf() do not in any sense free the memory that was managed by the deleted pool for use by other pools; it may only be managed by a pool of the same object id.

For both fixed and variable memory pools, the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict µITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (pk_cxxx->xxxatr = TA_TPRI) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (pk_cxxx->exinf) is ignored.

Error checking

The following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled:

- invalid object id; less than 1 or greater than CYGNUM_UITRON_MEMPOOLVAR/MEMPOOLFIXED as appropriate returns E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- pk_cxxx pointers in cre_xxx() must be valid pointers, or E_PAR
Chapter 36. \( \mu \)ITRON API

- return value pointer in \texttt{ref_xxx()} is a valid pointer, or \texttt{E\_PAR}
- return value pointers in get block routines is a valid pointer, or \texttt{E\_PAR}
- blocksize request in get variable block routines is greater than zero, or \texttt{E\_PAR}

The following conditions are checked for, and can return error codes, regardless of the setting of \texttt{CYGSEM\_UITRON\_BAD\_PARAMS\_RETURN\_ERRORS}:

- When create and delete functions \texttt{cre_xxx()} and \texttt{del_xxx()} are supported, all calls which use a valid object ID number check that the object exists. If not, \texttt{E\_NOEXS} is returned.
- When create functions \texttt{cre_xxx()} are supported, if the object already exists, then \texttt{E\_OBJ}
- In any call which can sleep, such as \texttt{get_blk()} : return codes \texttt{E\_TMOOUT, E\_RLWAI, E\_DLT} or of course \texttt{E\_OK} are returned depending on the reason for terminating the sleep
- In polling functions such as \texttt{pget_blk()} return codes \texttt{E\_TMOOUT or E\_OK} are returned depending on the state of the synchronization object
- In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in \texttt{cre_mpl()} \texttt{pk\_cmpl\_mplstr} must be equal to \texttt{TA\_TFIFO} else \texttt{E\_RSATR}.
- In \texttt{cre_mpf()} \texttt{pk\_cmpf\_mpfcnt} must be equal to \texttt{TA\_TFIFO} else \texttt{E\_RSATR}.
- In creation functions, the requested size of the memory pool must not be larger than that statically configured for the pool else \texttt{E\_RSATR}; see the configuration option “Option: Static initializers”. In \texttt{cre_mpl()} \texttt{pk\_cmpl\_mplsz} is the field of interest
- In \texttt{cre_mpf()} the product of \texttt{pk\_cmpf\_blfsz} and \texttt{pk\_cmpf\_mpfcnt} must be smaller than the memory statically configured for the pool else \texttt{E\_RSATR}
- In functions which return memory to the pool \texttt{rel_blk()} and \texttt{rel_blf()}, if the free fails, for example because the memory did not come from that pool originally, then \texttt{E\_PAR} is returned.

### Time Management Functions

These functions are fully supported in this release:

**Caution**

Setting the time may cause erroneous operation of the kernel, for example a task performing a wait with a time-out may never awaken.

\begin{verbatim}
ER set_tim(  
    SYSTIME *pk_tim )

ER get_tim(  
    SYSTIME *pk_tim )

ER dly_tsk(  
    DLYTIME dlytim )

ER def_cyc(  
    HNO cycno,  
    T\_DCYC *pk\_dcyc )

ER act_cyc(  
    HNO cycno,  
    UINT cycact )
\end{verbatim}
Chapter 36. µITRON API

Error checking

The following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled:

- invalid handler number; less than 1 or greater than CYGNUM_UITRON_CYCLICS/ALARMS as appropriate, or E_PAR
- dispatching is enabled in dly_tsk(), or E_CTX
- dlytim must be positive or zero, otherwise E_PAR
- return value pointer in ref_xxx() is a valid pointer, or E_PAR
- params within pk_dalm and pk_dcyc must be valid, or E_PAR
- cycact in act_cyc() must be valid, or E_PAR
- handler must be defined in ref_xxx() and act_cyc(), or E_NOEXS
- parameter pointer must be a good pointer in get_tim() and set_tim(), otherwise E_PAR is returned

The following conditions are checked for, and can return error codes, regardless of the setting of CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS:

- dly_tsk(): return code E_RLWAI is returned depending on the reason for terminating the sleep

System Management Functions

These functions are fully supported in this release:

ER get_ver(
    T_VER *pk_ver )

ER ref_sys(
    T_RSYS *pk_rsys )

ER ref_cfg(
    T_RCFG *pk_rcfg )

Note that the information returned by each of these calls may be configured to match the user’s own versioning system, and the values supplied by the default configuration may be inappropriate.

These functions are not supported in this release:

ER def_svc(
    FN s_fncd,
Chapter 36. µITRON API

```c
T_DSVC *pk_dsvc )

ER def_exc( 
    UINT exckind, 
    T_DEXC *pk_dexc )
```

### Error checking

The following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled:

- return value pointer in all calls is a valid pointer, or E_PAR

### Network Support Functions

None of these functions are supported in this release.

### µITRON Configuration FAQ

**Q: How are µITRON objects created?**

For each type of µITRON object (tasks, semaphores, flags, mboxes, mpf, mpl) these two quantities are controlled by configuration:

- The maximum number of this type of object.
- The number of these objects which exist initially.

This is assuming that for the relevant object type, create and delete operations are enabled; enabled is the default. For example, the option CYGPKG_UITRON_MBOXES_CREATE_DELETE controls whether the functions cre_mbx() and del_mbx() exist in the API. If not, then the maximum number of mboxes is the same as the initial number of mboxes, and so on for all µITRON object types.

Mboxes have no initialization, so there are only a few, simple configuration options:

- CYGNUM_UITRON_MBOXES is the total number of mboxes that you can have in the system. By default this is 4, so you can use mboxes 1, 2, 3 and 4. You cannot create mboxes outside this range; trying to cre_mbx(5, ...) will return an error.
- CYGNUM_UITRON_MBOXESINITIALLY is the number of mboxes created automatically for you, during startup. By default this is 4, so all 4 mboxes exist already, and an attempt to create one of these eg. cre_mbx(3, ...) will return an error because the mbox in question already exists. You can delete a pre-existing mbox, and then re-create it.

If you change CYGNUM_UITRON_MBOXESINITIALLY, for example to 0, no mboxes are created automatically for you during startup. Any attempt to use an mbox without creating it will return E_NOEXS because the mbox does not exist. You can create an mbox, say cre_mbx(3, ...) and then use it, say snd_msg(3, &foo), and all will be well.

**Q: How are µITRON objects initialized?**
Some object types have optional initialization. Semaphores are an example. You could have 
\texttt{CYGNUM\_UITRON\_SEMAS=10} and \texttt{CYGNUM\_UITRON\_SEMAS\_INITIALLY=5} which means you can use semaphores 1-5 straight off, but you must create semaphores 6-10 before you can use them. If you decide not to initialize semaphores, semaphores 1-5 will have an initial count of zero. If you decide to initialize them, you must supply a dummy initializer for semaphores 6-10 also. For example, in terms of the configuration output in \texttt{pkgconf/uitron.h}:

\begin{verbatim}
#define CYGDAT_UITRON_SEMA_INITIALIZERS 
  CYG_UIT_SEMA( 1 ), \n  CYG_UIT_SEMA( 0 ), \n  CYG_UIT_SEMA( 0 ), \n  CYG_UIT_SEMA( 99 ), \n  CYG_UIT_SEMA( 1 ), \n  CYG_UIT_SEMA_NOEXS, \n  CYG_UIT_SEMA_NOEXS, \n  CYG_UIT_SEMA_NOEXS, \n  CYG_UIT_SEMA_NOEXS
\end{verbatim}

Semaphore 1 will have initial count 1, semaphores 2 and 3 will be zero, number 4 will be 99 initially, 5 will be one and numbers 6 through 10 do not exist initially.

Aside: this is how the definition of the symbol would appear in the configuration header file \texttt{pkgconf/uitron.h} — unfortunately editing such a long, multi-line definition is somewhat cumbersome in the GUI config tool in current releases. The macros \texttt{CYG\_UIT\_SEMA()} — to create a semaphore initializer — and \texttt{CYG\_UIT\_SEMA\_NOEXS} — to invoke a dummy initializer — are provided in in the environment to help with this. Similar macros are provided for other object types. The resulting \#define symbol is used in the context of a C++ array initializer, such as:

\begin{verbatim}
Cyg\_Counting\_Semaphore2 cyg\_uitron\_SEMAS[ CYGNUM\_UITRON\_SEMAS ] = {
  CYGDAT\_UITRON\_SEMA\_INITIALIZERS
};
\end{verbatim}

which is eventually macro-processed to give

\begin{verbatim}
Cyg\_Counting\_Semaphore2 cyg\_uitron\_SEMAS[ 10 ] = {
  Cyg\_Counting\_Semaphore2( ( 1 ) ),
  Cyg\_Counting\_Semaphore2( ( 0 ) ),
  Cyg\_Counting\_Semaphore2( ( 0 ) ),
  Cyg\_Counting\_Semaphore2( ( 99 ) ),
  Cyg\_Counting\_Semaphore2( ( 1 ) ),
  Cyg\_Counting\_Semaphore2(0),
  Cyg\_Counting\_Semaphore2(0),
  Cyg\_Counting\_Semaphore2(0),
  Cyg\_Counting\_Semaphore2(0),
};
\end{verbatim}

so you can see how it is necessary to include the dummy entries in that definition, otherwise the resulting code will not compile correctly.

If you choose \texttt{CYGNUM\_UITRON\_SEMAS\_INITIALLY=0} it is meaningless to initialize them, for they must be created and so initialized then, before use.

\textit{Q: What about \mu\textsc{TRON} tasks?}

Some object types require initialization. Tasks are an example of this. You must provide a task with a priority, a function to enter when the task starts, a name (for debugging purposes), and some memory to use for the stack. For example (again in terms of the resulting definitions in \texttt{pkgconf/uitron.h}):
Chapter 36. µITRON API

#define CYGNUM_UITRON_TASKS 4 // valid task ids are 1,2,3,4
#define CYGNUM_UITRON_TASKS_INITIALIZALLY 4 // they all exist at start

#define CYGDAT_UITRON_TASK_EXTERNS
extern "C" void startup( unsigned int );
extern "C" void worktask( unsigned int );
extern "C" void lowtask( unsigned int );
static char stack1[ CYGNUM_UITRON_STACK_SIZE ],
  stack2[ CYGNUM_UITRON_STACK_SIZE ],
  stack3[ CYGNUM_UITRON_STACK_SIZE ],
  stack4[ CYGNUM_UITRON_STACK_SIZE ];

#define CYGDAT_UITRON_TASK_INITIALIZERS
  CYG_UIT_TASK("main task", 8, startup, &stack1, sizeof( stack1 ) ),
  CYG_UIT_TASK("worker 2", 9, worktask, &stack2, sizeof( stack2 ) ),
  CYG_UIT_TASK("worker 3", 9, worktask, &stack3, sizeof( stack3 ) ),
  CYG_UIT_TASK("low task", 20, lowtask, &stack4, sizeof( stack4 ) );

So this example has all four tasks statically configured to exist, ready to run, from the start of time. The “main task” runs a routine called startup() at priority 8. Two “worker” tasks run both a priority 9, and a “low priority” task runs at priority 20 to do useful non-urgent background work.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Exists at Startup</th>
<th>Function</th>
<th>Priority</th>
<th>Stack Stack address</th>
<th>Stack size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>startup</td>
<td>8</td>
<td>&amp;stack1 CYGNUM...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>worktask</td>
<td>9</td>
<td>&amp;stack2 CYGNUM...</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>worktask</td>
<td>9</td>
<td>&amp;stack3 CYGNUM...</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>lowtask</td>
<td>20</td>
<td>&amp;stack4 CYGNUM...</td>
<td></td>
</tr>
</tbody>
</table>

Q: How can I create µITRON tasks in the program?

You must provide free slots in the task table in which to create new tasks, by configuring the number of tasks existing initially to be smaller than the total. For a task ID which does not initially exist, it will be told what routine to call, and what priority it is, when the task is created. But you must still set aside memory for the task to use for its stack, and give it a name during initialization. For example:

#define CYGNUM_UITRON_TASKS 4 // valid task ids are 1-4
#define CYGNUM_UITRON_TASKS_INITIALIZALLY 1 // only task #1 exists

#define CYGDAT_UITRON_TASK_EXTERNS
extern "C" void startup( unsigned int );
static char stack1[ CYGNUM_UITRON_STACK_SIZE ],
  stack2[ CYGNUM_UITRON_STACK_SIZE ],
  stack3[ CYGNUM_UITRON_STACK_SIZE ],
  stack4[ CYGNUM_UITRON_STACK_SIZE ];

#define CYGDAT_UITRON_TASK_INITIALIZERS
  CYG_UIT_TASK( "main", 8, startup, &stack1, sizeof( stack1 ) ),
  CYG_UIT_TASK_NOEXS( "slave", &stack2, sizeof( stack2 ) ),
  CYG_UIT_TASK_NOEXS( "slave2", &stack3, sizeof( stack3 ) ),
  CYG_UIT_TASK_NOEXS( "slave3", &stack4, sizeof( stack4 ) );

So tasks numbered 2,3 and 4 have been given their stacks during startup, though they do not yet exist in terms of cre_tsk() and del_tsk() so you can create tasks 2–4 at runtime.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Exists at Startup</th>
<th>Function</th>
<th>Priority</th>
<th>Stack Stack address</th>
<th>Stack size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 36. µITRON API

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>startup</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>startup</td>
<td>8</td>
<td>stack1</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>stack2</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>stack3</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>stack4</td>
</tr>
</tbody>
</table>

(you must have at least one task at startup in order that the system can actually run; this is not so for other uITRON object types)

**Q**: Can I have different stack sizes for µITRON tasks?

Simply set aside different amounts of memory for each task to use for its stack. Going back to a typical default setting for the µITRON tasks, the definitions in pkgconf/uitron.h might look like this:

```c
#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void task1( unsigned int ); \
extern "C" void task2( unsigned int ); \
extern "C" void task3( unsigned int ); \
extern "C" void task4( unsigned int ); \
static char stack1[ CYGNUM_UITRON_STACK_SIZE ], \
         stack2[ CYGNUM_UITRON_STACK_SIZE ], \
         stack3[ CYGNUM_UITRON_STACK_SIZE ], \
         stack4[ CYGNUM_UITRON_STACK_SIZE ];
#define CYGDAT_UITRON_TASK_INITIALIZERS \
CYG_UIT_TASK( "t1", 1, task1, &stack1, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK( "t2", 2, task2, &stack2, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK( "t3", 3, task3, &stack3, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK( "t4", 4, task4, &stack4, CYGNUM_UITRON_STACK_SIZE )
```

Note that CYGNUM_UITRON_STACK_SIZE is used to control the size of the stack objects themselves, and to tell the system what size stack is being provided.

Suppose instead stack sizes of 2000, 1000, 800 and 800 were required: this could be achieved by using the GUI config tool to edit these options, or editing the .ecc file to get these results in pkgconf/uitron.h:

```c
#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void task1( unsigned int ); \
extern "C" void task2( unsigned int ); \
extern "C" void task3( unsigned int ); \
extern "C" void task4( unsigned int ); \
static char stack1[ 2000 ], \
         stack2[ 1000 ], \
         stack3[ 800 ], \
         stack4[ 800 ];
#define CYGDAT_UITRON_TASK_INITIALIZERS \
CYG_UIT_TASK( "t1", 1, task1, &stack1, sizeof( stack1 ) ), \
CYG_UIT_TASK( "t2", 2, task2, &stack2, sizeof( stack2 ) ), \
CYG_UIT_TASK( "t3", 3, task3, &stack3, sizeof( stack3 ) ), \
CYG_UIT_TASK( "t4", 4, task4, &stack4, sizeof( stack4 ) )
```

Note that the sizeof() operator has been used to tell the system what size stacks are provided, rather than quoting a number (which is difficult for maintenance) or the symbol CYGNUM_UITRON_STACK_SIZE (which is wrong).

We recommend using (if available in your release) the stacksize symbols provided in the architectural HAL for your target, called CYGNUM_HAL_STACK_SIZE_TYPICAL and CYGNUM_HAL_STACK_SIZE_MINIMUM. So a better (more portable) version of the above might be:

```c
#define CYGDAT_UITRON_TASK_EXTERNS \
```
extern "C" void task1(unsigned int); \
extern "C" void task2(unsigned int); \
extern "C" void task3(unsigned int); \
extern "C" void task4(unsigned int); \
static char stack1[ CYGNUM_HAL_STACK_SIZE_TYPICAL + 1200 ], \
    stack2[ CYGNUM_HAL_STACK_SIZE_TYPICAL + 200 ], \
    stack3[ CYGNUM_HAL_STACK_SIZE_TYPICAL ], \
    stack4[ CYGNUM_HAL_STACK_SIZE_TYPICAL ];

#define CYGDAT_UITRON_TASK_INITIALIZERS
    CYG_UIT_TASK( "t1", 1, task1, &stack1, sizeof( stack1 ) ), \n    CYG_UIT_TASK( "t2", 2, task2, &stack2, sizeof( stack2 ) ), \n    CYG_UIT_TASK( "t3", 3, task3, &stack3, sizeof( stack3 ) ), \n    CYG_UIT_TASK( "t4", 4, task4, &stack4, sizeof( stack4 ) )
XV. TCP/IP Stack Support for eCos

The Common Networking for eCos package provides support for a complete TCP/IP networking stack. The design allows for the actual stack to be modular and at the current time two different implementations, one based on OpenBSD from 2000 and a new version based on FreeBSD, are available. The particulars of each stack implementation are presented in separate sections following this top-level discussion.
Chapter 37. Ethernet Driver Design

Currently, the networking stack only supports ethernet based networking.

The network drivers use a two-layer design. One layer is hardware independent and contains all the stack specific code. The other layer is platform dependent and communicates with the hardware independent layer via a very simple API. In this way, hardware device drivers can actually be used with other stacks, if the same API can be provided by that stack. We designed the drivers this way to encourage the development of other stacks in eCos while allowing re-use of the actual hardware specific code.

More comprehensive documentation of the ethernet device driver and the associated API can be found in the generic ethernet device driver documentation Part XXI in eCos Reference Manual The driver and API is the same as the minimal debug stack used by the RedBoot application. See the RedBoot documentation for further information.
Chapter 38. Sample Code

Many examples using the networking support are provided. These are arranged as eCos test programs, primarily for use in verifying the package, but they can also serve as useful frameworks for program design. We have taken a KISS approach to building programs which use the network. A single include file `<network.h>` is all that is required to access the stack. A complete, annotated test program can be found at `net/common/VERSION/tests/ftp_test.c`, with its associated files.
Chapter 39. Configuring IP Addresses

Each interface (“eth0” and “eth1”) has independent configuration of its setup. Each can be set up manually (in which case you must write code to do this), or by using BOOTP/DHCP, or explicitly, with configured values. If additional interfaces are added, these must be configured manually.

The configurable values are:

- IP address
- netmask
- broadcast address
- gateway/router
- server address.

Server address is the DHCP server if applicable, but in addition, many test cases use it as “the machine to talk to” in whatever manner the test exercises the protocol stack.

The initialization is invoked by calling the C routine

```c
void init_all_network_interfaces(void);
```

Additionally, if the system is configured to support IPv6 then each interface may have an address assigned which is a composite of a 64 bit prefix and the 32 bit IPv4 address for that interface. The prefix is controlled by the CDL setting CYGHWR_NET_DRIVER_ETH0_IPV6_PREFIX for “eth0”, etc. This is a CDL booldata type, allowing this address to be suppressed if not desired.

Alternatively, the system can configure its IPv6 address using router solicitation. When the CDL option CYGOPT_NET_IPV6_ROUTING_THREAD is enabled, `init_all_network_interface` will start a thread which sends out router solicit messages, process router advertisements and thus configure an IPv6 address to the interface.

Refer to the test cases, `.../packages/net/common/VERSION/tests/ftp_test.c` for example usage, and the source files in `.../packages/net/common/VERSION/src/bootp_support.c` and `network_support.c` to see what that call does.

This assumes that the MAC address (also known as ESA or Ethernet Station Address) is already defined in the serial EEPROM or however the particular target implements this; support for setting the MAC address is hardware dependent.

DHCP support is active by default, and there are configuration options to control it. Firstly, in the top level of the “Networking” configuration tree, “Use full DHCP instead of BOOTP” enables DHCP, and it contains an option to have the system provide a thread to renew DHCP leases and manage lease expiry. Secondly, the individual interfaces “eth0” and “eth1” each have new options within the “Use BOOTP/DHCP to initialize ‘ethX’” to select whether to use DHCP rather than BOOTP.

Note that you are completely at liberty to ignore this startup code and its configuration in building your application. `init_all_network_interfaces()` is provided for three main purposes:

- For use by Red Hat’s own test programs.
- As an easy “get you going” utility for newcomers to eCos.
If your application has different requirements for bringing up available network interfaces, setting up routes, determining IP addresses and the like from the defaults that the example code provides, you can write your own initialization code to use whatever sequence of `ioctl()` function calls carries out the desired setup. Analogously, in larger systems, a sequence of “ifconfig” invocations is used; these mostly map to `ioctl()` calls to manipulate the state of the interface in question.
Chapter 40. Tests and Demonstrations

Loopback tests

By default, only tests which can execute on any target will be built. These therefore do not actually use external network interfaces (though they may configure and initialize them) but are limited to testing via the loopback interface.

ping_lo_test - ping test of the loopback address
tcp_lo_select - simple test of select with TCP via loopback
tcp_lo_test - trivial TCP test via loopback
udp_lo_test - trivial UDP test via loopback
multi_lo_select - test of multiple select() calls simultaneously

Building the Network Tests

To build further network tests, ensure that the configuration option CYGPKG_NET_BUILD_TESTS is set in your build and then make the tests in the usual way. Alternatively (with that option set) use

make -C net/common/VERSION/ tests

after building the eCos library, if you wish to build only the network tests.

This should give test executables in install/tests/net/common/VERSION/tests including the following:

socket_test - trivial test of socket creation API
mbuf_test - trivial test of mbuf allocation API
ftp_test - simple FTP test, connects to "server"
ing_test - pings "server" and non-existent host to test timeout
dhcp_test - ping test, but also relinquishes and reacquires DHCP leases periodically
flood - a flood ping test; use with care
tcp_echo - data forwarding program for performance test
nc_test_master - network characterization master
nc_test_slave - network characterization slave
server_test - a very simple server example
tftp_client_test - performs a tftp get and put from/to "server"
tftp_server_test - runs a tftp server for a short while
set_mac_address - set MAC address(es) of interfaces in NVRAM
bridge - contributed network bridge code
nc6_test_master - IPv4/IPv6 network characterization master
nc6_test_slave - IPv4/IPv6 network characterization slave
ga_server_test - a very simple IPv4/IPv6 server example

Standalone Tests

socket_test - trivial test of socket creation API
mbuf_test - trivial test of mbuf allocation API
Chapter 40. Tests and Demonstrations

These two do not communicate over the net; they just perform simple API tests then exit.

\textit{ftp\_test} - simple FTP test, connects to "server"

This test initializes the interface(s) then connects to the FTP server on the “server” machine for each active interface in turn, confirms that the connection was successful, disconnects and exits. This tests interworking with the server.

\textit{ping\_test} - pings "server" and non-existent host to test timeout

This test initializes the interface(s) then pings the server machine in the standard way, then pings address “32 up” from the server in the expectation that there is no machine there. This confirms that the successful ping is not a false positive, and tests the receive timeout. If there is such a machine, of course the 2nd set of pings succeeds, confirming that we can talk to a machine not previously mentioned by configuration or by bootp. It then does the same thing on the other interface, eth1.

If IPv6 is enabled, the program will also ping to the address it last received a router advertisement from. Also a ping will be made to that address plus 32, in a similar way the the IPv4 case.

\textit{dhcp\_test} - ping test, but also manipulates DHCP leases

This test is very similar to the ping test, but in addition, provided the network package is not configured to do this automatically, it manually relinquishes and reclaims DHCP leases for all available interfaces. This tests the external API to DHCP. See section below describing this.

\textit{flood} - a flood ping test; use with care

This test performs pings on all interfaces as quickly as possible, and only prints status information periodically. Flood pinging is bad for network performance; so do not use this test on general purpose networks unless protected by a switch.

Performance Test

\textit{tcp\_echo} - data forwarding program for performance test

\textit{tcp\_echo} is one part of the standard performance test we use. The other parts are host programs \textit{tcp\_source} and \textit{tcp\_sink}. To make these (under your \textit{HOST} system) cd to the tests source directory in the eCos repository and type “make -f make.host” - this should build \textit{tcp\_source} and \textit{tcp\_sink}.

The host program “\textit{tcp\_source}” sends data to the target. On the target, “\textit{tcp\_echo}” sends it onwards to “\textit{tcp\_sink}” running on your host. So the target must receive and send on all the data that \textit{tcp\_source} sends it; the time taken for this is measured and the data rate is calculated.

To invoke the test, first start \textit{tcp\_echo} on the target board and wait for it to become quiescent - it will report work to calibrate a CPU load which can be used to simulate real operating conditions for the stack.

Then on your host machine, in one terminal window, invoke \textit{tcp\_sink} giving it the IP address (or hostname) of one interface of the target board. For example “\textit{tcp\_sink 10.130.39.66}”. \textit{tcp\_echo} on the target will print something like “\textit{SINK connection from 10.130.39.13:1143}” when \textit{tcp\_sink} is correctly invoked.

Next, in another host terminal window, invoke \textit{tcp\_source}, giving it the IP address (or hostname) of an interface of the target board, and optionally a background load to apply to the target while the test runs. For example, “\textit{tcp\_source 194.130.39.66}” to run the test with no additional target CPU load, or “\textit{tcp\_source 194.130.39.66 85}” to load it up to 85% used. The target load must be a multiple of 5. \textit{tcp\_echo} on the target will print something like “\textit{SOURCE connection from 194.130.39.13:1144}” when \textit{tcp\_source} is correctly invoked.
Chapter 40. Tests and Demonstrations

You can connect tcp_sink to one target interface and tcp_source to another, or both to the same interface. Similarly, you can run tcp_sink and tcp_source on the same host machine or different ones. TCP/IP and ARP look after them finding one another, as intended.

nc_test_master - network characterization master
nc_test_slave - network characterization slave

These tests talk to each other to measure network performance. They can each run on either a test target or a host computer given some customization to your local environment. As provided, nc_test_slave must run on the test target, and nc_test_master must be run on a host computer, and be given the test target’s IP address or hostname.

The tests print network performance for various packet sizes over UDP and TCP, versus various additional CPU loads on the target.

The programs

nc6_test_slave
nc6_test_master

are additional forms which support both IPv4 and IPv6 addressing.

Interactive Tests

server_test - a very simple server example

This test simply awaits a connection on port 7734 and after accepting a connection, gets a packet (with a timeout of a few seconds) and prints it.

The connection is then closed. We then loop to await the next connection, and so on. To use it, telnet to the target on port 7734 then type something (quickly!)

% telnet 172.16.19.171 7734
Hello target board

and the test program will print something like:

connection from 172.16.19.13:3369
buf = "Hello target board"

ga_server_test - another very simple server example

This is a variation on the ga_server_test test with the difference being that it uses the getaddrinfo function to set up its addresses. On a system with IPv6 enabled, it will listen on port 7734 for a TCP connection via either IPv4 or IPv6.

tftp_client_test - performs a tftp get and put from/to “server”

This is only partially interactive. You need to set things up on the “server” in order for this to work, and you will need to look at the server afterwards to confirm that all was well.

For each interface in turn, this test attempts to read by tftp from the server, a file called tftp_get and prints the status and contents it read (if any). It then writes the same data to a file called tftp_put on the same server.

In order for this to succeed, both files must already exist. The TFTP protocol does not require that a WRQ request _create_ a file, just that it can write it. The TFTP server on Linux certainly will only allow writes to an existing file, given the appropriate permission. Thus, you need to have these files in place, with proper permission, before running the test.
Chapter 40. Tests and Demonstrations

The conventional place for the tftp server to operate in LINUX is /tftpboot/; you will likely need root privileges to create files there. The data contents of `tftp_get` can be anything you like, but anything very large will waste lots of time printing it on the test’s stdout, and anything above 32kB will cause a buffer overflow and unpredictable failure.

Creating an empty `tftp_put` file (e.g. by copying /dev/null to it) is neatest. So before the test you should have something like:

```
-rw-rw-rw- 1 root 1076 May 1 11:39 tftp_get
-rw-rw-rw- 1 root 0 May 1 15:52 tftp_put
```

note that both files have public permissions wide open. After running the test, `tftp_put` should be a copy of `tftp_get`.

```
-rw-rw-rw- 1 root 1076 May 1 11:39 tftp_get
-rw-rw-rw- 1 root 1076 May 1 15:52 tftp_put
```

If the configuration contains IPv6 support, the test program will also use IPv6. It will attempt to put/get the files listed above from the address it last received a router’s solicit from.

`tftp_server_test` - runs a tftp server for a short while

This test is truly interactive, in that you can use a standard tftp application to get and put files from the server, during the 5 minutes that it runs. The dummy filesystem which underlies the server initially contains one file, called “uu” which contains part of a familiar text and some padding. It also accommodates creation of 3 further files of up to 1Mb in size and names of up to 256 bytes. Exceeding these limits will cause a buffer overflow and unpredictable failure.

The dummy filesystem is an implementation of the generic API which allows a true filesystem to be attached to the tftp server in the network stack.

We have been testing the tftp server by running the test on the target board, then using two different host computers connecting to the different target interfaces, putting a file from each, getting the “uu” file, and getting the file from the other computer. This verifies that data is preserved during the transfer as well as interworking with standard tftp applications.

**Maintenance Tools**

`set_mac_address` - set MAC address(es) of interfaces in NVRAM

This program makes an example `ioctl()` call SIOCSIFHWADDR “Socket IO Set InterFace HardWare AD-Dress” to set the MAC address on targets where this is supported and enabled in the configuration. You must edit the source to choose a MAC address and further edit it to allow this very dangerous operation. Not all ethernet drivers support this operation, because most ethernet hardware does not support it — or it comes pre-set from the factory. *Do not use this program.*
Chapter 41. Support Features

TFTP

The TFTP client and server are described in tftp_support.h;

The TFTP client has a new and an older, deprecated, API. The new API works for both IPv4 and IPv6 where as the deprecated API is IPv4 only.

The new API is as follows:

```c
int tftp_client_get(const char * const filename,
    const char * const server,
    const int port,
    char * buf,
    int len,
    const int mode,
    int * const err);

int tftp_client_put(const char * const filename,
    const char * const server,
    const int port,
    const char * buf,
    int len,
    const int mode,
    int * const err);
```

Currently `server` can only be a numeric IPv4 or IPv6 address. The resolver is currently not used, but it is planned to add this feature (patches welcome). If `port` is zero the client connects to the default TFTP port on the server. Otherwise the specified port is used.

The deprecated API is:

```c
int tftp_get(const char * const filename,
    const struct sockaddr_in * const server,
    char * buf,
    int len,
    const int mode,
    int * const err);

int tftp_put(const char * const filename,
    const struct sockaddr_in * const server,
    const char * buffer,
    int len,
    const int mode,
    int * const err);
```

The `server` should contain the address of the server to contact. If the `sin_port` member of the structure is zero the default TFTP port is used. Otherwise the specified port is used.

Both API’s report errors in the same way. The functions return a value of -1 and `*err` will be set to one of the following values:
#define TFTP_ENOTFOUND 1 /* file not found */
#define TFTP_EACCESS 2 /* access violation */
#define TFTP_ENOSPACE 3 /* disk full or allocation exceeded */
#define TFTP_EBADOP 4 /* illegal TFTP operation */
#define TFTP_EBADID 5 /* unknown transfer ID */
#define TFTP_EEXISTS 6 /* file already exists */
#define TFTP_ENOUSER 7 /* no such user */
#define TFTP_TIMEOUT 8 /* operation timed out */
#define TFTP_NETERR 9 /* some sort of network error */
#define TFTP_INVALID 10 /* invalid parameter */
#define TFTP_PROTOCOL 11 /* protocol violation */
#define TFTP_TOOLARGE 12 /* file is larger than buffer */

If there are no errors the return value is the number of bytes transferred.

The server is more complex. It requires a filesystem implementation to be supplied by the user, and attached
to the tftp server by means of a vector of function pointers:

```c
struct tftpd_fileops {
    int (*open)(const char *, int);
    int (*close)(int);
    int (*write)(int, const void *, int);
    int (*read)(int, void *, int);
};
```

These functions have the obvious semantics. The structure describing the filesystem is an argument to the
tftpd_start:

```c
int tftp_start(int port,
                struct tftpd_fileops *ops);
```

The first argument is the port to use for the server. If this port number is zero, the default TFTP port number
will be used. The return value from tftpd_start is a handle which can be passed to tftpd_stop. This will
kill the tftpd thread. Note that this is not a clean shutdown. The thread will simply be killed. tftpd_stop will
attempt to close the sockets the thread was listening on and free some of its allocated memory. But if the thread
was actively transferring data at the time tftpd_stop is called, it is quite likely some memory and a socket
will be leaked. Use this function with caution (or implement a clean shutdown and please contribute the code
back :-).

There are two CDL configuration options that control how many servers on how many different ports tftp
can be started. CYGSEM_NET_TFTPD_MULTITHREADED, when enabled, allows multiple tftpd threads
to operate on the same port number. With only one thread, while the thread is active transferring data, new
requests for transfers will not be served until the active transfer is complete. When multiple threads are started
on the same port, multiple transfers can take place simultaneously, up to the number of threads started. How-
ever a semaphore is required to synchronise the threads. This semaphore is required per port. The CDL option
CYGNUM_NET_TFTPD_MULTITHREADED_PORTS controls how many different port numbers mul-
tithreaded servers can service.

If CYGSEM_NET_TFTPD_MULTITHREADED is not enabled, only one thread may be run per port number.
But this removes the need for a semaphore and so CYGNUM_NET_TFTPD_MULTITHREADED_PORTS is
not required and unlimited number of ports can be used.

It should be noted that the TFTPD does not perform any form of file locking. When multiple servers are active,
it is assumed the underlying filesystem will refuse to open the same file multiple times, operate correctly with
simultaneous read/writes to the same file, or if you are unlucky, corrupt itself beyond all repair.

When IPv6 is enabled the tftpd thread will listen for requests from both IPv4 and IPv6 addresses.
As discussed in the description of the tftp_server_test above, an example filesystem is provided in net/common/VERSION/src/tftp_dummy_file.c for use by the tftp server test. The dummy filesystem is not a supported part of the network stack, it exists purely for demonstration purposes.

**DHCP**

This API publishes a routine to maintain DHCP state, and a semaphore that is signalled when a lease requires attention: this is your clue to call the aforementioned routine.

The intent with this API is that a simple DHCP client thread, which maintains the state of the interfaces, can go as follows: (after init_all_network_interfaces() is called from elsewhere)

```c
while ( 1 ) {
    while ( 1 ) {
        cyg_semaphore_wait( &dhcp_needs_attention );
        if ( ! dhcp_bind() ) // a lease expired
            break; // If we need to re-bind
    }
    dhcp_halt(); // tear everything down
    init_all_network_interfaces(); // re-initialize
}
```

and if the application does not want to suffer the overhead of a separate thread and its stack for this, this functionality can be placed in the app’s server loop in an obvious fashion. That is the goal of breaking out these internal elements. For example, some server might be arranged to poll DHCP from time to time like this:

```c
while ( 1 ) {
    init_all_network_interfaces();
    open-my-listen-sockets();
    while ( 1 ) {
        serve-one-request();
        // sleeps if no connections, but not forever;
        // so this loop is polled a few times a minute...
        if ( cyg_semaphore_trywait( &dhcp_needs_attention ) ) {
            if ( ! dhcp_bind() ) {
                close-my-listen-sockets();
                dhcp_halt();
                break;
            }
        }
    }
}
```

If the configuration option CYGOPT_NET_DHCP_DHCP_THREAD is defined, then eCos provides a thread as described initially. Independent of this option, initialization of the interfaces still occurs in init_all_network_interfaces() and your startup code can call that. It will start the DHCP management thread if configured. If a lease fails to be renewed, the management thread will shut down all interfaces and attempt to initialize all the interfaces again from scratch. This may cause chaos in the app, which is why managing the DHCP state in an application aware thread is actually better, just far less convenient for testing.

If the configuration option CYGOPT_NET_DHCP_OPTION_HOST_NAME is defined, then the TAG_HOST_NAME DHCP option will be included in any DHCP lease requests. The text for the hostname is set by calling dhcp_set_hostname(). Any DHCP lease requests made prior to calling dhcp_set_hostname() will not include the TAG_HOST_NAME DHCP option. The configuration option CYGNUM_NET_DHCP_OPTION_HOST_NAME_LEN controls the maximum length allowed for the
hostname. This permits the hostname text to be determined at run-time. Setting the hostname to the empty string will have the effect of disabling the TAG_HOST_NAME DHCP option.

If the configuration option CYGOPT_NET_DHCP_OPTION_DHCP_CLIENTID_MAC is defined, then the TAG_DHCP_CLIENTID DHCP option will be included in any DHCP lease requests. The client ID used will be the current MAC address of the network interface.

The option CYGOPT_NET_DHCP_PARM_REQ_LIST_ADDITIONAL allows additional DHCP options to be added to the request sent to the DHCP server. This option should be set to a comma separated list of options.

The option CYGOPT_NET_DHCP_PARM_REQ_LIST_REPLACE is similar to CYGOPT_NET_DHCP_PARM_REQ_LIST_ADDITIONAL but in this case it completely replaces the default list of options with the configured set of comma separated options.
getdomainname

getdomainname, setdomainname - get/set YP domain name of current host

SYNOPSIS
#include <unistd.h>

int getdomainname(char *name, size_t namelen);

int setdomainname(const char *name, size_t namelen);

DESCRIPTION
The getdomainname() function returns the YP domain name for the current
processor, as previously set by setdomainname(). The parameter namelen
specifies the size of the name array. If insufficient space is provided,
the returned name is truncated. The returned name is always null termi-
nated.

setdomainname() sets the domain name of the host machine to be name,
which has length namelen. This call is restricted to the superuser and
is normally used only when the system is bootstrapped.

RETURN VALUES
If the call succeeds a value of 0 is returned. If the call fails, a
value of -1 is returned and an error code is placed in the global vari-
able errno.

ERRORS
The following errors may be returned by these calls:

[EFAULT] The name or namelen parameter gave an invalid address.

[EPERM] The caller tried to set the domain name and was not
the superuser.

SEE ALSO
domainname(1), gethostid(3), gethostname(3), sysctl(3), sysct1(8), yp(8)

BUGS
Domain names are limited to MAXHOSTNAMELEN (from <sys/param.h>) charac-
ters, currently 256. This includes the terminating NUL character.

If the buffer passed to getdomainname() is too small, other operating
systems may not guarantee termination with NUL.

HISTORY
The getdomainname function call appeared in SunOS 3.x.
gethostname

GETHOSTNAME(3) BSD Library Functions Manual GETHOSTNAME(3)

NAME
gethostname, sethostname - get/set name of current host

SYNOPSIS
#include <unistd.h>

int gethostname(char *name, size_t namelen);

int sethostname(const char *name, size_t namelen);

DESCRIPTION
The gethostname() function returns the standard host name for the current
processor, as previously set by sethostname(). The parameter namelen
specifies the size of the name array. If insufficient space is provided,
the returned name is truncated. The returned name is always null termi-
nated.

sethostname() sets the name of the host machine to be name, which has
length namelen. This call is restricted to the superuser and is normally
used only when the system is bootstrapped.

RETURN VALUES
If the call succeeds a value of 0 is returned. If the call fails, a
value of -1 is returned and an error code is placed in the global vari-
able errno.

ERRORS
The following errors may be returned by these calls:

EFAULT] The name or namelen parameter gave an invalid address.

EPERM] The caller tried to set the hostname and was not the
superuser.

SEE ALSO
hostname(1), getdomainname(3), gethostid(3), sysctl(3), sysctl(8), yp(8)

STANDARDS
The gethostname() function call conforms to X/Open Portability Guide
Issue 4.2 (“XPG4.2”).

HISTORY
The gethostname() function call appeared in 4.2BSD.

BUGS
Host names are limited to MAXHOSTNAMELEN (from <sys/param.h>) characters,
currently 256. This includes the terminating NUL character.

If the buffer passed to gethostname() is smaller than MAXHOSTNAMELEN,
other operating systems may not guarantee termination with NUL.

NAME
htonl, htons, ntohl, ntohs, htobe32, htobe16, betoh32, betoh16, htole32, htole16, letoh32, letoh16, swap32, swap16 - convert values between different byte orderings

SYNOPSIS

```
#include <sys/types.h>
#include <machine/endian.h>

u_int32_t htonl(u_int32_t host32);

u_int16_t htons(u_int16_t host16);

u_int32_t ntohl(u_int32_t net32);

u_int16_t ntohs(u_int16_t net16);

u_int32_t htobe32(u_int32_t host32);

u_int16_t htobe16(u_int16_t host16);

u_int32_t betoh32(u_int32_t big32);

u_int16_t betoh16(u_int16_t big16);

u_int32_t htole32(u_int32_t host32);

u_int16_t htole16(u_int16_t host16);

u_int32_t letoh32(u_int32_t little32);

u_int16_t letoh16(u_int16_t little16);

u_int32_t swap32(u_int32_t val32);

u_int16_t
```

347
swap16(u_int16_t vall6);

DESCRIPTION
These routines convert 16- and 32-bit quantities between different byte orderings. The "swap" functions reverse the byte ordering of the given quantity, the others converts either from/to the native byte order used by the host to/from either little- or big-endian (a.k.a network) order.

Apart from the swap functions, the names can be described by this form: {src-order}to{dst-order}{size}. Both {src-order} and {dst-order} can take the following forms:

- h Host order.
- n Network order (big-endian).
- be Big-endian (most significant byte first).
- le Little-endian (least significant byte first).

One of the specified orderings must be 'h'. {size} will take these forms:

- l Long (32-bit, used in conjunction with forms involving 'n').
- s Short (16-bit, used in conjunction with forms involving 'n').
- 16 16-bit.
- 32 32-bit.

The swap functions are of the form: swap{size}.

Names involving 'n' convert quantities between network byte order and host byte order. The last letter ('s' or 'l') is a mnemonic for the traditional names for such quantities, short and long, respectively. Today, the C concept of short and long integers need not coincide with this traditional misunderstanding. On machines which have a byte order which is the same as the network order, routines are defined as null macros.

The functions involving either "be", "le", or "swap" use the numbers 16 and 32 for specifying the bitwidth of the quantities they operate on. Currently all supported architectures are either big- or little-endian so either the "be" or "le" variants are implemented as null macros.

The routines mentioned above which have either {src-order} or {dst-order} set to 'n' are most often used in conjunction with Internet addresses and ports as returned by gethostbyname(3) and getservent(3).

SEE ALSO
gethostbyname(3), getservent(3)

HISTORY
The byteorder functions appeared in 4.2BSD.

BUGS
On the vax, alpha, i386, and so far mips, bytes are handled backwards from most everyone else in the world. This is not expected to be fixed in the near future.
NAME
ethernet_address, ether_aton, ether_ntoa, ether_addr, ether_ntohost, ether_hostton,
ether_line - get ethers entry

SYNOPSIS
#include <netinet/if_ether.h>

char *
ether_ntoa(struct ether_addr *e);

struct ether_addr *
ether_aton(char *s);

int
ether_ntohost(char *hostname, struct ether_addr *e);

int
ether_hostton(char *hostname, struct ether_addr *e);

int
ether_line(char *l, struct ether_addr *e, char *hostname);

DESCRIPTION
Ethernet addresses are represented by the following structure:

    struct ether_addr {
        u_int8_t ether_addr_octet[6];
    };

The ether_ntoa() function converts this structure into an ASCII string of the form "xx:xx:xx:xx:xx:xx", consisting of 6 hexadecimal numbers separated by colons. It returns a pointer to a static buffer that is reused for each call. The ether_aton() converts an ASCII string of the same form and to a structure containing the 6 octets of the address. It returns a pointer to a static structure that is reused for each call.

The ether_ntohost() and ether_hostton() functions interrogate the database mapping host names to Ethernet addresses, /etc/ethers. The ether_ntohost() function looks up the given Ethernet address and writes the associated host name into the character buffer passed. This buffer should be MAXHOSTNAMELEN characters in size. The ether_hostton() function looks up the given host name and writes the associated Ethernet address into the structure passed. Both functions return zero if they find the requested host name or address, and -1 if not.

Each call reads /etc/ethers from the beginning; if a '+' appears alone on a line in the file, then ether_hostton() will consult the ethers.byname YP map, and ether_ntohost() will consult the ethers.byaddr YP map.

The ether_line() function parses a line from the /etc/ethers file and fills in the passed struct ether_addr and character buffer with the Ethernet address and host name on the line. It returns zero if the line was successfully parsed and -1 if not. The character buffer should be MAXHOSTNAMELEN characters in size.

FILES
/etc/ethers
Chapter 42. TCP/IP Library Reference

SEE ALSO
ethers(5)

HISTORY
The ether_ntoa(), ether_aton(), ether_ntohost(), ether_hostton(), and ether_line() functions were adopted from SunOS and appeared in NetBSD 0.9b.

BUGS
The data space used by these functions is static; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it.

BSD December 16, 1993 BSD

getaddrinfo

GETADDRINFO(3) BSD Library Functions Manual GETADDRINFO(3)

NAME
getaddrinfo, freeaddrinfo, gai_strerror - nodename-to-address translation in protocol-independent manner

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

char *gai_strerror(int ecode);

DESCRIPTION
The getaddrinfo() function is defined for protocol-independent nodename-
to-address translation. It performs the functionality of
gethostbyname(3) and getservbyname(3), but in a more sophisticated man-
ner.

The addrinfo structure is defined as a result of including the <netdb.h>
header:

struct addrinfo {
    int ai_flags; /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST */
    int ai_family; /* PF_xxx */
    int ai_socktype; /* SOCK_xxx */
    int ai_protocol; /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    size_t ai_addrlen; /* length of ai_addr */
    char *ai_canonname; /* canonical name for nodename */
    struct sockaddr *ai_addr; /* binary address */
    struct addrinfo *ai_next; /* next structure in linked list */
};
The nodename and servname arguments are pointers to NUL-terminated strings or NULL. One or both of these two arguments must be a non-null pointer. In the normal client scenario, both the nodename and servname are specified. In the normal server scenario, only the servname is specified. A non-null nodename string can be either a node name or a numeric host address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). A non-null servname string can be either a service name or a decimal port number.

The caller can optionally pass an addrinfo structure, pointed to by the third argument, to provide hints concerning the type of socket that the caller supports. In this hints structure all members other than ai_flags, ai_family, ai_socktype, and ai_protocol must be zero or a null pointer. A value of PF_UNSPEC for ai_family means the caller will accept any protocol family. A value of 0 for ai_socktype means the caller will accept any socket type. A value of 0 for ai_protocol means the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, then the ai_socktype member of the hints structure should be set to SOCK_STREAM when getaddrinfo() is called. If the caller handles only IPv4 and not IPv6, then the ai_family member of the hints structure should be set to PF_INET when getaddrinfo() is called. If the third argument to getaddrinfo() is a null pointer, this is the same as if the caller had filled in an addrinfo structure initialized to zero with ai_family set to PF_UNSPEC.

Upon successful return a pointer to a linked list of one or more addrinfo structures is returned through the final argument. The caller can process each addrinfo structure in this list by following the ai_next pointer, until a null pointer is encountered. In each returned addrinfo structure the three members ai_family, ai_socktype, and ai_protocol are the corresponding arguments for a call to the socket() function. In each addrinfo structure the ai_addr member points to a filled-in socket address structure whose length is specified by the ai_addrlen member.

If the AI_PASSIVE bit is set in the ai_flags member of the hints structure, then the caller plans to use the returned socket address structure in a call to bind(). In this case, if the nodename argument is a null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address.

If the AI_PASSIVE bit is not set in the ai_flags member of the hints structure, then the returned socket address structure will be ready for a call to connect() (for a connection-oriented protocol) or either connect(), sendto(), or sendmsg() (for a connectionless protocol). In this case, if the nodename argument is a null pointer, then the IP address portion of the socket address structure will be set to the loopback address.

If the AI_CANONNAME bit is set in the ai_flags member of the hints structure, then upon successful return the ai_canonname member of the first addrinfo structure in the linked list will point to a NUL-terminated string containing the canonical name of the specified nodename.

If the AI_NUMERICHOST bit is set in the ai_flags member of the hints structure, then a non-null nodename string must be a numeric host address string. Otherwise an error of EAI_NONAME is returned. This flag prevents any type of name resolution service (e.g., the DNS) from being called.
Chapter 42. TCP/IP Library Reference

The arguments to getaddrinfo() must sufficiently be consistent and unambiguous. Here are pitfall cases you may encounter:

- getaddrinfo() will raise an error if members of the hints structure are not consistent. For example, for internet address families, getaddrinfo() will raise an error if you specify SOCK_STREAM to ai_socktype while you specify IPPROTO_UDP to ai_protocol.

- If you specify a servname which is defined only for certain ai_socktype, getaddrinfo() will raise an error because the arguments are not consistent. For example, getaddrinfo() will raise an error if you ask for “tftp” service on SOCK_STREAM.

- For internet address families, if you specify servname while you set ai_socktype to SOCK_RAW, getaddrinfo() will raise an error, because service names are not defined for the internet SOCK_RAW space.

- If you specify a numeric servname, while leaving ai_socktype and ai_protocol unspecified, getaddrinfo() will raise an error. This is because the numeric servname does not identify any socket type, and getaddrinfo() is not allowed to glob the argument in such case.

All of the information returned by getaddrinfo() is dynamically allocated: the addrinfo structures, the socket address structures, and canonical node name strings pointed to by the addrinfo structures. To return this information to the system the function freeaddrinfo() is called. The addrinfo structure pointed to by the ai argument is freed, along with any dynamic storage pointed to by the structure. This operation is repeated until a NULL ai_next pointer is encountered.

To aid applications in printing error messages based on the EAI_xxx codes returned by getaddrinfo(), gai_strerror() is defined. The argument is one of the EAI_xxx values defined earlier and the return value points to a string describing the error. If the argument is not one of the EAI_xxx values, the function still returns a pointer to a string whose contents indicate an unknown error.

Extension for scoped IPv6 address

The implementation allows experimental numeric IPv6 address notation with scope identifier. By appending the percent character and scope identifier to addresses, you can fill sin6_scope_id field for addresses. This would make management of scoped address easier, and allows cut-and-paste input of scoped address.

At this moment the code supports only link-local addresses with the format. Scope identifier is hardcoded to name of hardware interface associated with the link. (such as ne0). Example would be like “fe80::1%ne0”, which means “fe80::1 on the link associated with ne0 interface”.

The implementation is still very experimental and non-standard. The current implementation assumes one-by-one relationship between interface and link, which is not necessarily true from the specification.

EXAMPLES

The following code tries to connect to “www.kame.net” service “http”. via stream socket. It loops through all the addresses available, regardless from address family. If the destination resolves to IPv4 address, it will use AF_INET socket. Similarly, if it resolves to IPv6, AF_INET6 socket is used. Observe that there is no hardcoded reference to particu-
lar address family. The code works even if getaddrinfo returns addresses that are not IPv4/v6.

```c
struct addrinfo hints, *res, *res0;
int error;
int s;
const char *cause = NULL;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error) {
    errx(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype,
               res->ai_protocol);
    if (s < 0) {
        cause = "socket";
        continue;
    }
    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "connect";
        close(s);
        s = -1;
        continue;
    }
    break; /* okay we got one */
}
if (s < 0) {
    err(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```

The following example tries to open a wildcard listening socket onto service "http", for all the address families available.

```c
struct addrinfo hints, *res, *res0;
int error;
int s[MAXSOCK];
int nsock;
const char *cause = NULL;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res0);
if (error) {
    errx(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
nsock = 0;
for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
    s[nsock] = socket(res->ai_family, res->ai_socktype,
```
Chapter 42. TCP/IP Library Reference

```c
res->ai_protocol);
if (s[nsock] < 0) {
    cause = "socket";
    continue;
}

if (bind(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
    cause = "bind";
    close(s[nsock]);
    continue;
}
(void) listen(s[nsock], 5);

nsock++;
}
if (nsock == 0) {
    err(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```

DIAGNOSTICS
Error return status from getaddrinfo() is zero on success and non-zero on errors. Non-zero error codes are defined in `<netdb.h>', and as follows:

- **EAI_ADDRFAMILY** Address family for nodename not supported.
- **EAI_AGAIN** Temporary failure in name resolution.
- **EAI_BADFLAGS** Invalid value for ai_flags.
- **EAI_FAIL** Non-recoverable failure in name resolution.
- **EAI_FAMILY** ai_family not supported.
- **EAI_MEMORY** Memory allocation failure.
- **EAI_NODATA** No address associated with nodename.
- **EAI_NONAME** nodename nor servname provided, or not known.
- **EAI_SERVICE** servname not supported for ai_socktype.
- **EAI_SOCKTYPE** ai_socktype not supported.
- **EAI_SYSTEM** System error returned in errno.

If called with proper argument, gai_strerror() returns a pointer to a string describing the given error code. If the argument is not one of the EAI_xxx values, the function still returns a pointer to a string whose contents indicate an unknown error.

SEE ALSO
getnameinfo(3), gethostbyname(3), getservbyname(3), hosts(5), resolv.conf(5), services(5), hostname(7), named(8)


HISTORY
The implementation first appeared in WIDE Hydrangea IPv6 protocol stack kit.

STANDARDS
The `getaddrinfo()` function is defined in IEEE POSIX 1003.1g draft specification, and documented in "Basic Socket Interface Extensions for IPv6" (RFC2553).

**BUGS**

The current implementation is not thread-safe.

The text was shamelessly copied from RFC2553.

gethostbyname

```
GETHOSTBYNAME(3) BSD Library Functions Manual GETHOSTBYNAME(3)

NAME
  gethostbyname, gethostbyname2, gethostbyaddr, gethostent, sethostent,
  endhostent, hstrerror, herror - get network host entry

SYNOPSIS

#include <netdb.h>
extern int h_errno;

struct hostent *
gethostbyname(const char *name);

struct hostent *
gethostbyname2(const char *name, int af);

struct hostent *
gethostbyaddr(const char *addr, int len, int af);

struct hostent *
gethostent(void);

void
sethostent(int stayopen);

void
endhostent(void);

void
herror(const char *string);

const char *
hstrerror(int err);
```

**DESCRIPTION**

The `gethostbyname()` and `gethostbyaddr()` functions each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either information obtained from the name server (i.e., resolver(3) and named(8)), broken-out fields from a line in /etc/hosts, or database entries supplied by the yp(8) system. `resolv.conf(5)` describes how the particular database is chosen.

```c
struct hostent {
```
Chapter 42. TCP/IP Library Reference

```c
char *h_name; /* official name of host */
char **h_aliases; /* alias list */
int h_addrtype; /* host address type */
int h_length; /* length of address */
char **h_addr_list; /* list of addresses from name server */

#define h_addr h_addr_list[0] /* address, for backward compatibility */
```

The members of this structure are:

- **h_name**: Official name of the host.
- **h_aliases**: A zero-terminated array of alternate names for the host.
- **h_addrtype**: The type of address being returned.
- **h_length**: The length, in bytes, of the address.
- **h_addr_list**: A zero-terminated array of network addresses for the host. Host addresses are returned in network byte order.
- **h_addr**: The first address in h_addr_list; this is for backward compatibility.

The function `gethostbyname()` will search for the named host in the current domain and its parents using the search lookup semantics detailed in `resolv.conf(5)` and `hostname(7)`.

`gethostbyname2()` is an advanced form of `gethostbyname()` which allows lookups in address families other than `AF_INET`, for example `AF_INET6`.

The `gethostbyaddr()` function will search for the specified address of length len in the address family af. The only address family currently supported is `AF_INET`.

The `sethostent()` function may be used to request the use of a connected TCP socket for queries. If the stayopen flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to `gethostbyname()` or `gethostbyaddr()`. Otherwise, queries are performed using UDP datagrams.

The `endhostent()` function closes the TCP connection.

The `herror()` function prints an error message describing the failure. If its argument string is non-null, it is prepended to the message string and separated from it by a colon (`:`) and a space. The error message is printed with a trailing newline. The contents of the error message is the same as that returned by `hstrerror()` with argument `h_errno`.

**FILES**

`/etc/hosts`  
`/etc/resolv.conf`

**DIAGNOSTICS**

Error return status from `gethostbyname()`, `gethostbyname2()`, and `gethostbyaddr()` is indicated by return of a null pointer. The external integer `h_errno` may then be checked to see whether this is a temporary failure or an invalid or unknown host.

The variable `h_errno` can have the following values:
HOST_NOT_FOUND  No such host is known.

TRY_AGAIN  This is usually a temporary error and means that the
local server did not receive a response from an authenti-
tative server. A retry at some later time may succeed.

NO_RECOVERY  Some unexpected server failure was encountered. This is
a non-recoverable error.

NO_DATA  The requested name is valid but does not have an IP
address; this is not a temporary error. This means that
the name is known to the name server but there is no
address associated with this name. Another type of
request to the name server using this domain name will
result in an answer; for example, a mail-forwarder may be
registered for this domain.

SEE ALSO
resolver(3), getaddrinfo(3), getnameinfo(3), hosts(5), resolv.conf(5),
hostname(7), named(8)

CAVEAT
If the search routines in resolv.conf(5) decide to read the /etc/hosts
file, gethostent() and other functions will read the next line of the
file, re-opening the file if necessary.

The sethostent() function opens and/or rewinds the file /etc/hosts. If
the stayopen argument is non-zero, the file will not be closed after each
call to gethostbyname(), gethostbyname2(), or gethostbyaddr().

The endhostent() function closes the file.

HISTORY
The herror() function appeared in 4.3BSD. The endhostent(),
gethostbyaddr(), gethostbyname(), gethostent(), and sethostent() func-
tions appeared in 4.2BSD.

BUGS
These functions use static data storage; if the data is needed for future
use, it should be copied before any subsequent calls overwrite it. Only
the Internet address formats are currently understood.

YP does not support any address families other than AF_INET and uses the
traditional database format.

BSD  March 13, 1997  BSD
#include <ifaddrs.h>

int getifaddrs(struct ifaddrs **ifap);

void freeifaddrs(struct ifaddrs *ifap);

DESCRIPTION

The getifaddrs() function stores a reference to a linked list of the network interfaces on the local machine in the memory referenced by ifap. The list consists of ifaddrs structures, as defined in the include file <ifaddrs.h>. The ifaddrs structure contains at least the following entries:

- `struct ifaddrs *ifa_next;` /* Pointer to next struct */
- `char *ifa_name;` /* Interface name */
- `u_int ifa_flags;` /* Interface flags */
- `struct sockaddr *ifa_addr;` /* Interface address */
- `struct sockaddr *ifa_netmask;` /* Interface netmask */
- `struct sockaddr *ifa_broadaddr;` /* Interface broadcast address */
- `struct sockaddr *ifa_dstaddr;` /* P2P interface destination */
- `void *ifa_data;` /* Address specific data */

`ifa_next`
Contains a pointer to the next structure on the list. This field is set to NULL in last structure on the list.

`ifa_name`
Contains the interface name.

`ifa_flags`
Contains the interface flags, as set by ifconfig(8).

`ifa_addr`
References either the address of the interface or the link level address of the interface, if one exists, otherwise it is NULL. (The sa_family field of the ifa_addr field should be consulted to determine the format of the ifa_addr address.)

`ifa_netmask`
References the netmask associated with ifa_addr, if one is set, otherwise it is NULL.

`ifa_broadaddr`
This field, which should only be referenced for non-P2P interfaces, references the broadcast address associated with ifa_addr, if one exists, otherwise it is NULL.

`ifa_dstaddr`
References the destination address on a P2P interface, if one exists, otherwise it is NULL.

`ifa_data`
References address family specific data. For AF_LINK addresses it contains a pointer to the struct if_data (as defined in include file <net/if.h>) which contains various interface attributes and statistics. For all other address families, it contains a pointer to the struct ifa_data (as defined in include file <net/if.h>) which contains per-address interface statistics.
The data returned by getifaddrs() is dynamically allocated and should be freed using freeifaddrs() when no longer needed.

RETURN VALUES
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS
The getifaddrs() may fail and set errno for any of the errors specified for the library routines ioctl(2), socket(2), malloc(3), or sysctl(3).

BUGS
If both <net/if.h> and <ifaddrs.h> are being included, <net/if.h> must be included before <ifaddrs.h>.

SEE ALSO
ioctl(2), socket(2), sysctl(3), networking(4), ifconfig(8)

HISTORY
The getifaddrs() function first appeared in BSDI BSD/OS. The function is supplied on OpenBSD since OpenBSD 2.7.

BSD October 13, 2005 BSD

getnameinfo

GETNAMEINFO(3) BSD Library Functions Manual GETNAMEINFO(3)

NAME
getnameinfo - address-to-nodename translation in protocol-independent manner

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags);

DESCRIPTION
The getnameinfo() function is defined for protocol-independent address-to-nodename translation. Its functionality is a reverse conversion of getaddrinfo(3), and implements similar functionality with gethostbyaddr(3) and getservbyport(3) in more sophisticated manner.

This function looks up an IP address and port number provided by the caller in the DNS and system-specific database, and returns text strings for both in buffers provided by the caller. The function indicates successful completion by a zero return value; a non-zero return value indicates failure.

The first argument, sa, points to either a sockaddr_in structure (for IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address and port number. The salen argument gives the length of the sockaddr_in or sockaddr_in6 structure.
The function returns the nodename associated with the IP address in the buffer pointed to by the host argument. The caller provides the size of this buffer via the hostlen argument. The service name associated with the port number is returned in the buffer pointed to by serv, and theservlen argument gives the length of this buffer. The caller specifies not to return either string by providing a zero value for the hostlen orservlen arguments. Otherwise, the caller must provide buffers large enough to hold the nodename and the service name, including the terminating null characters.

Unfortunately most systems do not provide constants that specify the maximum size of either a fully-qualified domain name or a service name. Therefore to aid the application in allocating buffers for these two returned strings the following constants are defined in `<netdb.h>`:

```c
#define NI_MAXHOST MAXHOSTNAMELEN
#define NI_MAXSERV 32
```

The first value is actually defined as the constant MAXDNAME in recent versions of BIND’s `<arpa/nameser.h>` header (older versions of BIND define this constant to be 256) and the second is a guess based on the services listed in the current Assigned Numbers RFC.

The final argument is a flag that changes the default actions of this function. By default the fully-qualified domain name (FQDN) for the host is looked up in the DNS and returned. If the flag bit NI_NOFQDN is set, only the nodename portion of the FQDN is returned for local hosts.

If the flag bit NI_NUMERICHOST is set, or if the host’s name cannot be located in the DNS, the numeric form of the host’s address is returned instead of its name (e.g., by calling inet_ntop() instead of gethostbyaddr()). If the flag bit NI_NAMEREQD is set, an error is returned if the host’s name cannot be located in the DNS.

If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (e.g., its port number) instead of its name. The two NI_NUMERICxxxx flags are required to support the `-n` flag that many commands provide.

A fifth flag bit, NI_DGRAM, specifies that the service is a datagram service, and causes getservbyport() to be called with a second argument of "udp" instead of its default of "tcp". This is required for the few ports (512-514) that have different services for UDP and TCP.

These NI_xxx flags are defined in `<netdb.h>`.

**Extension for scoped IPv6 address**

The implementation allows experimental numeric IPv6 address notation with scope identifier. IPv6 link-local address will appear as string like "fe80::1%ne0", if NI_WITHSCOPEID bit is enabled in flags argument. Refer to `getaddrinfo(3)` for the notation.

**EXAMPLES**

The following code tries to get numeric hostname, and service name, for given socket address. Observe that there is no hardcoded reference to particular address family.

```c
struct sockaddr *sa; /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
```
if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
    errx(1, "could not get numeric hostname");
    /*NOTREACHED*/
}
printf("host=%s, serv=%s\n", hbuf, sbuf);

The following version checks if the socket address has reverse address mapping.

struct sockaddr *sa; /* input */
char hbuf[NI_MAXHOST];
if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0, NI_NAMEREQD)) {
    errx(1, "could not resolve hostname");
    /*NOTREACHED*/
}
printf("host=%s\n", hbuf);

DIAGNOSTICS

The function indicates successful completion by a zero return value; a non-zero return value indicates failure. Error codes are as below:

EAI_AGAIN The name could not be resolved at this time. Future attempts may succeed.
EAI_BADFLAGS The flags had an invalid value.
EAI_FAIL A non-recoverable error occurred.
EAI_FAMILY The address family was not recognized or the address length was invalid for the specified family.
EAI_MEMORY There was a memory allocation failure.
EAI_NONAME The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both nodename and servname were null.
EAI_SYSTEM A system error occurred. The error code can be found in errno.

SEE ALSO

getaddrinfo(3), gethostbyaddr(3), getservbyport(3), hosts(5), resolv.conf(5), services(5), hostname(7), named(8)


HISTORY

The implementation first appeared in WIDE Hydrangea IPv6 protocol stack kit.
The getaddrinfo() function is defined IEEE POSIX 1003.1g draft specification, and documented in "Basic Socket Interface Extensions for IPv6" (RFC2553).

The current implementation is not thread-safe.

The text was shamelessly copied from RFC2553.

OpenBSD intentionally uses different NI_MAXHOST value from what RFC2553 suggests, to avoid buffer length handling mistakes.

```
getnetent
```

**NAME**
getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent - get network entry

**SYNOPSIS**
```
#include <netdb.h>

struct netent *
getnetent(void);

struct netent *
getnetbyname(char *name);

struct netent *
getnetbyaddr(in_addr_t net, int type);

void
setnetent(int stayopen);

void
endnetent(void);
```

**DESCRIPTION**
The getnetent(), getnetbyname(), and getnetbyaddr() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network database, /etc/networks.

```
struct netent {
    char *n_name; /* official name of net */
    char **n_aliases; /* alias list */
    int n_addrtype; /* net number type */
    in_addr_t n_net; /* net number */
};
```

The members of this structure are:

- **n_name**: The official name of the network.
n_aliases A zero-terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only
AF_INET.

n_net The network number. Network numbers are returned in machine
byte order.

The getnetent() function reads the next line of the file, opening the
file if necessary.

The setnetent() function opens and rewinds the file. If the stayopen
flag is non-zero, the net database will not be closed after each call to
getnetbyname() or getnetbyaddr().

The endnetent() function closes the file.

The getnetbyname() and getnetbyaddr() functions search the domain name
server if the system is configured to use one. If the search fails, or
no name server is configured, they sequentially search from the beginning
of the file until a matching net name or net address and type is found,
or until EOF is encountered. Network numbers are supplied in host order.

FILES
/etc/networks

DIAGNOSTICS
Null pointer (0) returned on EOF or error.

SEE ALSO
resolver(3), networks(5)

HISTORY
The getnetent(), getnetbyaddr(), getnetbyname(), setnetent(), and
endnetent() functions appeared in 4.2BSD.

BUGS
The data space used by these functions is static; if future use requires
the data, it should be copied before any subsequent calls to these func-
tions overwrite it. Only Internet network numbers are currently under-
stood. Expecting network numbers to fit in no more than 32 bits is
naive.

BSD March 13, 1997 BSD

getprotoent

GETPROTOENT(3) BSD Library Functions Manual GETPROTOENT(3)

NAME
getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent -
get protocol entry

SYNOPSIS
#include <netdb.h>

struct protoent *
Chapter 42. TCP/IP Library Reference

getprotoent(void);

struct protoent *
getprotobyname(char *name);

struct protoent *
getprotobynumber(int proto);

void
setprotoent(int stayopen);

void
endprotoent(void);

DESCRIPTION

The getprotoent(), getprotobyname(), and getprotobynumber() functions
each return a pointer to an object with the following structure contain-
ing the broken-out fields of a line in the network protocol database,
/etc/protocols.

```
struct protoent {
    char *p_name; /* official name of protocol */
    char **p_aliases; /* alias list */
    int p_proto; /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.
p_aliases A zero-terminated list of alternate names for the protocol.
p_proto The protocol number.

The getprotoent() function reads the next line of the file, opening the
file if necessary.

The setprotoent() function opens and rewinds the file. If the stayopen
flag is non-zero, the net database will not be closed after each call to
getprotobyname() or getprotobynumber().

The endprotoent() function closes the file.

The getprotobyname() and getprotobynumber() functions sequentially search
from the beginning of the file until a matching protocol name or protocol
number is found, or until EOF is encountered.

RETURN VALUES

Null pointer (0) returned on EOF or error.

FILES

/etc/protocols

SEE ALSO

protocols(5)

HISTORY

The getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(),
and endprotoent() functions appeared in 4.2BSD.
BUGS
These functions use a static data space; if the data is needed for future
use, it should be copied before any subsequent calls overwrite it. Only
the Internet protocols are currently understood.

BSD June 4, 1993 BSD

getrrsetbyname

GETRRSETBYNAME(3) BSD Library Functions Manual GETRRSETBYNAME(3)

NAME
getrrsetbyname — retrieve DNS records

SYNOPSIS
#include <netdb.h>

int getrrsetbyname(const char *hostname, unsigned int rdclass,
                     unsigned int rdtype, unsigned int flags, struct rrsetinfo **res);

int freerrset(struct rrsetinfo **rrset);

DESCRIPTION
getrrsetbyname() gets a set of resource records associated with a
hostname, class and type. hostname is a pointer to null-terminated
string. The flags field is currently unused and must be zero.

After a successful call to getrrsetbyname(), *res is a pointer to an
rrsetinfo structure, containing a list of one or more rdatainfo struc-
tures containing resource records and potentially another list of
rdatainfo structures containing SIG resource records associated with
those records. The members rri_rdclass and rri_rdtype are copied from
the parameters. rri_ttl and rri_name are properties of the obtained
rrset. The resource records contained in rri_rdatas and rri_sigs are in
uncompressed DNS wire format. Properties of the rdataset are represented
in the rri_flags bitfield. If the RRSET_VALIDATED bit is set, the data
has been DNSSEC validated and the signatures verified.

The following structures are used:

struct rdatainfo {
    unsigned int rdi_length; /* length of data */
    unsigned char *rdi_data; /* record data */
};

struct rrsetinfo {
    unsigned int rri_flags; /* RRSET_VALIDATED ... */
    unsigned int rri_rdclass; /* class number */
    unsigned int rri_rdtype; /* RR type number */
    unsigned int rri_ttl; /* time to live */
    unsigned int rri_nrdatas; /* size of rdatas array */
    unsigned int rri_nsigs; /* size of sigs array */
    char *rri_name; /* canonical name */
    struct rdatainfo *rri_rdatas; /* individual records */
    struct rdatainfo *rri_sigs; /* individual signatures */
};
Chapter 42. TCP/IP Library Reference

All of the information returned by getrrsetbyname() is dynamically allocated: the rrsetinfo and rdatainfo structures, and the canonical host name strings pointed to by the rrsetinfo structure. Memory allocated for the dynamically allocated structures created by a successful call to getrrsetbyname() is released by freerrset(). rrset is a pointer to a struct rrset created by a call to getrrsetbyname().

If the EDNS0 option is activated in resolv.conf(3), getrrsetbyname() will request DNSSEC authentication using the EDNS0 DNSSEC OK (DO) bit.

RETURN VALUES
getrrsetbyname() returns zero on success, and one of the following error codes if an error occurred:

ERRSET_NONAME the name does not exist
ERRSET_NODATA the name exists, but does not have data of the desired type
ERRSET_NOMEMORY memory could not be allocated
ERRSET_INVAL a parameter is invalid
ERRSET_FAIL other failure

SEE ALSO
resolver(3), resolv.conf(5), named(8)

AUTHORS
Jakob Schlyter <jakob@openbsd.org>

HISTORY
getrrsetbyname() first appeared in OpenBSD 3.0. The API first appeared in ISC BIND version 9.

BUGS
The data in *rdi_data should be returned in uncompressed wire format. Currently, the data is in compressed format and the caller can’t uncompress since it doesn’t have the full message.

CAVEATS
The RRSET_VALIDATED flag in rri_flags is set if the AD (authenticated data) bit in the DNS answer is set. This flag should not be trusted unless the transport between the nameserver and the resolver is secure (e.g. IPsec, trusted network, loopback communication).

BSD Oct 18, 2000 BSD

getservent

GETSERVENT(3) BSD Library Functions Manual GETSERVENT(3)

NAME
getservent, getservbyport, getservbyname, setservent, endservent - get service entry

SYNOPSIS
#include <netdb.h>
struct servent *
getservent(void);

struct servent *
getservbyname(char *name, char *proto);

struct servent *
getservbyport(int port, char *proto);

void
setservent(int stayopen);

void
derendservent(void);

DESCRIPTION
The getservent(), getservbyname(), and getservbyport() functions each
return a pointer to an object with the following structure containing the
broken-out fields of a line in the network services database,
/etc/services.

struct servent {
    char   *s_name;      /* official name of service */
    char **s_aliases;   /* alias list */
    int    s_port;      /* port service resides at */
    char   *s_proto;    /* protocol to use */
};

The members of this structure are:

s_name     The official name of the service.

s_aliases  A zero-terminated list of alternate names for the service.

s_port     The port number at which the service resides. Port numbers
are returned in network byte order.

s_proto    The name of the protocol to use when contacting the service.

The getservent() function reads the next line of the file, opening the
file if necessary.

The setservent() function opens and rewinds the file. If the stayopen
flag is non-zero, the net database will not be closed after each call to
getservbyname() or getservbyport().

The endservent() function closes the file.

The getservbyname() and getservbyport() functions sequentially search
from the beginning of the file until a matching protocol name or port
number (specified in network byte order) is found, or until EOF is
encountered. If a protocol name is also supplied (non-null), searches
must also match the protocol.

FILES
/etc/services

DIAGNOSTICS
Null pointer (0) returned on EOF or error.

SEE ALSO
Chapter 42. TCP/IP Library Reference

getprotoent(3), services(5)

HISTORY
The getservent(), getservbyport(), getservbyname(), setservent(), and
endservent() functions appeared in 4.2BSD.

BUGS
These functions use static data storage; if the data is needed for future
use, it should be copied before any subsequent calls overwrite it.
Expecting port numbers to fit in a 32-bit quantity is probably naïve.

BSD January 12, 1994 BSD

if_nametoindex

IF_NAMEETOINDEX(3) BSD Library Functions Manual IF_NAMEETOINDEX(3)

NAME
if_nametoindex, if_indextoname, if_nameindex, if_freenamexindex - convert
interface index to name, and vice versa

SYNOPSIS
#include <net/if.h>

unsigned int
if_nametoindex(const char *ifname);

char *
if_indextoname(unsigned int ifindex, char *ifname);

struct if_nameindex *
if_nameindex(void);

void
if_freenamexindex(struct if_nameindex *ptr);

DESCRIPTION
These functions map interface indexes to interface names (such as
"lo0"), and vice versa.

The if_nametoindex() function converts an interface name specified by the
ifname argument to an interface index (positive integer value). If the
specified interface does not exist, 0 will be returned.

if_indextoname() converts an interface index specified by the ifindex
argument to an interface name. The ifname argument must point to a
buffer of at least IF_NAMESIZE bytes into which the interface name corre-
sponding to the specified index is returned. (IF_NAMESIZE is also
defined in <net/if.h> and its value includes a terminating null byte at
the end of the interface name.) This pointer is also the return value of
the function. If there is no interface corresponding to the specified
index, NULL is returned.

if_nameindex() returns an array of if_nameindex structures.
if_nametoindex is also defined in <net/if.h>, and is as follows:

struct if_nameindex {
Chapter 42. TCP/IP Library Reference

unsigned int if_index; /* 1, 2, ... */
char *if_name; /* null terminated name: "le0", ... */
};

The end of the array of structures is indicated by a structure with an if_index of 0 and an if_name of NULL. The function returns a null pointer on error. The memory used for this array of structures along with the interface names pointed to by the if_name members is obtained dynamically. This memory is freed by the if_freenamerindex() function.

if_freenamerindex() takes a pointer that was returned by if_nameindex() as argument (ptr), and it reclaims the region allocated.

**DIAGNOSTICS**

- if_nametoindex() returns 0 on error, positive integer on success.
- if_indextoname() and if_nameindex() return NULL on errors.

**SEE ALSO**


**STANDARDS**

These functions are defined in “Basic Socket Interface Extensions for IPv6” (RFC2533).

BSD May 21, 1998 BSD

---

**inet**

INET(3) BSD Library Functions Manual INET(3)

**NAME**

inet_addr, inet_aton, inet_lnaof, inet_makeaddr, inet_netof, inet_network, inet_ntoa, inet_ntop, inet_pton - Internet address manipulation routines

**SYNOPSIS**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t
inet_addr(const char *cp);

int
inet_aton(const char *cp, struct in_addr *addr);

in_addr_t
inet_lnaof(struct in_addr in);

struct in_addr
inet_makeaddr(unsigned long net, unsigned long lna);

in_addr_t
inet_netof(struct in_addr in);

in_addr_t
inet_ntop(369
```
inet_network(const char *cp);
char *
inet_ntoa(struct in_addr in);
const char *
inet_ntop(int af, const void *src, char *dst, size_t size);
int
inet_pton(int af, const char *src, void *dst);

DESCRIPTION
The routines inet_aton(), inet_addr() and inet_network() interpret character strings representing numbers expressed in the Internet standard '.' notation. The inet_pton() function converts a presentation format address (that is, printable form as held in a character string) to network format (usually a struct in_addr or some other internal binary representation, in network byte order). It returns 1 if the address was valid for the specified address family, or 0 if the address wasn’t parseable in the specified address family, or -1 if some system error occurred (in which case errno will have been set). This function is presently valid for AF_INET and AF_INET6. The inet_aton() routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string was invalid. The inet_addr() and inet_network() functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively.

The function inet_ntop() converts an address from network format (usually a struct in_addr or some other binary form, in network byte order) to presentation format (suitable for external display purposes). It returns NULL if a system error occurs (in which case, errno will have been set), or it returns a pointer to the destination string. The routine inet_ntoa() takes an Internet address and returns an ASCII string representing the address in '.' notation. The routine inet_makeaddr() takes an Internet network number and a local network address and constructs an Internet address from it. The routines inet_netof() and inet_lnaof() break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES (IP VERSION 4)
Values specified using the '.' notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (such as the Intel 386, 486 and Pentium processors) the bytes referred to above appear as "d.c.b.a". That is, little-endian bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a
16-bit quantity and placed in the rightmost two bytes of the network
address. This makes the three part address format convenient for speci-
fying Class B network addresses as “128.net.host”.

When a two part address is supplied, the last part is interpreted as a
24-bit quantity and placed in the rightmost three bytes of the network
address. This makes the two part address format convenient for specifying
Class A network addresses as “net.host”.

When only one part is given, the value is stored directly in the network
address without any byte rearrangement.

All numbers supplied as “parts” in a ‘.’ notation may be decimal,
octal, or hexadecimal, as specified in the C language (i.e., a leading 0x
or 0X implies hexadecimal; otherwise, a leading 0 implies octal; other-
wise, the number is interpreted as decimal).

INTERNET ADDRESSES (IP VERSION 6)

In order to support scoped IPv6 addresses, getaddrinfo(3) and
getnameinfo(3) are recommended rather than the functions presented here.

The presentation format of an IPv6 address is given in [RFC1884 2.2]:

There are three conventional forms for representing IPv6 addresses as
text strings:

1. The preferred form is x:x:x:x:x:x:x:x, where the ‘x’s are the hex-
adecimal values of the eight 16-bit pieces of the address. Exam-

2. Due to the method of allocating certain styles of IPv6 addresses, it
will be common for addresses to contain long strings of zero bits.
In order to make writing addresses

containing zero bits easier a special syntax is available to com-
press the zeros. The use of “::” indicates multiple groups of 16
bits of zeros. The “::” can only appear once in an address. The
“::” can also be used to compress the leading and/or trailing
zeros in an address.

For example the following addresses:

may be represented as:

1080:0:0:0:8:800:200C:417A a unicast address
FF01:0:0:0:0:0:0:43 a multicast address
0:0:0:0:0:0:0:1 the loopback address
0:0:0:0:0:0:0:0 the unspecified addresses

1080::8:800:200C:417A a unicast address
FF01::43 a multicast address
::1 the loopback address
:: the unspecified addresses
3. An alternative form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is $x:x:x:x:x:d.d.d$, where the ‘x’s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the ’d’s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation). Examples:

- $0:0:0:0:0:0:13.1.68.3$
- $0:0:0:0:0:FFFF:129.144.52.38$

or in compressed form:

- $::13.1.68.3$
- $::FFFF:129.144.52.38$

**DIAGNOSTICS**

The constant INADDR_NONE is returned by inet_addr() and inet_network() for malformed requests.

**SEE ALSO**

byteorder(3), gethostbyname(3), getnetent(3), inet_net(3), hosts(5), networks(5)

**STANDARDS**

The inet_ntop and inetpton functions conforms to the IETF IPv6 BSD API and address formatting specifications. Note that inetpton does not accept 1-, 2-, or 3-part dotted addresses; all four parts must be specified. This is a narrower input set than that accepted by inet_aton.

**HISTORY**

The inet_addr, inet_network, inet_makeaddr, inet_lnaof and inet_netof functions appeared in 4.2BSD. The inet_aton and inet_ntoa functions appeared in 4.3BSD. The inetpton and inet_ntop functions appeared in BIND 4.9.4.

**BUGS**

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but inet_addr() cannot return that value without indicating failure. Also, inet_addr() should have been designed to return a struct in_addr. The newer inet_aton() function does not share these problems, and almost all existing code should be modified to use inet_aton() instead.

The problem of host byte ordering versus network byte ordering is confusing.

The string returned by inet_ntoa() resides in a static memory area.
SYNOPSIS
#include <netinet/in.h>

int inet6_option_space(int nbytes);

int inet6_option_init(void *bp, struct cmsghdr **cmsgp, int type);

int inet6_option_append(struct cmsghdr *cmsg, const u_int8_t *typep,
                        int multx, int plusy);

u_int8_t *inet6_option_alloc(struct cmsghdr *cmsg, int datalen, int multx,
                              int plusy);

int inet6_option_next(const struct cmsghdr *cmsg, u_int8_t **tptrp);

int inet6_option_find(const struct cmsghdr *cmsg, u_int8_t **tptrp,
                       int type);

DESCRIPTION
Building and parsing the Hop-by-Hop and Destination options is com-
licated due to alignment constrains, padding and ancillary data manipula-
tion. RFC2292 defines a set of functions to help the application. The
function prototypes for these functions are all in the <netinet/in.h>
header.

inet6_option_space
inet6_option_space() returns the number of bytes required to hold an
option when it is stored as ancillary data, including the cmsghdr struc-
ture at the beginning, and any padding at the end (to make its size a
multiple of 8 bytes). The argument is the size of the structure defining
the option, which must include any pad bytes at the beginning (the value
y in the alignment term “xn + y”), the type byte, the length byte, and
the option data.

Note: If multiple options are stored in a single ancillary data object,
which is the recommended technique, this function overestimates the
amount of space required by the size of N-1 cmsghdr structures, where N
is the number of options to be stored in the object. This is of little
consequence, since it is assumed that most Hop-by-Hop option headers and
Destination option headers carry only one option (appendix B of
[RFC-2460]).

inet6_option_init
inet6_option_init() is called once per ancillary data object that will
contain either Hop-by-Hop or Destination options. It returns 0 on suc-
cess or -1 on an error.

bp is a pointer to previously allocated space that will contain the
ancillary data object. It must be large enough to contain all the indi-
vidual options to be added by later calls to inet6_option_append() and
inet6_option_alloc().

cmsgp is a pointer to a pointer to a cmsghdr structure. *cmsgp is ini-
tialized by this function to point to the cmsghdr structure constructed
by this function in the buffer pointed to by bp.

type is either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the
cmsg_type member of the cmsghdr structure pointed to by *cmsgp.

inet6_option_append
This function appends a Hop-by-Hop option or a Destination option into an
ancillary data object that has been initialized by inet6_option_init().
This function returns 0 if it succeeds or -1 on an error.

cmsg is a pointer to the cmsghdr structure that must have been initial-
ized by inet6_option_init().
typep is a pointer to the 8-bit option type. It is assumed that this
field is immediately followed by the 8-bit option data length field,
which is then followed immediately by the option data. The caller ini-
itializes these three fields (the type-length-value, or TLV) before call-
ing this function.

The option type must have a value from 2 to 255, inclusive. (0 and 1 are
reserved for the Pad1 and PadN options, respectively.)

The option data length must have a value between 0 and 255, inclusive,
and is the length of the option data that follows.

multx is the value x in the alignment term "xn + y". It must have a
value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a
value between 0 and 7, inclusive.

inet6_option_alloc
This function appends a Hop-by-Hop option or a Destination option into an
ancillary data object that has been initialized by inet6_option_init().
This function returns a pointer to the 8-bit option type field that
starts the option on success, or NULL on an error.

The difference between this function and inet6_option_append() is that
the latter copies the contents of a previously built option into the
ancillary data object while the current function returns a pointer to the
space in the data object where the option’s TLV must then be built by the
caller.

cmsg is a pointer to the cmsghdr structure that must have been initial-
ized by inet6_option_init().
datalen is the value of the option data length byte for this option.
This value is required as an argument to allow the function to determine
if padding must be appended at the end of the option. (The
inet6_option_append() function does not need a data length argument since
the option data length must already be stored by the caller.)

multx is the value x in the alignment term "xn + y". It must have a
value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y". It must have a
value between 0 and 7, inclusive.

inet6_option_next
This function processes the next Hop-by-Hop option or Destination option
in an ancillary data object. If another option remains to be processed,
the return value of the function is 0 and \*tptrp points to the 8-bit
option type field (which is followed by the 8-bit option data length,
followed by the option data). If no more options remain to be processed,
the return value is -1 and \*tptrp is NULL. If an error occurs, the
return value is -1 and \*tptrp is not NULL.

cmsg is a pointer to cmshdr structure of which cmsg_level equals
IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and \*tptrp is used by
the function to remember its place in the ancillary data object each time
the function is called. The first time this function is called for a
given ancillary data object, \*tptrp must be set to NULL.

Each time this function returns success, \*tptrp points to the 8-bit
option type field for the next option to be processed.

\textbf{inet6\_option\_find}

This function is similar to the previously described \textit{inet6\_option\_next()} function, except this function lets the caller specify the option type to
be searched for, instead of always returning the next option in the
ancillary data object. cmsg is a pointer to cmshdr structure of which
cmsg\_level equals IPPROTO_IPV6 and cmsg\_type equals either IPV6_HOPOPTS
or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and \*tptrp is used by
the function to remember its place in the ancillary data object each time
the function is called. The first time this function is called for a
given ancillary data object, \*tptrp must be set to NULL. This function
starts searching for an option of the specified type beginning after the
value of \*tptrp. If an option of the specified type is located, this
function returns 0 and \*tptrp points to the 8-bit option type field for
the option of the specified type. If an option of the specified type is
not located, the return value is -1 and \*tptrp is NULL. If an error
occurs, the return value is -1 and \*tptrp is not NULL.

\textbf{DIAGNOSTICS}

\textit{inet6\_option\_init()} and \textit{inet6\_option\_append()} return 0 on success or -1
on an error.

\textit{inet6\_option\_alloc()} returns NULL on an error.

On errors, \textit{inet6\_option\_next()} and \textit{inet6\_option\_find()} return -1 setting
\*tptrp to non NULL value.

\textbf{EXAMPLES}

RFC2292 gives comprehensive examples in chapter 6.

\textbf{SEE ALSO}

W. Stevens and M. Thomas, Advanced Sockets API for IPv6, RFC2292,
February 1998.

S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6)

\textbf{HISTORY}

The implementation first appeared in KAME advanced networking kit.

\textbf{STANDARDS}

The functions are documented in “Advanced Sockets API for IPv6”
(RFC2292).
Chapter 42. TCP/IP Library Reference

BUGS
The text was shamelessly copied from RFC2292.

BSD December 10, 1999 BSD

inet6_rthdr_space

INET6_RTHDR_SPACE(3) BSD Library Functions Manual INET6_RTHDR_SPACE(3)

NAME
inet6_rthdr_space, inet6_rthdr_init, inet6_rthdr_add,
inet6_rthdr_lasthop, inet6_rthdr_reverse, inet6_rthdr_segments,
inet6_rthdr_getaddr, inet6_rthdr_getflags - IPv6 Routing Header Options
manipulation

SYNOPSIS
#include <netinet/in.h>

size_t
inet6_rthdr_space(int type, int segments);

struct cmsghdr *
inet6_rthdr_init(void *bp, int type);

int
inet6_rthdr_add(struct cmsghdr *cmsg, const struct in6_addr *addr,
         unsigned int flags);

int
inet6_rthdr_lasthop(struct cmsghdr *cmsg, unsigned int flags);

int
inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);

int
inet6_rthdr_segments(const struct cmsghdr *cmsg);

struct in6_addr *
inet6_rthdr_getaddr(struct cmsghdr *cmsg, int index);

int
inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);

DESCRIPTION
RFC2292 IPv6 advanced API defines eight functions that the application
calls to build and examine a Routing header. Four functions build a
Routing header:
inet6_rthdr_space() return #bytes required for ancillary data
inet6_rthdr_init() initialize ancillary data for Routing header
inet6_rthdr_add() add IPv6 address & flags to Routing header
inet6_rthdr_lasthop() specify the flags for the final hop
Four functions deal with a returned Routing header:

inet6_rthdr_reverse() reverse a Routing header
inet6_rthdr_segments() return #segments in a Routing header
inet6_rthdr_getaddr() fetch one address from a Routing header
inet6_rthdr_getflags() fetch one flag from a Routing header

The function prototypes for these functions are all in the `<netinet/in.h>` header.

inet6_rthdr_space
This function returns the number of bytes required to hold a Routing header of the specified type containing the specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the cmsghdr structure that precedes the Routing header, and any required padding.

If the return value is 0, then either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.

Note: This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, since all the ancillary data objects must be specified to sendmsg(2) as a single msg_control buffer.

inet6_rthdr_init
This function initializes the buffer pointed to by bp to contain a cmsghdr structure followed by a Routing header of the specified type. The cmsg_len member of the cmsghdr structure is initialized to the size of the structure plus the amount of space required by the Routing header. The cmsg_level and cmsg_type members are also initialized as required.

The caller must allocate the buffer and its size can be determined by calling inet6_rthdr_space().

Upon success the return value is the pointer to the cmsghdr structure, and this is then used as the first argument to the next two functions. Upon an error the return value is NULL.

inet6_rthdr_add
This function adds the address pointed to by addr to the end of the Routing header being constructed and sets the type of this hop to the value of flags. For an IPv6 Type 0 Routing header, flags must be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

If successful, the cmsg_len member of the cmsghdr structure is updated to account for the new address in the Routing header and the return value of the function is 0. Upon an error the return value of the function is -1.

inet6_rthdr_lasthop
This function specifies the Strict/Loose flag for the final hop of a Routing header. For an IPv6 Type 0 Routing header, flags must be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

The return value of the function is 0 upon success, or -1 upon an error.
Notice that a Routing header specifying N intermediate nodes requires N+1 Strict/Loose flags. This requires N calls to inet6_rthdr_add() followed by one call to inet6_rthdr_lasthop().

**inet6_rthdr_reverse**

This function takes a Routing header that was received as ancillary data (pointed to by the first argument, in) and writes a new Routing header that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

The return value of the function is 0 on success, or -1 upon an error.

**inet6_rthdr_segments**

This function returns the number of segments (addresses) contained in the Routing header described by cmsg. On success the return value is between 1 and 23, inclusive. The return value of the function is -1 upon an error.

**inet6_rthdr_getaddr**

This function returns a pointer to the IPv6 address specified by index (which must have a value between 1 and the value returned by inet6_rthdr_segments()) in the Routing header described by cmsg. An application should first call inet6_rthdr_segments() to obtain the number of segments in the Routing header.

Upon an error the return value of the function is NULL.

**inet6_rthdr_getflags**

This function returns the flags value specified by index (which must have a value between 0 and the value returned by inet6_rthdr_segments()) in the Routing header described by cmsg. For an IPv6 Type 0 Routing header the return value will be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

Upon an error the return value of the function is -1.

Note: Addresses are indexed starting at 1, and flags starting at 0, to maintain consistency with the terminology and figures in RFC2460.

**DIAGNOSTICS**

inet6_rthdr_space() returns 0 on errors.

inet6_rthdr_add(), inet6_rthdr_lasthop() and inet6_rthdr_reverse() return 0 on success, and returns -1 on error.

inet6_rthdr_init() and inet6_rthdr_getaddr() return NULL on error.

inet6_rthdr_segments() and inet6_rthdr_getflags() return -1 on error.

**EXAMPLES**

RFC2292 gives comprehensive examples in chapter 8.

**SEE ALSO**


**HISTORY**
Chapter 42. TCP/IP Library Reference

The implementation first appeared in KAME advanced networking kit.

STANDARDS
The functions are documented in “Advanced Sockets API for IPv6” (RFC2292).

BUGS
The text was shamelessly copied from RFC2292.

inet6_rthdr_reverse() is not implemented yet.

BSD December 10, 1999 BSD

inet_net

INET_NET(3) BSD Library Functions Manual INET_NET(3)

NAME
inet_net_ntop, inet_net_pton — Internet network number manipulation routines

SYNOPSIS
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *
inet_net_ntop(int af, const void *src, int bits, char *dst, size_t size);

int
inet_net_pton(int af, const char *src, void *dst, size_t size);

DESCRIPTION
The inet_net_ntop() function converts an Internet network number from network format (usually a struct in_addr or some other binary form, in network byte order) to CIDR presentation format (suitable for external display purposes). bits is the number of bits in src that are the network number. It returns NULL if a system error occurs (in which case, errno will have been set), or it returns a pointer to the destination string.

The inet_net_pton() function converts a presentation format Internet network number (that is, printable form as held in a character string) to network format (usually a struct in_addr or some other internal binary representation, in network byte order). It returns the number of bits (either computed based on the class, or specified with /CIDR), or -1 if a failure occurred (in which case errno will have been set. It will be set to ENOENT if the Internet network number was not valid).

The only value for af currently supported is AF_INET. size is the size of the result buffer dst.

NETWORK NUMBERS (IP VERSION 4)
Internet network numbers may be specified in one of the following forms:

a.b.c.d/bits
a.b.c.d
When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet network number. Note that when an Internet network number is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (such as the Intel 386, 486, and Pentium processors) the bytes referred to above appear as “d.c.b.a”. That is, little-endian bytes are ordered from right to left.

When a three part number is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the Internet network number. This makes the three part number format convenient for specifying Class B network numbers as “128.net.host”.

When a two part number is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the Internet network number. This makes the two part number format convenient for specifying Class A network numbers as “net.host”.

When only one part is given, the value is stored directly in the Internet network number without any byte rearrangement.

All numbers supplied as “parts” in a ‘.’ notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO
byteorder(3), inet(3), networks(5)

HISTORY
The inet_net_ntop and inet_net_pton functions first appeared in BIND 4.9.4.

BSD June 18, 1997 BSD
The routine `ipx_addr()` interprets character strings representing IPX addresses, returning binary information suitable for use in system calls. The routine `ipx_ntoa()` takes IPX addresses and returns ASCII strings representing the address in a notation in common use:

\<network number>.\<host number>.\<port number>

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to `ipx_addr()`. Any fields lacking super-decimal digits will have a trailing 'H' appended.

An effort has been made to ensure that `ipx_addr()` be compatible with most formats in common use. It will first separate an address into 1 to 3 fields using a single delimiter chosen from period ('.'), colon (':'), or pound-sign ('#'). Each field is then examined for byte separators (colon or period). If there are byte separators, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-network-order bytes. Next, the field is inspected for hyphens, in which case the field is assumed to be a number in decimal notation with hyphens separating the millennia. Next, the field is assumed to be a number: It is interpreted as hexadecimal if there is a leading '0x' (as in C), a trailing 'H' (as in Mesa), or there are any super-decimal digits present. It is interpreted as octal if there is a leading '0' and there are no super-octal digits. Otherwise, it is converted as a decimal number.

RETURN VALUES
None. (See BUGS.)

SEE ALSO
ns(4), hosts(5), networks(5)

HISTORY
The precursor `ns_addr()` and `ns_ntoa()` functions appeared in 4.3BSD.

BUGS
The string returned by `ipx_ntoa()` resides in a static memory area. The function `ipx_addr()` should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

ISO_ADDR

ISO_ADDR(3) BSD Library Functions Manual ISO_ADDR(3)

NAME
iso_addr, iso_ntoa - network address conversion routines for Open System Interconnection

SYNOPSIS

```
#include <sys/types.h>
#include <netiso/iso.h>

struct iso_addr *
iso_addr(char *cp);
```
char *
iso_ntoa(struct iso_addr *isoa);

DESCRIPTION
The routine iso_addr() interprets character strings representing OSI addresses, returning binary information suitable for use in system calls. The routine iso_ntoa() takes OSI addresses and returns ASCII strings representing NSAPs (network service access points) in a notation inverse to that accepted by iso_addr().

Unfortunately, no universal standard exists for representing OSI network addresses.

The format employed by iso_addr() is a sequence of hexadecimal “digits” (optionally separated by periods), of the form:

<hex digits>.<hex digits>.<hex digits>

Each pair of hexadecimal digits represents a byte with the leading digit indicating the higher-ordered bits. A period following an even number of bytes has no effect (but may be used to increase legibility). A period following an odd number of bytes has the effect of causing the byte of address being translated to have its higher order bits filled with zeros.

RETURN VALUES
iso_ntoa() always returns a null terminated string. iso_addr() always returns a pointer to a struct iso_addr. (See BUGS.)

SEE ALSO
iso(4)

HISTORY
The iso_addr() and iso_ntoa() functions appeared in 4.3BSD-Reno.

BUGS
The returned values reside in a static memory area.

The function iso_addr() should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

BSD June 4, 1993 BSD

link_addr

LINK_ADDR(3) BSD Library Functions Manual LINK_ADDR(3)

NAME
link_addr, link_ntoa - elementary address specification routines for link level access

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if_dl.h>

void
link_addr(const char *addr, struct sockaddr_dl *sdl);
The link_addr() function interprets character strings representing link-level addresses, returning binary information suitable for use in system calls. link_ntoa() takes a link-level address and returns an ASCII string representing some of the information present, including the link level address itself, and the interface name or number, if present. This facility is experimental and is still subject to change.

For link_addr(), the string addr may contain an optional network interface identifier of the form "name unit-number", suitable for the first argument to ifconfig(8), followed in all cases by a colon and an interface address in the form of groups of hexadecimal digits separated by periods. Each group represents a byte of address; address bytes are filled left to right from low order bytes through high order bytes.

Thus le0:8.0.9.13.d.30 represents an Ethernet address to be transmitted on the first Lance Ethernet interface.

net_addrcmp

net_addrcmp - compare socket address structures

NAME

net_addrcmp - compare socket address structures

SYNOPSIS

#include <netdb.h>

int

net_addrcmp(struct sockaddr *sa1, struct sockaddr *sa2);
Chapter 42. TCP/IP Library Reference

DESCRIPTION

The net_addrcmp() function compares two socket address structures, sa1 and sa2.

RETURN VALUES

If sa1 and sa2 are for the same address, net_addrcmp() returns 0.

The sa_len fields are compared first. If they do not match, net_addrcmp() returns -1 or 1 if sa1->sa_len is less than or greater than sa2->sa_len, respectively.

Next, the sa_family members are compared. If they do not match, net_addrcmp() returns -1 or 1 if sa1->sa_family is less than or greater than sa2->sa_family, respectively.

Lastly, if each socket address structure’s sa_len and sa_family fields match, the protocol-specific data (the sa_data field) is compared. If there’s a match, both sa1 and sa2 must refer to the same address, and 0 is returned; otherwise, a value >0 or <0 is returned.

HISTORY

A net_addrcmp() function was added in OpenBSD 2.5.

BSD July 3, 1999 BSD

NS(3)
BSD Library Functions Manual

NAME

ns_addr, ns_ntoa - Xerox NS(tm) address conversion routines

SYNOPSIS

#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr
ns_addr(char *cp);

char *
ns_ntoa(struct ns_addr ns);

DESCRIPTION

The routine ns_addr() interprets character strings representing XNS addresses, returning binary information suitable for use in system calls. The routine ns_ntoa() takes XNS addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

<network number>.<host number>.<port number>

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to ns_addr(). Any fields lacking super-decimal digits will have a trailing ‘H’ appended.

Unfortunately, no universal standard exists for representing XNS addresses.
addresses. An effort has been made to ensure that ns_addr() be compatible with most formats in common use. It will first separate an address into 1 to 3 fields using a single delimiter chosen from period ('.'), colon (':'), or pound-sign ('#'). Each field is then examined for byte separators (colon or period). If there are byte separators, each sub-field separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-network-order bytes. Next, the field is inspected for hyphens, in which case the field is assumed to be a number in decimal notation with hyphens separating the millennia. Next, the field is assumed to be a number: It is interpreted as hexadecimal if there is a leading '0x' (as in C), a trailing 'H' (as in Mesa), or there are any super-decimal digits present. It is interpreted as octal is there is a leading '0' and there are no super-octal digits. Otherwise, it is converted as a decimal number.

RETURN VALUES
None. (See BUGS.)

SEE ALSO
hosts(5), networks(5)

HISTORY
The ns_addr() and ns_toa() functions appeared in 4.3BSD.

BUGS
The string returned by ns_ntoa() resides in a static memory area. The function ns_addr() should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

BSD June 4, 1993 BSD
res_send(char *msg, int msglen, char *answer, int anslen);
int
res_init(void);
int
dn_comp(char *exp_dn, char *comp_dn, int length, char **dnptrs,
char **lastdnptr);
int
dn_expand(u_char *msg, u_char *eomorig, u_char *comp_dn, u_char *exp_dn,
int length);

DESCRIPTION
These routines are used for making, sending, and interpreting query and
reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver
routines is kept in the structure _res. Most of the values have reason-
able defaults and can be ignored. Options stored in _res.options are
defined in <resolv.h> and are as follows. Options are stored as a simple
bit mask containing the bitwise OR of the options enabled.

RES_INIT      True if the initial name server address and default domain
name are initialized (i.e., res_init() has been called).
RES_DEBUG     Print debugging messages.
RES_AAONLY    Accept authoritative answers only. With this option,
res_send() should continue until it finds an authoritative
answer or finds an error. Currently this is not imple-
mented.
RES_USEVC     Use TCP connections for queries instead of UDP datagrams.
RES_STAYOPEN  Used with RES_USEVC to keep the TCP connection open
between queries. This is useful only in programs that
regularly do many queries. UDP should be the normal mode
used.
RES_IGNTC     Unused currently (ignore truncation errors, i.e., don’t
retry with TCP).
RES_RECURSE   Set the recursion-desired bit in queries. This is the
default. (res_send() does not do iterative queries and
expects the name server to handle recursion.)
RES_DEFNAMES  If set, res_search() will append the default domain name
to single-component names (those that do not contain a
dot). This option is enabled by default.
RES_DNSRCH    If this option is set, res_search() will search for host
names in the current domain and in parent domains; see
hostname(7). This is used by the standard host lookup
routine gethostbyname(3). This option is enabled by
default.
RES_USE_INET6 Enables support for IPv6-only applications. This causes
IPv4 addresses to be returned as an IPv4 mapped address.
For example, 10.1.1.1 will be returned as ::ffff:10.1.1.1.
The option is not meaningful on OpenBSD.
The `res_init()` routine reads the configuration file (if any; see `resolv.conf(5)`) to get the default domain name, search list, and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable `LOCALDOMAIN`. This environment variable may contain several blank-separated tokens if you wish to override the search list on a per-process basis. This is similar to the search command in the configuration file. Another environment variable `RES_OPTIONS` can be set to override certain internal resolver options which are otherwise set by changing fields in the `_res` structure or are inherited from the configuration file’s options command. The syntax of the `RES_OPTIONS` environment variable is explained in `resolv.conf(5)`.

Initialization normally occurs on the first call to one of the following routines.

The `res_query()` function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the specified fully qualified domain name `dname`. The reply message is left in the answer buffer with length `anslen` supplied by the caller.

The `res_search()` routine makes a query and awaits a response like `res_query()`, but in addition, it implements the default and search rules controlled by the `RES_DEFNAMES` and `RES_DNSRCH` options. It returns the first successful reply.

The remaining routines are lower-level routines used by `res_query()`. The `res_mkquery()` function constructs a standard query message and places it in `buf`. It returns the size of the query, or -1 if the query is larger than `buflen`. The query type `op` is usually QUERY, but can be any of the query types defined in `<arpa/nameser.h>`. The domain name for the query is given by `dname`. `newrr` is currently unused but is intended for making update messages.

The `res_send()` routine sends a pre-formatted query and returns an answer. It will call `res_init()` if `RES_INIT` is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or -1 if there were errors.

The `dn_comp()` function compresses the domain name `exp_dn` and stores it in `comp_dn`. The size of the compressed name is returned or -1 if there were errors. The size of the array pointed to by `comp_dn` is given by length. The compression uses an array of pointers `dnptrs` to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit to the array is specified by `lastdnptr`. A side effect of `dn_comp()` is to update the list of pointers for labels inserted into the message as the name is compressed. If `dnptr` is NULL, names are not compressed. If `lastdnptr` is NULL, the list of labels is not updated.

The `dn_expand()` entry expands the compressed domain name `comp_dn` to a full domain name. The compressed name is contained in a query or reply message; `msg` is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by `exp_dn` which is of size `length`. The size of compressed name is returned or -1 if there was an error.

FILES
Chapter 42. TCP/IP Library Reference

/etc/resolv.conf configuration file see resolv.conf(5).

SEE ALSO
gethostbyname(3), resolv.conf(5), hostname(7), named(8)
RFC1032, RFC1033, RFC1034, RFC1035, RFC1535, RFC974
Name Server Operations Guide for BIND.

HISTORY
The res_query function appeared in 4.3BSD.

BSD June 4, 1993 BSD

accept

ACCEPT(2) BSD System Calls Manual ACCEPT(2)

NAME
accept - accept a connection on a socket

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

DESCRIPTION
The argument s is a socket that has been created with socket(2), bound to
an address with bind(2), and is listening for connections after a
listen(2). The accept() argument extracts the first connection request
on the queue of pending connections, creates a new socket with the same
properties of s, and allocates a new file descriptor for the socket. If
no pending connections are present on the queue, and the socket is not
marked as non-blocking, accept() blocks the caller until a connection is
present. If the socket is marked non-blocking and no pending connections
are present on the queue, accept() returns an error as described below.
The accepted socket may not be used to accept more connections. The
original socket s remains open.

The argument addr is a result parameter that is filled in with the
address of the connecting entity as known to the communications layer.
The exact format of the addr parameter is determined by the domain in
which the communication is occurring. The addrlen is a value-result
parameter; it should initially contain the amount of space pointed to by
addr; on return it will contain the actual length (in bytes) of the
address returned. This call is used with connection-based socket types,
currently with SOCK_STREAM.

It is possible to select(2) or poll(2) a socket for the purposes of doing
an accept() by selecting it for read.

For certain protocols which require an explicit confirmation, such as ISO
or DATAKIT, accept() can be thought of as merely dequeuing the next con-
nexion request and not implying confirmation. Confirmation can be
implied by a normal read or write on the new file descriptor, and rejec-
tion can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an msg_iovlen of 0 and a non-zero msg_controllen, or by issuing a getsockopt(2) request. Similarly, one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).

RETURN VALUES
The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS
The accept() will fail if:

- **[EBADF]** The descriptor is invalid.
- **[ENOTSOCK]** The descriptor references a file, not a socket.
- **[EOPNOTSUPP]** The referenced socket is not of type SOCK_STREAM.
- **[EINVAL]** The referenced socket is not listening for connections (that is, listen(2) has not yet been called).
- **[EFAULT]** The addr parameter is not in a writable part of the user address space.
- **[EWOULDBLOCK]** The socket is marked non-blocking and no connections are present to be accepted.
- **[EMFILE]** The per-process descriptor table is full.
- **[ENFILE]** The system file table is full.
- **[ECONNABORTED]** A connection has been aborted.

SEE ALSO
bind(2), connect(2), listen(2), poll(2), select(2), poll(2), socket(2)

HISTORY
The accept() function appeared in 4.2BSD.

BSD February 15, 1999 BSD

bind

### NAME
bind - bind a name to a socket

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/socket.h>

int bind(int, struct sockaddr *, int)
```
bind(int s, const struct sockaddr *name, socklen_t namelen);

DESCRIPTION
bind() assigns a name to an unnamed socket. When a socket is created
with socket(2) it exists in a name space (address family) but has no name
assigned. bind() requests that name be assigned to the socket.

NOTES
Binding a name in the UNIX domain creates a socket in the file system
that must be deleted by the caller when it is no longer needed (using
unlink(2)).

The rules used in name binding vary between communication domains. Con-
sult the manual entries in section 4 for detailed information.

RETURN VALUES
If the bind is successful, a 0 value is returned. A return value of -1
indicates an error, which is further specified in the global errno.

ERRORS
The bind() call will fail if:

- [EBADF] S is not a valid descriptor.
- [ENOTSOCK] S is not a socket.
- [EADDRNOTAVAIL] The specified address is not available from the local
  machine.
- [EADDRINUSE] The specified address is already in use.
- [EINVAL] The socket is already bound to an address.
- [EINVAL] The family of the socket and that requested in
  name->sa_family are not equivalent.
- [EACCES] The requested address is protected, and the current
  user has inadequate permission to access it.
- [EFAULT] The name parameter is not in a valid part of the user
  address space.

The following errors are specific to binding names in the UNIX domain.

- [ENOTDIR] A component of the path prefix is not a directory.
- [ENAMETOOLONG] A component of a pathname exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.
- [ENOENT] A prefix component of the path name does not exist.
- [ELOOP] Too many symbolic links were encountered in translat-
  ing the pathname.
- [EIO] An I/O error occurred while making the directory entry
  or allocating the inode.
- [EROFS] The name would reside on a read-only file system.
- [EISDIR] An empty pathname was specified.
connect

connect(2) BSD System Calls Manual connect(2)

NAME
connect - initiate a connection on a socket

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int
connect(int s, const struct sockaddr *name, socklen_t namelen);

DESCRIPTION
The parameter s is a socket. If it is of type SOCK_DGRAM, this call
specifies the peer with which the socket is to be associated; this
address is that to which datagrams are to be sent, and the only address
from which datagrams are to be received. If the socket is of type
SOCK_STREAM, this call attempts to make a connection to another socket.
The other socket is specified by name, which is an address in the commu-
ications space of the socket. Each communications space interprets the
name parameter in its own way. Generally, stream sockets may success-
fully connect() only once; datagram sockets may use connect() multiple
times to change their association. Datagram sockets may dissolve the
association by connecting to an invalid address, such as a null address.

RETURN VALUES
If the connection or binding succeeds, 0 is returned. Otherwise a -1 is
returned, and a more specific error code is stored in errno.

ERRORS
The connect() call fails if:

[EBADF] S is not a valid descriptor.
[ENOTSOCK] S is a descriptor for a file, not a socket.
[EADDRNOTAVAIL] The specified address is not available on this
machine.
[EAFNOSUPPORT] Addresses in the specified address family cannot be
used with this socket.
[EISCONN] The socket is already connected.
[ETIMEDOUT] Connection establishment timed out without establish-
ing a connection.
A TCP connection with a local broadcast, the all-ones or a multicast address as the peer was attempted.

The attempt to connect was forcefully rejected.

A connect was interrupted before it succeeded by the delivery of a signal.

The network isn’t reachable from this host.

The address is already in use.

The name parameter specifies an area outside the process address space.

The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) or poll(2) for completion by selecting the socket for writing, and also use getsockopt(2) with SO_ERROR to check for error conditions.

The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

A component of the path prefix is not a directory.

A component of a pathname exceeded (NAME_MAX) characters, or an entire path name exceeded (PATH_MAX) characters.

The named socket does not exist.

Search permission is denied for a component of the path prefix.

Write access to the named socket is denied.

Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), getsockname(2), getsockopt(2), poll(2), select(2), socket(2)

HISTORY

The connect() function call appeared in 4.2BSD.

BSD February 15, 1999 BSD
NAME
getpeername - get name of connected peer

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *name, socklen_t *namelen);

DESCRIPTION
getpeername() returns the address information of the peer connected to
socket s. One common use occurs when a process inherits an open socket,
such as TCP servers forked from inetd(8). In this scenario,
getpeername() is used to determine the connecting client’s IP address.

getpeername() takes three parameters:

s Contains the file descriptor of the socket whose peer should be looked
up.

name Points to a sockaddr structure that will hold the address informa-
tion for the connected peer. Normal use requires one to use a structure
specific to the protocol family in use, such as sockaddr_in (IPv4) or
sockaddr_in6 (IPv6), cast to a (struct sockaddr *).

For greater portability, especially with the newer protocol families, the
new struct sockaddr_storage should be used. sockaddr_storage is large
enough to hold any of the other sockaddr_* variants. On return, it can
be cast to the correct sockaddr type, based the protocol family contained
in its ss_family field.

namelen Indicates the amount of space pointed to by name, in bytes.

If address information for the local end of the socket is required, the
getsockname(2) function should be used instead.

If name does not point to enough space to hold the entire socket address,
the result will be truncated to namelen bytes.

RETURN VALUES
If the call succeeds, a 0 is returned and namelen is set to the actual
size of the socket address returned in name. Otherwise, errno is set and
a value of -1 is returned.

ERRORS
On failure, errno is set to one of the following:

[EBADF] The argument s is not a valid descriptor.
[ENOTSOCK] The argument s is a file, not a socket.
[ENOTCONN] The socket is not connected.
[ENOBUFS] Insufficient resources were available in the system to
perform the operation.
The name parameter points to memory not in a valid part of the process address space.

SEE ALSO
accept(2), bind(2), gethostname(2), getpeername(2), socket(2)

HISTORY
The getpeername() function call appeared in 4.2BSD.

BSD July 17, 1999 BSD

getsockname

GETSOCKNAME(2) BSD System Calls Manual GETSOCKNAME(2)

NAME
getsockname - get socket name

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);

DESCRIPTION
getsockname() returns the locally bound address information for a specified socket.

Common uses of this function are as follows:

- When bind(2) is called with a port number of 0 (indicating the kernel should pick an ephemeral port) getsockname() is used to retrieve the kernel-assigned port number.
- When a process calls bind(2) on a wildcard IP address, getsockname() is used to retrieve the local IP address for the connection.
- When a function wishes to know the address family of a socket, getsockname() can be used.

getsockname() takes three parameters:

- s, Contains the file descriptor for the socket to be looked up.
- name points to a sockaddr structure which will hold the resulting address information. Normal use requires one to use a structure specific to the protocol family in use, such as sockaddr_in (IPv4) or sockaddr_in6 (IPv6), cast to a (struct sockaddr *).

For greater portability (such as newer protocol families) the new structure sockaddr_storage exists. sockaddr_storage is large enough to hold any of the other sockaddr_* variants. On return, it should be cast to the correct sockaddr type, according to the current protocol family.

- namelen Indicates the amount of space pointed to by name, in bytes. Upon return, namelen is set to the actual size of the returned address infor-
If the address of the destination socket for a given socket connection is needed, the getpeermaname(2) function should be used instead.

If name does not point to enough space to hold the entire socket address, the result will be truncated to namelen bytes.

RETURN VALUES
On success, getsockname() returns a 0, and namelen is set to the actual size of the socket address returned in name. Otherwise, errno is set, and a value of -1 is returned.

ERRORS
If getsockname() fails, errno is set to one of the following:

[EBADF] The argument s is not a valid descriptor.
[ENOTSOCK] The argument s is a file, not a socket.
[ENOBUFS] Insufficient resources were available in the system to perform the operation.
[EFAULT] The name parameter points to memory not in a valid part of the process address space.

SEE ALSO
accept(2), bind(2), getpeermaname(2), getpeereid(2), socket(2)

BUGS
Names bound to sockets in the UNIX domain are inaccessible; getsockname returns a zero length name.

HISTORY
The getsockname() function call appeared in 4.2BSD.

BSD July 17, 1999 BSD

getsockopt

GETSOCKOPT(2) BSD System Calls Manual GETSOCKOPT(2)

NAME
getsockopt, setsockopt — get and set options on sockets

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
Chapter 42. TCP/IP Library Reference

DESCRIPTION

getsocket() and setsockopt() manipulate the options associated with a
socket. Options may exist at multiple protocol levels; they are always
present at the uppermost “socket” level.

When manipulating socket options the level at which the option resides
and the name of the option must be specified. To manipulate options at
the socket level, level is specified as SOL_SOCKET. To manipulate
options at any other level the protocol number of the appropriate proto-
col controlling the option is supplied. For example, to indicate that an
option is to be interpreted by the TCP protocol, level should be set to
the protocol number of TCP; see getprotoent(3).

The parameters optval and optlen are used to access option values for
setsockopt(). For getsockopt() they identify a buffer in which the value
for the requested option(s) are to be returned. For getsockopt(), optlen
is a value-result parameter, initially containing the size of the buffer
pointed to by optval, and modified on return to indicate the actual size
of the value returned. If no option value is to be supplied or returned,
optval may be NULL.

optname and any specified options are passed uninterpreted to the appro-
priate protocol module for interpretation. The include file
<sys/socket.h> contains definitions for socket level options, described
below. Options at other protocol levels vary in format and name; consult
the appropriate entries in section 4 of the manual.

Most socket-level options utilize an int parameter for optval. For
setsockopt(), the parameter should be non-zero to enable a boolean
option, or zero if the option is to be disabled. SO_LINGER uses a struct
linger parameter, defined in <sys/socket.h>, which specifies the desired
state of the option and the linger interval (see below). SO_SNDTIMEO and
SO_RCVTIMEO use a struct timeval parameter, defined in <sys/time.h>.

The following options are recognized at the socket level. Except as
noted, each may be examined with getsockopt() and set with setsockopt().

SO_DEBUG enables recording of debugging information
SO_REUSEADDR enables local address reuse
SO_REUSEPORT enables duplicate address and port bindings
SO_KEEPALIVE enables keep connections alive
SO_DONTROUTE enables routing bypass for outgoing messages
SO_LINGER linger on close if data present
SO_BROADCAST enables permission to transmit broadcast messages
SO_OOBINLINE enables reception of out-of-band data in band
SO_SNDBUF set buffer size for output
SO_RCVBUF set buffer size for input
SO_SNLOWAT set minimum count for output
SO_RCVLOWAT set minimum count for input
SO_SNDTIMEO set timeout value for output
SO_RCVTIMEO set timeout value for input
SO_TYPE get the type of the socket (get only)
SO_ERROR get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules.
SO_REUSEADDR indicates that the rules used in validating addresses sup-
plied in a bind(2) call should allow reuse of local addresses.
SO_REUSEPORT allows completely duplicate bindings by multiple processes
if they all set SO_REUSEPORT before binding the port. This option per-
mits multiple instances of a program to each receive UDP/IP multicast or
broadcast datagrams destined for the bound port. SO_KEEPALIVE enables
the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal when attempting to send data. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close(2) attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period measured in seconds, termed the linger interval, is specified in the setsockopt() call when SO_LINGER is requested). If SO_LINGER is disabled and a close(2) is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv(2) or read(2) calls without the MSG_OOB flag. Some protocols always behave as if this option is set. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select(2) or poll(2) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024. SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error EWOULDBLOCK if no data was sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. SO_RCVTIMEO is an option to set a timeout value for input operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for input operations to complete.
In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received.

Finally, SO_TYPE and SO_ERROR are options used only with getsockopt(). SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES
A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
The call succeeds unless:

[EBADF] The argument s is not a valid descriptor.

[ENOTSOCK] The argument s is a file, not a socket.

[ENOPROTOOPT] The option is unknown at the level indicated.

[EFAULT] The address pointed to by optval is not in a valid part of the process address space. For getsockopt(), this error may also be returned if optlen is not in a valid part of the process address space.

SEE ALSO
connect(2), ioctl(2), poll(2), select(2), poll(2), socket(2), getprotoent(3), protocols(5)

BUGS
Several of the socket options should be handled at lower levels of the system.

HISTORY
The getsockopt() system call appeared in 4.2BSD.

BSD February 15, 1999 BSD

ioctl

IOCTL(2) BSD System Calls Manual IOCTL(2)

NAME
iocntl - control device

SYNOPSIS

#include <sys/ioctl.h>

int iocntl(int d, unsigned long request, ...);

DESCRIPTION
The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests.

The argument d must be an open file descriptor. The third argument is called arg and contains additional information needed by this device to perform the requested function. arg is either an int or a pointer to a device-specific data structure, depending upon the given request.

An ioctl request has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the third argument (arg) in bytes. Macros and defines used in specifying an ioctl request are located in the file <sys/ioct1.h>.

RETURN VALUES
If an error has occurred, a value of -1 is returned and errno is set to indicate the error.

ERRORS
ioctl() will fail if:

[EBADF] d is not a valid descriptor.

[ENOTTY] d is not associated with a character special device.

[ENOTTY] The specified request does not apply to the kind of object that the descriptor d references.

[EINVAL] request or arg is not valid.

[EFAULT] arg points outside the process’s allocated address space.

SEE ALSO
cdio(1), chio(1), mt(1), execve(2), fcntl(2), intro(4), tty(4)

HISTORY
An ioctl() function call appeared in Version 7 AT&T UNIX.
The arguments are as follows:

- **fds** Points to an array of pollfd structures, which are defined as:
  ```c
  struct pollfd {
      int fd;
      short events;
      short revents;
  };
  ```

  The `fd` member is an open file descriptor. The `events` and `revents` members are bitmasks of conditions to monitor and conditions found, respectively.

- **nfds** The number of pollfd structures in the array.

- **timeout** Maximum interval to wait for the poll to complete, in milliseconds. If this value is 0, then `poll()` will return immediately. If this value is `INFTIM` (-1), `poll()` will block indefinitely until a condition is found.

The calling process sets the `events` bitmask and `poll()` sets the `revents` bitmask. Each call to `poll()` resets the `revents` bitmask for accuracy. The condition flags in the bitmasks are defined as:

- **POLLIN** Data is available on the file descriptor for reading.
- **POLLNORM** Same as **POLLIN**.
- **POLLPRI** Same as **POLLIN**.
- **POLLOUT** Data can be written to the file descriptor without blocking.
- **POLLEERR** This flag is not used in this implementation and is provided only for source code compatibility.
- **POLLHUP** The file descriptor was valid before the polling process and invalid after. Presumably, this means that the file descriptor was closed sometime during the poll.
- **POLLNVAL** The corresponding file descriptor is invalid.
- **POLLRDNORM** Same as **POLLIN**.
- **POLLRDBAND** Same as **POLLIN**.
- **POLLRNORM** Same as **POLLOUT**.
- **POLLRNDBAND** Same as **POLLOUT**.
- **POLLMMSG** This flag is not used in this implementation and is provided only for source code compatibility.

All flags except **POLLIN**, **POLLOUT**, and their synonyms are for use only in the `revents` member of the `pollfd` structure. An attempt to set any of these flags in the `events` member will generate an error condition.

In addition to I/O multiplexing, `poll()` can be used to generate simple timeouts. This functionality may be achieved by passing a null pointer for `fds`. 
WARNINGS
The POLLHUP flag is only a close approximation and may not always be accurate.

RETURN VALUES
Upon error, poll() returns a -1 and sets the global variable errno to indicate the error. If the timeout interval was reached before any events occurred, a 0 is returned. Otherwise, poll() returns the number of file descriptors for which revents is non-zero.

ERRORS
poll() will fail if:

- [EINVAL] nfds was either a negative number or greater than the number of available file descriptors.
- [EINVAL] An invalid flags was set in the events member of the pollfd structure.
- [EINVAL] The timeout passed to poll() was too large.
- [EAGAIN] Resource allocation failed inside of poll(). Subsequent calls to poll() may succeed.
- [EINTR] poll() caught a signal during the polling process.

SEE ALSO
poll(2), select(2), sysconf(3)

HISTORY
A poll() system call appeared in AT&T System V UNIX.

BSD December 13, 1994 BSD

select

SELECT(2) BSD System Calls Manual SELECT(2)

NAME
select - synchronous I/O multiplexing

SYNOPSIS
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int
select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
       struct timeval *timeout);

FD_SET(fd, &fdset);

FD_CLR(fd, &fdset);

FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);

DESCRIPTION
select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set; i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets:
FD_ZERO(fdset) initializes a descriptor set fdset to the null set.
FD_SET(fd, &fdset) includes a particular descriptor fd in fdset.
FD_CLR(fd, &fdset) removes fd from fdset. FD_ISSET(fd, &fdset) is non-zero if fd is a member of fdset, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which is normally at least equal to the maximum number of descriptors supported by the system.

If timeout is a non-null pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a null pointer, the select blocks indefinitely. To effect a poll, the timeout argument should be non-null, pointing to a zero-valued timeval structure. timeout is not changed by select(), and may be reused on subsequent calls; however, it is good style to re-initialize it before each invocation of select().

Any of readfds, writefds, and exceptfds may be given as null pointers if no descriptors are of interest.

RETURN VALUES
select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 is an error occurred. If the time limit expires, select() returns 0. If select() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS
An error return from select() indicates:

[EFAULT] One or more of readfds, writefds, or exceptfds points outside the process's allocated address space.

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO
accept(2), connect(2), gettimeofday(2), poll(2), read(2), recv(2), send(2), write(2), getdtablesize(3)

BUGS
Although the provision of getdtablesize(3) was intended to allow user
programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default bit size of fd_set is based on the symbol FD_SETSIZE (currently 256), but that is somewhat smaller than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>. The kernel will cope, and the userland libraries provided with the system are also ready for large numbers of file descriptors.

Alternatively, to be really safe, it is possible to allocate fd_set bit-arrays dynamically. The idea is to permit a program to work properly even if it is execve(2)’d with 4000 file descriptors pre-allocated. The following illustrates the technique which is used by userland libraries:

```c
fd_set *fdsr;
int max = fd;

fdsr = (fd_set *)calloc(howmany(max+1, NFDBITS),
    sizeof(fd_mask));
if (fdsr == NULL) {
    ...
    return (-1);
}
FD_SET(fd, fdsr);
if (max+1, fdsr, NULL, NULL, &tv);
...
free(fdsr);
```

Alternatively, it is possible to use the poll(2) interface. poll(2) is more efficient when the size of select()'s fd_set bit-arrays are very large, and for fixed numbers of file descriptors one need not size and dynamically allocate a memory object.

select() should probably have been designed to return the time remaining from the original timeout, if any, by modifying the time value in place. Even though some systems stupidly act in this different way, it is unlikely this semantic will ever be commonly implemented, as the change causes massive source code compatibility problems. Furthermore, recent new standards have dictated the current behaviour. In general, due to the existence of those brain-damaged non-conforming systems, it is unwise to assume that the timeout value will be unmodified by the select() call, and the caller should reinitialize it on each invocation. Calculating the delta is easily done by calling gettimeofday(2) before and after the call to select(), and using timersub() (as described in getitimer(2)).

Internally to the kernel, select() works poorly if multiple processes wait on the same file descriptor. Given that, it is rather surprising to see that many daemons are written that way (i.e., httpd(8)).

**HISTORY**

The select() function call appeared in 4.2BSD.

BSD March 25, 1994 BSD
send

NAME
send, sendto, sendmsg - send a message from a socket

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *msg, size_t len, int flags);

ssize_t sendto(int s, const void *msg, size_t len, int flags,
                const struct sockaddr *to, socklen_t tolen);

ssize_t sendmsg(int s, const struct msghdr *msg, int flags);

DESCRIPTION
send(), sendto(), and sendmsg() are used to transmit a message to another
socket. send() may be used only when the socket is in a connected state,
while sendto() and sendmsg() may be used at any time.

The address of the target is given by to with tolen specifying its size.
The length of the message is given by len. If the message is too long to
pass atomically through the underlying protocol, the error EMSGSIZE is
returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send(). Locally
detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be
transmitted, then send() normally blocks, unless the socket has been
placed in non-blocking I/O mode. The select(2) or poll(2) system calls
may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

#define MSG_OOB 0x1 /* process out-of-band data */
#define MSG_DONTROUTE 0x4 /* bypass routing, use direct interface */

The flag MSG_OOB is used to send "out-of-band" data on sockets that
support this notion (e.g., SOCK_STREAM); the underlying protocol must
also support "out-of-band" data. MSG_DONTROUTE is usually used only by
diagnostic or routing programs.

See recv(2) for a description of the msghdr structure.

RETURN VALUES
The call returns the number of characters sent, or -1 if an error
occurred.

ERRORS
send(), sendto(), and sendmsg() fail if:

[EBADF] An invalid descriptor was specified.
[ENOTSOCK] The argument s is not a socket.
[EFAULT] An invalid user space address was specified for a parameter.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EAGAIN] The socket is marked non-blocking and the requested operation would block.

[ENOBUFFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EACCES] The SO_BROADCAST option is not set on the socket, and a broadcast address was given as the destination.

[EHOSTUNREACH] The destination address specified an unreachable host.

[EINVAL] The flags parameter is invalid.

[EHOSTDOWN] The destination address specified a host that is down.

[ENETDOWN] The destination address specified a network that is down.

[ECONNREFUSED] The destination host rejected the message (or a previous one). This error can only be returned by connected sockets.

[ENOPROTOOPT] There was a problem sending the message. This error can only be returned by connected sockets.

[EDESTADDRREQ] The socket is not connected, and no destination address was specified.

[EISCONN] The socket is already connected, and a destination address was specified.

In addition, send() and sendto() may return the following error:

[EINVAL] len was larger than SSIZE_MAX.

Also, sendmsg() may return the following errors:

[EINVAL] The sum of the iov_len values in the msg_iov array overflowed an ssize_t.

[EMSGSIZE] The msg_iovlen member of msg was less than 0 or larger than IOV_MAX.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

SEE ALSO
shutdown

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int shutdown(int s, int how);

DESCRIPTION
The shutdown() call causes all or part of a full-duplex connection on the
socket associated with s to be shut down. If how is SHUT_RD, further
receives will be disallowed. If how is SHUT_WR, further sends will be
disallowed. If how is SHUT_RDWR, further sends and receives will be dis-
allowed.

RETURN VALUES
A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
The call succeeds unless:

[EINVAL] how is not SHUT_RD, SHUT_WR, or SHUT_RDWR.
[EBADF] s is not a valid descriptor.
[ENOTSOCK] s is a file, not a socket.
[ENOTCONN] The specified socket is not connected.

SEE ALSO
    connect(2), socket(2)

HISTORY
The shutdown() function call appeared in 4.2BSD. The how arguments used
to be simply 0, 1, and 2, but now have named values as specified by
X/Open Portability Guide Issue 4 ("XPG4").
socket

NAME
socket - create an endpoint for communication

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int
socket(int domain, int type, int protocol);

DESCRIPTION
socket() creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which com-
munication will take place; this selects the protocol family which should
be used. These families are defined in the include file <sys/socket.h>. The
currently understood formats are

AF_UNIX     (UNIX internal protocols),
AF_INET     (ARPA Internet protocols),
AF_INET6    (ARPA IPv6 protocols),
AF_ISO      (ISO protocols),
AF_NS       (Xerox Network Systems protocols),
AF_IPX      (Internetwork Packet Exchange), and
AF_IMPLINK  (IMP host at IMP link layer).

The socket has the indicated type, which specifies the semantics of com-
munication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based
byte streams. An out-of-band data transmission mechanism may be sup-
ported. A SOCK_DGRAM socket supports datagrams (connectionless, unreli-
able messages of a fixed (typically small) maximum length). A
SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connec-
tion-based data transmission path for datagrams of fixed maximum length;
a consumer may be required to read an entire packet with each read system
call. This facility is protocol specific, and presently implemented only
for PF_NS. SOCK_RAW sockets provide access to internal network protocols
and interfaces. The types SOCK_RAW, which is available only to the supe-
ruser, and SOCK_RDM, which is planned, but not yet implemented, are not
described here.

The protocol specifies a particular protocol to be used with the socket.
Normally only a single protocol exists to support a particular socket
type within a given protocol family. However, it is possible that many
protocols may exist, in which case a particular protocol must be speci-
fied in this manner. The protocol number to use is particular to the
communication domain in which communication is to take place; see
protocols(5). A value of 0 for protocol will let the system select an
appropriate protocol for the requested socket type.
Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets “warm” by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g., 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level options. These options are defined in the file <sys/socket.h>. setsockopt(2) and getsockopt(2) are used to set and get options, respectively.

RETURN VALUES
A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS
The socket() call fails if:

[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCES] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO
accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2),
listen(2), poll(2), read(2), recv(2), select(2), send(2), setsockopt(2),
shutdown(2), socketpair(2), write(2), getprotoent(3), netintro(4)

An Introductory 4.3 BSD Interprocess Communication Tutorial, reprinted in
UNIX Programmer’s Supplementary Documents Volume 1.

BSD Interprocess Communication Tutorial, reprinted in UNIX Programmer’s
Supplementary Documents Volume 1.

HISTORY
The socket() function call appeared in 4.2BSD.
XVI. FreeBSD TCP/IP Stack port for eCos

TCP/IP Networking for eCos now provides a complete TCP/IP networking stack, based on a recent snapshot of the FreeBSD code, released by the KAME project. The networking support is fully featured and well tested within the eCos environment.
Chapter 43. Networking Stack Features

Since this networking package is based on BSD code, it is very complete and robust. The eCos implementation includes support for the following protocols:

- IPv4
- UDP
- TCP
- ICMP
- raw packet interface
- Multi-cast addressing
- IPv6 (including UDP, ICP, ICMP)

These additional features are also present in the package, but are not supported:

- Berkeley Packet Filter
- Uni-cast support
- Multi-cast routing
Chapter 44. Freebsd TCP/IP stack port

This document describes how to get started with the Freebsd TCP/IP network stack.

Targets

A number of ethernet devices may be supported. The default configuration supports two instances of the interface by default, and you will need to write your own driver instantiation code, and supplemental startup and initialization code, if you should add additional ones.

The target for your board will normally be supplied with an ethernet driver, in which case including the network stack and generic ethernet driver package to your build will automatically enable usage of the ethernet device driver. If your target is not supplied with an ethernet driver, you will need to use loopback (see the Section called Loopback tests in Chapter 40).

Building the Network Stack

Using the Build->Packages dialog, add the packages “Networking”, “Freebsd TCP/IP Stack” and “Common Ethernet Support” to your configuration. Their package names are CYGPKG_NET, CYGPKG_NET_FREEBSD_STACK and CYGPKG_NET_ETH_DRIVERS respectively.

A short-cut way to do this is by using the “net” template if it is available for your platform.

The platform-specific ethernet device driver for your platform will be added as part of the target selection (in the Build->Templates “Hardware” item), along with the PCI I/O subsystem (if relevent) and the appropriate serial device driver.

For example, the PowerPC MBX target selection adds the package PKG_NET_QUICC_ETH_DRIVERS, and the Cirrus Logic EDB7xxx target selection adds the package CYGPKG_NET_EDB7XXX_ETH_DRIVERS. After this, eCos and its tests can be built exactly as usual.

**Note:** By default, most of the network tests are not built. This is because some of them require manual intervention, i.e. they are to be run “by hand”, and are not suitable for automated testing. To build the full set of network tests, set the configuration option CYGPKG_NET_BUILD_TESTS “Build networking tests (demo programs)” within “Networking support build options”.

415
Chapter 44. FreeBSD TCP/IP stack port
Chapter 45. APIs

Standard networking

The APIs for the standard networking calls such as `socket()`, `recv()` and so on, are in header files relative to the top-level include directory, within the standard subdirectories as conventionally found in `/usr/include`. For example:

```sh
install/include/arpa/tftp.h
install/include/netinet/tcpip.h
install/include/sys/socket.h
install/include/sys/socketvar.h
install/include/sys/sockio.h
```

`network.h` at the top level defines various extensions, for example the API `init_all_network_interfaces(void)` described above. We advise including `network.h` whether you use these features or not.

In general, using the networking code may require definition of two symbols: `_KERNEL` and `__ECOS`. `_KERNEL` is not normally required; `__ECOS` is normally required. So add this to your compile lines for files which use the network stack:

```sh
-D__ECOS
```

To expand a little, it’s like this because this is a port of a standard distribution external to eCos. One goal is to perturb the sources as little as possible, so that upgrading and maintenance from the external distribution is simplified. The `__ECOS` symbol marks out the eCos additions in making the port. The `_KERNEL` symbol is traditional UNIX practice; it distinguishes a compilation which is to be linked into the kernel from one which is part of an application. eCos applications are fully linked, so this distinction does not apply. `_KERNEL` can however be used to control the visibility of the internals of the stack, so depending on what features your application uses, it may or may not be necessary.

The include file `network.h` undefines `_KERNEL` unconditionally, to provide an application-like compilation environment. If you were writing code which, for example, enumerates the stack’s internal structures, that is a kernel-like compilation environment, so you would need to define `_KERNEL` (in addition to `__ECOS`) and avoid including `network.h`.

Enhanced Select()

The network stack supports an extension to the standard select semantics which allows all threads that are waiting to be restarted even if the select conditions are not satisfied.

The standard select() API:

```c
int
select(int nfd,
       fd_set *in, fd_set *out, fd_set *ex,
       struct timeval *tv);
```
does not support the restart.

The additional API:

```c
int
cyg_select_with_abort(int nfd,
        fd_set *in, fd_set *out, fd_set *ex,
        struct timeval *tv)
```

behaves exactly as `select()` with the additional feature that a call to

```c
void cyg_select_abort(void)
```

will cause all threads waiting in any `cyg_select_with_abort()` call to cease waiting and continue execution.
XVII. OpenBSD TCP/IP Stack port for eCos

TCP/IP Networking for eCos now provides a complete TCP/IP networking stack, which is derived from a recent stable release of OpenBSD. The networking support is fully featured and well tested within the eCos environment.
Chapter 46. Networking Stack Features

Introduction

Since this networking package is based on BSD code, it is very complete and robust. The eCos implementation includes support for the following protocols:

- IPv4
- UDP
- TCP
- ICMP
- raw packet interface

Along with support for the above mentioned protocols, the OpenBSD stack also supports ethernet bridging.

The following additional features are also present in the package, but are not supported:

- Berkeley Packet Filter
- Multi-cast and uni-cast support, including multi-cast routing
- IPv6
Chapter 47. OpenBSD TCP/IP stack port

This document describes how to get started with the OpenBSD TCP/IP network stack.

Targets

A number of ethernet devices may be supported. The default configuration supports two instances of the interface by default, and you will need to write your own driver instantiation code, and supplemental startup and initialization code, if you should add additional ones.

The target for your board will normally be supplied with an ethernet driver, in which case including the network stack and generic ethernet driver package to your build will automatically enable usage of the ethernet device driver. If your target is not supplied with an ethernet driver, you will need to use loopback (see the Section called Loopback tests in Chapter 40).

Building the Network Stack

Using the Build->Packages dialog, add the packages “Networking”, “OpenBSD TCP/IP Stack” and “Common Ethernet Support” to your configuration. Their package names are CYGPKG_NET, CYGPKG_NET_OPENBSD_STACK and CYGPKG_NET_ETH_DRIVERS respectively.

A short-cut way to do this is by using the “net” template if it is available for your platform.

The platform-specific ethernet device driver for your platform will be added as part of the target selection (in the Build->Templates “Hardware” item), along with the PCI I/O subsystem (if relevent) and the appropriate serial device driver.

For example, the PowerPC MBX target selection adds the package PKG_NET_QUICC_ETH_DRIVERS, and the Cirrus Logic EDB7xxx target selection adds the package CYGPKG_NET_EDB7XXX_ETH_DRIVERS. After this, eCos and its tests can be built exactly as usual.

Note: By default, most of the network tests are not built. This is because some of them require manual intervention, i.e. they are to be run “by hand”, and are not suitable for automated testing. To build the full set of network tests, set the configuration option CYGPKG_NET_BUILD_TESTS “Build networking tests (demo programs)” within “Networking support build options”.

Inclusion of bridge code

The OpenBSD stack does not by default result in the inclusion of the bridge code. To include bridging functionality the CDL option CYGPKG_NET_BRIDGE must be enable. It is also possible to enable more than one concurrent bridge. The number of bridges active in a device is configured by the CDL option CYGNUM_NET_BRIDGES, which has a default value of 1.

The default behavior of a bridge is to operate without the spanning tree protocol. When devices are operated in this default mode it must be ensured the network topology is loop-free. Any loops will cause broadcast storms.
and general mayhem on the network. Including spanning tree code during the build process will allow the bridge to communicate with other bridges. They can then detect such loops and by disabling selected interfaces break the loop. To enable spanning tree enable the CDL option CYGPKG_NET_BRIDGE_STP_CODE.
Chapter 48. APIs

Standard networking

The APIs for the standard networking calls such as `socket()`, `recv()` and so on, are in header files relative to the top-level include directory, within the standard subdirectories as conventionally found in `/usr/include`. For example:

```
install/include/arpa/tftp.h
install/include/netinet/tcpip.h
install/include/sys/socket.h
install/include/sys/socketvar.h
install/include/sys/sockio.h
```

`network.h` at the top level defines various extensions, for example the API `init_all_network_interfaces(void)` described above. We advise including `network.h` whether you use these features or not.

In general, using the networking code may require definition of two symbols: `_KERNEL` and `__ECOS`. `_KERNEL` is not normally required; `__ECOS` is normally required. So add this to your compile lines for files which use the network stack:

```
-D__ECOS
```

To expand a little, it’s like this because this is a port of a standard distribution external to Red Hat. One goal is to perturb the sources as little as possible, so that upgrading and maintenance from the external distribution is simplified. The `__ECOS` symbol marks out Red Hat’s additions in making the port. The `_KERNEL` symbol is traditional UNIX practice: it distinguishes a compilation which is to be linked into the kernel from one which is part of an application. eCos applications are fully linked, so this distinction does not apply. `_KERNEL` can however be used to control the visibility of the internals of the stack, so depending on what features your application uses, it may or may not be necessary.

The include file `network.h` undefines `_KERNEL` unconditionally, to provide an application-like compilation environment. If you were writing code which, for example, enumerates the stack’s internal structures, that is a kernel-like compilation environment, so you would need to define `_KERNEL` (in addition to `__ECOS`) and avoid including `network.h`.

Enhanced Select()

The network stack supports an extension to the standard select semantics which allows all threads that are waiting to be restarted even if the select conditions are not satisfied.

The standard select() API:

```
int select(int nfd,
           fd_set *in, fd_set *out, fd_set *ex,
           struct timeval *tv);
```
Chapter 48. APIs

does not support the restart.

The additional API:

```c
int cyg_select_with_abort(int nfd,
    fd_set *in, fd_set *out, fd_set *ex,
    struct timeval *tv)
```

behaves exactly as select() with the additional feature that a call to

```c
void cyg_select_abort(void)
```

will cause all threads waiting in any cyg_select_with_abort() call to cease waiting and continue execution.

OpenBSD networking facilities

NAME

networking - introduction to networking facilities

SYNOPSIS

```c
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>
```

DESCRIPTION

This section is a general introduction to the networking facilities available in the system. The general introduction on this page is broken up into three areas: protocol families (domains), protocols, and network interfaces.

All network protocols are associated with a specific protocol family. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per socket type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in socket. A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families and/or address formats.
Chapter 48. APIs

PROTOCOL

The system currently supports the Internet protocols. Raw socket interfaces are provided to the IP protocol layer of the Internet.

ADDRESSING

Associated with each protocol family is an address format. All network addresses adhere to a general structure, called a sockaddr, described below. However, each protocol imposes a finer, more specific structure, generally renaming the variant, which is discussed in the protocol family manual page alluded to above.

```
struct sockaddr {
    u_int8_t sa_len;
    sa_family_t sa_family;
    char sa_data[14];
};
```

The field sa_len contains the total length of the structure, which may exceed 16 bytes. The following address values for sa_family are known to the system (and additional formats are defined for possible future implementation):

- #define AF_UNIX 1 /* local to host (pipes, portals) */
- #define AF_INET 2 /* internetwork: UDP, TCP, etc. */
- #define AF_NS 6 /* Xerox NS protocols */
- #define AF_CCITT 10 /* CCITT protocols, X.25 etc */
- #define AF_HYLINK 15 /* NSC Hyperchannel */
- #define AF_APPLETALK 16 /* AppleTalk */
- #define AF_ISO 18 /* ISO protocols */
- #define AF_IPX 23 /* Novell Internet Protocol */
- #define AF_INET6 24 /* IPv6 */
- #define AF_NATM 27 /* native ATM access */

ROUTING

OpenBSD provides some packet routing facilities. The kernel maintains a routing information database, which is used in selecting the appropriate network interface when transmitting packets.

This facility is however, untested in eCos ports.

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, lo, do not.

The following ioctl calls may be used to manipulate network interfaces. The ioctl is made on a socket (typically of type SOCK_DGRAM) in the desired domain. Most of the requests supported in earlier releases take an ifreq structure as its parameter. This structure has the form

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
    }
};
```
Chapter 48. APIs

```c

caddr_t ifru_data;
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags /* flags */
#define ifr_metric ifr_ifru.ifru_metric /* metric */
#define ifr_media ifr_ifru.ifru_metric /* media options (overload) */
#define ifr_data ifr_ifru.ifru_data /* for use by interface */
```

Calls which are now deprecated are:

**SIOCSIFADDR** Set interface address for protocol family. Following the address assignment, the “initialization” routine for the interface is called.

**SIOCSIFDSTADDR** Set point to point address for protocol family and interface.

**SIOCSIFBRDADDR** Set broadcast address for protocol family and interface.

 ioctl requests to obtain addresses and requests both to set and retrieve other data are still fully supported and use the ifreq structure:

**SIOCGIFADDR** Get interface address for protocol family.

**SIOCGIFDSTADDR** Get point to point address for protocol family and interface.

**SIOCGIFBRDADDR** Get broadcast address for protocol family and interface.

**SIOCSIFFLAGS** Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

**SIOCGIFFLAGS** Get interface flags.

**SIOCSIFMEDIA** Set interface media. See ifmedia(4) for possible values.

**SIOCGIFMEDIA** Get interface media. See ifmedia(4) for interpreting this value.

**SIOCSIFMETRIC** Set interface routing metric. The metric is used only by user-level routers.

**SIOCGIFMETRIC** Get interface metric.

There are two requests that make use of a new structure:

**SIOCAIFADDR** An interface may have more than one address associated with it in some protocols. This request provides a means to add additional addresses (or modify characteristics of the primary address if the default address for the address family is specified). Rather than making separate calls to set destination or broadcast addresses, or network masks (now an integral feature of multiple protocols) a separate structure is used to specify all three facets simultaneously (see below). One would use a
slightly tailored version of this struct specific to each family (replacing each sockaddr by one of the family-specific type). Where the sockaddr itself is larger than the default size, one needs to modify the ioctl(2) identifier itself to include the total size, as described in ioctl(2).

SIOCDIFADDR This request deletes the specified address from the list associated with an interface. It also uses the if_aliasreq structure to allow for the possibility of protocols allowing multiple masks or destination addresses, and also adopts the convention that specification of the default address means to delete the first address for the interface belonging to the address family in which the original socket was opened.

SIOCGIFCONF Get interface configuration list. This request takes an ifconf structure (see below) as a value-result parameter. The ifc_len field should be initially set to the size of the buffer pointed to by ifc_buf. On return it will contain the length, in bytes, of the configuration list. Alternately, if the ifc_len passed in is set to 0, SIOCGIFCONF will set ifc_len to the size that ifc_buf needs to be to fit the entire configuration list and not fill in the other parameters. This is useful for determining the exact size that ifc_buf needs to be in advance. Note, however, that this is an extension that not all operating systems support.

/*
 * Structure used in SIOCAIFADDR request.
 */
struct ifaliasreq {
    char ifra_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    struct sockaddr ifra_addr;
    struct sockaddr ifra_broadaddr;
    struct sockaddr ifra_mask;
};

/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct ifconf {
    int ifc_len; /* size of associated buffer */
    union {
        caddr_t ifc_ifcu.ifcu_buf;
        struct ifreq *ifc_ifcu.ifc_req;
    } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifc_ifcu.ifc_buf /* buffer address */
#define ifc_req ifc_ifcu.ifc_ifcu.ifc_req /* array of structures returned */
};

SEE ALSO
bridge, spanning-tree
NAME
bridge - Ethernet bridge interface

SYNOPSIS
pseudo-device bridge

DESCRIPTION
The bridge device creates a logical link between two or more Ethernet interfaces. This link between the interfaces selectively forwards frames from each interface on the bridge to every other interface on the bridge. A bridge can serve several services, including isolation of traffic between sets of machines so that traffic local to one set of machines is not available on the wire of another set of machines, and it can act as a transparent filter for ip4 datagrams.

The bridges provided by this interface are learning bridges with filtering. In general a bridge works like a hub, forwarding traffic from one interface to another. It differs from a hub in that it will "learn" which machines are on each of its attached segments by actively listening to incoming traffic and examining the headers of each frame. A table is built containing the MAC address and segment to which the MAC address is attached. This allows a bridge to be more selective about what it forwards, which can be used to reduce traffic on a set of segments and also to provide an IP firewall without changing the topology of the network.

The algorithm works as follows by default, but can be modified via ioctl. When a frame comes in, the origin segment and the source address are recorded. If the bridge has no knowledge about where the destination is to be found, the bridge will forward the frame to all attached segments. If the destination is known to be on a different segment from its origin, the bridge will forward the packet only to the destination segment. If the destination is on the same segment as the origin segment, the bridge will drop the packet because the receiver has already had a chance to see the frame.

IOCTLS
A bridge interface responds to all of the ioctl calls specific to other interfaces listed in netintro. The following ioctl calls are specific to bridge devices. They are defined in <sys/sockio.h>.

SIOCBRDGIFS (struct ifbifconf) Retrieve member interface list from a bridge. This request takes an ifbifconf structure (see below) as a value-result parameter. The ifbic_len field should be initially set to the size of the buffer pointed to by ifbic_buf. On return it will contain the length, in bytes, of the configuration list. Alternatively, if the ifbic_len passed in is set to 0, SIOCBRDGIFS will set ifbic_len to the size that ifbic_buf needs to be to fit the entire configuration list, and will not fill in the other parameters. This is useful for determining the exact size that ifbic_buf needs to be in advance.

The argument structure is defined as follows:

```c
struct ifbreq {
    char ifbr_name[IFNAMSIZ]; /* brdg nam */
    char ifbr_ifsname[IFNAMSIZ]; /* if name */
};
```
Chapter 48. APIs

u_int32_t ifbr_ifsflags; /* if flags */
u_int8_t ifbr_state; /* member stp state */
u_int8_t ifbr_priority; /* member stp priority */
u_int8_t ifbr_portno; /* member port number */
u_int32_t ifbr_path_cost; /* member stp path cost */
}

#define IFBIF_LEARNING 0x0001 /* ifs can learn */
#define IFBIF_DISCOVER 0x0002 /* ifs sends packets w/unknown dest */
#define IFBIF_BLOCKNONIP 0x0004 /* ifs blocks non-IP/ARP in/out */
#define IFBIF_STP 0x0008 /* ifs participates in spanning tree */
#define IFBIF_SPAN 0x0100 /* ifs is a span port (ro) */
#define IFBIF_RO_MASK 0xff00 /* read only bits */

struct ifbconf {
    char ifbic_name[IFNAMSIZ]; /* brdg name */
    u_int32_t ifbic_len; /* buf size */
    union {
        caddr_t ifbicu_buf; /* buffer */
        struct ifbreq *ifbicu_req;
    } ifbic_ifbicu;
    #define ifbic_buf ifbic_ifbicu.ifbicu_buf
    #define ifbic_req ifbic_ifbicu.ifbicu_req
};

SIOCBRDGADD (struct ifbreq) Add the interface named in ifbr_ifsname to the bridge named in ifbr_name.

SIOCBRDGDEL (struct ifbreq) Delete the interface named in ifbr_ifsname from the bridge named in ifbr_name.

SIOCBRDGADDS (struct ifbreq) Add the interface named in ifbr_ifsname as a span port to the bridge named in ifbr_name.

SIOCBRDGDELS (struct ifbreq) Delete the interface named in ifbr_ifsname from the list of span ports of the bridge named in ifbr_name.

SIOCBRDGSIFFLGS (struct ifbreq) Set the bridge member interface flags for the interface named in ifbr_ifsname attached to the bridge ifbr_name. If the flag IFBIF_LEARNING is set on an interface, source addresses from frames received on the interface are recorded in the address cache. If the flag IFBIF_DISCOVER is set, the interface will receive packets destined for unknown destinations, otherwise a frame that has a destination not found in the address cache is not forwarded to this interface. The default for newly added interfaces has both flags set. If the flag IFBIF_BLOCKNONIP is set, packets that are one of ip(4), ip6(4), arp(4), or Reverse ARP, will not be bridged from and to the interface.

SIOCBRDGGIFFLGS Retrieve the bridge member interface flags for the interface named in ifbr_ifsname attached to the bridge ifbr_name.

SIOCBRDGRTS (struct ifbaconf) Retrieve the address cache of the bridge named in ifbac_name. This request takes an ifbaconf structure (see below) as a value result parameter. The ifbac_len field should be initially set to the size of the buffer pointed to by ifbac_buf. On return,
it will contain the length, in bytes, of the configuration list. Alternatively, if the ifbac_len passed in is set to 0, SIOCBRDBGRTS will set it to the size that ifbac_buf needs to be to fit the entire configuration list and not fill in the other parameters. As with SIOCBRDBGIFS, this is useful for determining the exact size that ifbac_buf needs to be in advance.

The argument structure is defined as follows:

```c
struct ifbareq {
    char ifba_name[IFNAMSIZ]; /* brdg nam */
    char ifba_ifsname[IFNAMSIZ]; /* dest ifs */
    u_int8_t ifba_age; /* addr age */
    u_int8_t ifba_flags; /* addr flag */
    struct ether_addr ifba_dst; /* dst addr */
};
```

```c
#define IFBAF_TYPEMASK 0x03 /* addr type mask */
#define IFBAF_DYNAMIC 0x00 /* dynamic addr */
#define IFBAF_STATIC 0x01 /* static address */
```

```c
struct ifbaconf {
    char ifbac_name[IFNAMSIZ]; /* brdg name */
    u_int32_t ifbac_len; /* buf size */
    union {
        caddr_t ifbacu_buf; /* buf */
        struct ifbareq *ifbacu_req;
    } ifbac_ifbacu;
} ifbac_ifbacu;
```

Address cache entries with the type set to IFBAF_DYNAMIC in ifba_flags are entries learned by the bridge. Entries with the type set to IFBAF_STATIC are manually added entries.

**SIOCBRDGSADDR** (struct ifbareq) Add an entry, manually, to the address cache for the bridge named in ifba_name. The address and its associated interface and flags are set in the ifba_dst, ifba_ifsname, and ifba_flags fields, respectively.

**SIOCBRDGDADDR** (struct ifbareq) Delete an entry from the address cache of the bridge named in ifba_name. Entries are deleted strictly based on the address field ifba_dst.

**SIOCBRDGSCACHE** (struct ifbcachereq) Set the maximum address cache size for the bridge named in ifbc_name to ifbc_size entries.

The argument structure is as follows:

```c
struct ifbcachereq {
    char ifbc_name[IFNAMSIZ]; /* bridge */
    u_int32_t ifbc_size; /* size */
};
```

**SIOCBRDGGCACHE** (struct ifbcachereq) Retrieve the maximum size of the address cache for the bridge ifbc_name.

**SIOCBRDGSTO** (struct ifbcachetoreq) Set the time, in seconds, that
addresses which have not been seen on the network (transmitted a packet) remain in the cache. If the time is set to zero, no aging is performed on the address cache. The argument structure is as follows:

```c
struct ifbcachetoreq {
    char ifbct_name[IFNAMSIZ]; /* bridge */
    u_int32_t ifbct_time;  /* time */
};
```

SIOCBRDGGTO (struct ifbcachetoreq) Retrieve the address cache expiration time (see above).

SIOCBRDGFLUSH (struct ifbreq) Flush addresses from the cache. ifbr_name contains the name of the bridge device, and ifbr_ifsflags should be set to IFBF_FLUSHALL to flush all addresses from the cache or IFBF_FLUSHdyn to flush only the dynamically learned addresses from the cache.

SIOCBRDGARL (struct ifbrlreq) Add an Ethernet address filtering rule to the bridge on a specific interface. ifbr_name contains the name of the bridge device, and ifbr_ifsname contains the name of the bridge member interface. The ifbr_action field is one of BRL_ACTION_PASS or BRL_ACTION_BLOCK, to pass or block matching frames respectively. The ifbr_flags specifies whether the rule should match on input, output, or both be using the flags BRL_FLAG_IN and BRL_FLAG_OUT. It also specifies whether either (or both) of the source and destination addresses should be matched by using the BRL_FLAG_SRCVALID and BRL_FLAG_DSTVALID flags. The ifbr_src field is the source address that triggers the rule (only considered if ifbr_flags has the BRL_FLAG_SRCVALID bit set). The ifbr_dst field is the destination address that triggers the rule (only considered if ifbr_flags has the BRL_FLAG_DSTVALID bit set).

The argument structure is as follows:

```c
struct ifbrlreq {
    char ifbr_name[IFNAMSIZ];
    char ifbr_ifsname[IFNAMSIZ];
    u_int8_t ifbr_action;
    u_int8_t ifbr_flags;
    struct ether_addr ifbr_src;
    struct ether_addr ifbr_dst;
    char ifbr_tagname[PF_TAG_NAME_SIZE];
};
```

#define BRL_ACTION_BLOCK 0x01
#define BRL_ACTION_PASS 0x02
#define BRL_FLAG_IN 0x08
#define BRL_FLAG_OUT 0x04

SIOCBRDGFRL (struct ifbrlreq) Remove all filtering rules from a bridge interface member. ifbr_name contains the name of the bridge device, and ifbr_ifsname contains the name of the bridge member interface.

SIOCBRDGGRL (struct ifbrlconf) Retrieve all of the rules from the bridge, ifbrl_name, for the member interface, ifbrl_ifsname. This request takes an ifbrlconf struc-
Chapter 48. APIs

ture (see below) as a value result parameter. The ifbrl_len field should be initially set to the size of the buffer pointed to by ifbrl_buf. On return, it will contain the length, in bytes, of the configuration list. Alternatively, if the ifbrl_len passed in is set to 0, SIOCBRDGRL will set it to the size that ifbrl_buf needs to be to fit the entire configuration list and not fill in the other parameters. As with SIOCBRDGIFS, this is useful for determining the exact size that ifbrl_buf needs to be in advance.

The argument structure is defined as follows:

```c
struct ifbrlconf {
    char ifbrl_name[IFNAMSIZ]; /* brdg nam */
    char ifbrl_ifsname[IFNAMSIZ]; /* ifs name */
    u_int32_t ifbr_len; /* buf len */
    union {
        caddr_t ifbrlu_buf;
        struct ifbrlreq *ifbrlu_req;
    } ifrl_ifbrlu;
} ifbrl_ifbrlu;
#define ifbrl_buf ifbrl_ifbrlu.ifbrlu_buf
#define ifbrl_req ifbrl_ifbrlu.ifbrlu_req
};
```

SIOCBRDGARL (struct ifbrlreq) Add a filtering rule to the bridge named in ifbr_name on the interface named in ifbr_ifsname. The argument structure is as follows:

```c
struct ifbrlreq {
    char ifbr_name[IFNAMSIZ]; /* bridge */
    char ifbr_ifsname[IFNAMSIZ]; /* ifs */
    u_int8_t ifbr_action; /* handling */
    u_int8_t ifbr_flags; /* flags */
    struct ether_addr ifbr_src; /* src mac */
    struct ether_addr ifbr_dst; /* dst mac */
};
#define BRL_ACTION_BLOCK 0x01
#define BRL_ACTION_PASS 0x02
#define BRL_FLAG_IN 0x08
#define BRL_FLAG_OUT 0x04
#define BRL_FLAG_SRCVALID 0x02
#define BRL_FLAG_DSTVALID 0x01
```

Rules are applied in the order in which they were added to the bridge, and the first matching rule’s action parameter determines the fate of the packet. The ifbr_action parameter specifies whether a frame matching the rule is to be blocked or passed.

If the BRL_FLAG_IN bit is set in ifbr_flags, then the rule applies to frames received by the interface. If the BRL_FLAG_OUT bit is set, then the rule applies to frame transmitted by the interface. At least one of BRL_FLAG_IN or BRL_FLAG_OUT must be set.

The source Ethernet address in ifbr_src is checked if the BRL_FLAG_SRCVALID bit is set in ifbr_flags. The destination address in ifbr_dst is checked if the BRL_FLAG_DSTVALID bit is set. If neither bit is set, the rule matches all frames.
SIOCBRDGFRL (struct ifbrlreq) Flush rules from the bridge ifbr_name on the interface ifbr_ifsname.

SIOCBRDGGRL (struct ifbrlconf) Retrieve an array of rules from the bridge for a particular interface. This request takes an ifbrlconf structure (see below) as a value-result parameter. The ifbrl_len field should be initially set to the size of the buffer pointed to by ifbrl_buf. On return it will contain the length, in bytes, of the rule list. Alternatively, if the ifbrl_len passed in is set to 0, SIOCBRDGGRL will set ifbrl_len to the size that ifbrl_buf needs to be to fit the entire configuration list, and will not fill in the other parameters. This is useful for determining the exact size that ifbrl_buf needs to be in advance.

The argument structure is as follows:

```c
struct ifbrlconf {
    char ifbrl_name[IFNAMSIZ]; /* bridge */
    char ifbrl_ifsname[IFNAMSIZ]; /* member */
    u_int32_t ifbrl_len; /* buflen */
    union {
        caddr_t ifbrlu_buf;
        struct ifbrlreq *ifbrlu_req;
    } ifbrl_ifbrlu;
}
```

```c
#define ifbrl_buf ifbrl_ifbrlu.ifbrlu_buf
#define ifbrl_req ifbrl_ifbrlu.ifbrlu_req
```

**ERRORS**

If the ioctl call fails, errno is set to one of the following values:

- **[ENOENT]** For an add request, this means that the named interface is not configured into the system. For a delete operation, it means that the named interface is not a member of the bridge. For an address cache deletion, the address was not found in the table.

- **[ENOMEM]** Memory could not be allocated for an interface or cache entry to be added to the bridge.

- **[EEXIST]** The named interface is already a member of the bridge.

- **[EBUSY]** The named interface is already a member of another bridge.

- **[EINVAL]** The named interface is not an Ethernet interface or an invalid ioctl was performed on the bridge.

- **[ENETDOWN]** Address cache operation (flush, add, delete) on a bridge that is in the down state.

- **[ESRCH]** No such member interface in the bridge.

**SEE ALSO**

netintro, spanning-tree

**AUTHORS**

The bridge kernel interface was written by Jason L. Wright

<jason@thought.net> as part of an undergraduate independent study at
Spanning Tree Protocol

NAME
stp - Spanning Tree Protocol

SYNOPSIS

DESCRIPTION
Spanning Tree (STP) is a layer 2 protocol designed to run on bridges. The main purpose of STP is to ensure that there are no loop situations when redundant paths are provisioned in the network. STP detects and disables creation of network loops by blocking certain ports on some of the bridges in the network. The process of selection of blocking ports (on occurrence of redundant paths) is governed by the following three parameters:

- Relative priority of each bridge. A higher value means lower priority.
- Relative priority of each port within a bridge. A higher value means lower priority.
- Path cost (based on physical media type) associated with each port.

A given port/interface participates in STP if the flag IFBIF_STP is set for the interface. A possible time for setting this flag is at the time when the interface in context is added to bridge.

IOCTLS
The STP code is invoked for all ioctl calls specified in bridge section. The following ioctl calls are specific to STP functionality. They are defined in `<sys/sockio.h>`.

SIOC_BRDGGPRI (struct ifbrparam) Get the configured priority of this bridge. The priority value could vary from 0 to 65535. 0 being the highest priority and 65535 the lowest. The configured value is returned in field ifbrp_prio.

SIOC_BRDGSPI (struct ifbrparam) Set priority of this bridge to the value specified in field ifbrp_prio.

SIOC_BRDGGHT (struct ifbrparam) Get the configured frequency of transmission of hello packets from non-blocking interfaces on this bridge. The configured frequency is returned in field ifbrp_hellotime.

SIOC_BRDGSHT (struct ifbrparam) Set the frequency of transmission of hello packets to the value specified in field ifbrp_hellotime. The specified value should be greater than 0, else EINVAL is returned.

SIOC_BRDGGFD (struct ifbrparam) Get the forwarding delay time associated with ports/interfaces on this bridge. The forwarding delay time is the time taken by a port to transit from one state to other (for eg. from LEARNING state to FORWARDING state). The configured value if returned in field ifbrp-fwddelay.

SIOC_BRDGSFD (struct ifbrparam) Set the forwarding delay for ports.
attached to this bridge to a value specified in field ifbrp_fwddelay. The specified value should be greater than 0, else EINVAL is returned.

**SIOCBRDGGMA**
(struct ifbrparam) Get aging timeout values of spanning data. The timeout value is returned in field ifbrp_maxage.

**SIOCBRDGSMA**
(struct ifbrparam) Set the aging timeout value of BPDUs to the value specified in ifbrp_maxage. The specified value should be greater than 0, else EINVAL is returned.

**SIOCBRDGSIFPRIO**
(struct ifbreq) Set the priority of specified interface to the value given in field ifbr_priority.

**SIOCBRDGSIFCOST**
(struct ifbreq) Set the cost associated with the given interface to the value specified in field ifbr_path_cost.

**ERRORS**
Same as the ones specified for bridge.

**SEE ALSO**
netintro, bridge
Chapter 48. APIs
XVIII. DNS for eCos and RedBoot

eCos and RedBoot can both use the DNS package to perform network name lookups.
Chapter 49. DNS

DNS API

The DNS client uses the normal BSD API for performing lookups: `gethostbyname()`, `gethostbyaddr()`, `getaddrinfo()`, `getnameinfo()`.

There are a few restrictions:

- If the DNS server returns multiple authoritative records for a host name to `gethostbyname`, the hostent will only contain a record for the first entry. If multiple records are desired, use `getaddrinfo`, which will return multiple results.

- The code has been made thread safe. ie multiple threads may call `gethostbyname()` without causing problems to the hostent structure returned. What is not safe is one thread using both `gethostbyname()` and `gethostbyaddr()`. A call to one will destroy the results from the previous call to the other function. `getaddrinfo()` and `getnameinfo()` are thread safe and so these are the preferred interfaces. They are also address family independent so making it easier to port code to IPv6.

- The DNS client will only return IPv4 addresses to RedBoot. At the moment this is not really a limitation, since RedBoot only supports IPv4 and not IPv6.

To initialise the DNS client the following function must be called:

```c
#include <network.h>
int cyg_dns_res_start(char * dns_server)
```

Where `dns_server` is the address of the DNS server. The address must be in numeric form and can be either an IPv4 or an IPv6 address.

There also exists a deprecated function to start the DNS client:

```c
int cyg_dns_res_init(struct in_addr *dns_server)
```

where `dns_server` is the address of the DNS server the client should query. The address should be in network order and can only be an IPv4 address.

On error both this function returns -1, otherwise 0 for success. If lookups are attempted before this function has been called, they will fail and return NULL, unless numeric host addresses are passed. In this cause, the address will be converted and returned without the need for a lookup.

A default, hard coded, server may be specified in the CDL option `CYGDAT_NS_DNS_DEFAULT_SERVER`. The use of this is controlled by `CYGPKG_NS_DNS_DEFAULT`. If this is enabled, `init_all_network_interfaces()` will initialize the resolver with the hard coded address. The DHCP client or user code may override this address by calling `cyg_dns_res_init` again.

The DNS client understands the concepts of the target being in a domain. By default no domain will be used. Host name lookups should be for fully qualified names. The domain name can be set and retrieved using the functions:

```c
int getdomainname(char *name, size_t len);
int setdomainname(const char *name, size_t len);
```
Alternatively, a hard coded domain name can be set using CDL. The boolean `CYGPKG_NS_DNS_DOMAINNAME` enables this and the domain name is taken from `CYGPKG_NS_DNS_DOMAINNAME_NAME`.

Once set, the DNS client will use some simple heuristics when deciding how to use the domain name. If the name given to the client ends with a "." it is assumed to be a FQDN and the domain name will not be used. If the name contains a "." somewhere within it, first a lookup will be performed without the domain name. If that fails the domain name will be appended and looked up. If the name does not contain a ".", the domain name is appended and used for the first query. If that fails, the unadorned name is lookup.

The `getaddrinfo` will return both IPv4 and IPv6 addresses for a given host name, when IPv6 is enabled in the eCos configuration. The CDL option `CYGOPT_NS_DNS_FIRST_FAMILY` controls the order IPv6 and IPv4 addresses are returned in the linked list of `addrinfo` structures. If the value `AF_INET` is used, the IPv4 addresses will be first. If the value `AF_INET6`, which is the default, is used, IPv6 address will be first. This ordering will control how clients attempt to connect to servers, i.e. using IPv6 or IPv4 first.

### DNS Client Testing

The DNS client has a test program, `dns1.c`, which tests many of the features of the DNS client and the functions `gethostbyname()`, `gethostbyaddr()`, `getaddrinfo()`, `getnameinfo()`.

In order for this test to work, a DNS server must be configured with a number of names and addresses. The following is an example forward address resolution database for bind v9, which explains the requirements.

```
$TTL 680400
@ IN SOA lunn.org. andrew.lunn.lunn.org ( 2003041801 ; serial
10800 ; refresh
1800 ; retry
3600000 ; expire
2592000 ; minimum
)
IN NS londo.lunn.org.

hostnamev4 IN A 192.168.88.1
cnamev4 IN CNAME hostnamev4
hostnamev6 IN AAAA fec0::88:4:3:2:1
cnamev6 IN CNAME hostnamev6
hostnamev46 IN A 192.168.88.2
hostnamev46 IN AAAA fec0::88:4:3:2:2
cnamev46 IN CNAME hostnamev46
```

The actual names and addresses do not matter, since they are configurable in the test. What is important is the relationship between the names and the addresses and there family, i.e. `hostnamev4` should map to one IPv4 address, `hostnamev46` should map to both an IPv4 and an IPv6 address. `cnamev4` should be a `CNAME` record for `hostnamev4`. Reverse lookup information is also needed by the test.

The information placed into the DNS server is also need in the test case. A structure is defined to hold this information:

```c
struct test_info_s {
    char * dns_server_v4;
    char * dns_server_v6;
    char * domain_name;
    char * hostname_v4;
    char * cname_v4;
    char * ip_addr_v4;
    char * hostname_v6;
}
```
char * cname_v6;
char * ip_addr_v6;
char * hostname_v46;
char * cname_v46;
char * ip_addr_v46_v4;
char * ip_addr_v46_v6;
};

The test program may hold a number of such structures for different DNS server. The test will use each structure in turn to perform the tests. If IPv6 is not enabled in the eCos configuration, the entries which use IPv6 may be assigned to NULL.
XIX. IPSEC for eCos

The FreeBSD network stack which is part of eCos can be configured to use IPSEC to provide more secure communications between Internet hosts. IPSEC can be used with both IPv4 and IPv6.
Chapter 50. Installation and Configuration

Due to the restrictions imposed by various countries on the exportation and importation of cryptographic software, it has been decided to distribute the crypto parts of FreeBSD IPSEC separately from the rest of eCos. Before IPSEC can be enabled the eCos package bsd_crypto must be installed. This package can be found on the eCosCentric server at ftp.ecoscentric.com:/pub/contrib (ftp://ftp.ecoscentric.com:/pub/contrib). Once the package has been downloaded it must be installed using the ecosadmin.tcl script.

When the bsd_crypto package has been installed, the building of IPSEC will automatically be enabled when the FreeBSD stack is used. It can be disabled using the configuration option CYGPKG_NET_IPSEC. There are no other configuration options for IPSEC as a whole.

In order to use IPSEC, connections must be configured. This can be performed using setsockopt() calls. A more convenient way is the use the libipsec library from the KAME distribution. eCos contains a snapshot of this library, which is documented else where. The aim is to also port the racoon daemon to eCos in the near future.

It should be noted that the FreeBSD stack in eCos is quite old. IPSEC and IPv6 have continued to develop. It is quite possible there could be interoperability problems when using the IPSEC implementation in eCos with more modern implementations.

It should also be noted that IPSEC, libipsec etc are currently work in progress items.
Chapter 51. libipsec Reference

ipsec_set_policy

IPSEC_SET_POLICY(3) System Library Functions Manual IPSEC_SET_POLICY(3)

NAME
    ipsec_set_policy, ipsec_get_policylen, ipsec_dump_policy - manipulate
    IPsec policy specification structure from readable string

LIBRARY
    IPsec Policy Control Library (libipsec, -lipsec)

SYNOPSIS
    #include <netinet6/ipsec.h>

    char *
    ipsec_set_policy(char *policy, int len);

    int
    ipsec_get_policylen(char *buf);

    char *
    ipsec_dump_policy(char *buf, char *delim);

DESCRIPTION
    ipsec_set_policy() generates IPsec policy specification structure, namely
    struct sadb_x_policy and/or struct sadb_x_ipsecrequest from human-readable
    policy specification. policy specification must be given as C
    string policy and length len of policy. ipsec_set_policy() will return
    the buffer of IPsec policy specification structure. The buffer is dynam-
    ically allocated, and must be freed by the caller by calling free(3).

    You may want the length of the generated buffer such when calling
    setsockopt(2). ipsec_get_policylen() will return the length.

    ipsec_dump_policy() converts IPsec policy structure into readable form.
    Therefore, ipsec_dump_policy() can be regarded as inverse conversion of
    ipsec_set_policy(). buf points to a IPsec policy structure, struct
    sadb_x_policy. delim is a delimiter string, which is usually a blank
    character. If you set delim to NULL, single whitespace is assumed.
    ipsec_dump_policy() returns pointer to dynamically allocated string. It
    is caller’s responsibility to reclaim the region, by using free(3).

    policy is formatted as either of the following:

    direction discard
        direction must be in or out. direction specifies which direc-
        tion the policy needs to be applied. With discard policy, pack-
        ets will be dropped if they match the policy.

    direction entrust
        entrust means to consult to SPD defined by setkey(8).

    direction bypass
**Chapter 51. libipsec Reference**

bypass means to be bypassed the IPsec processing. (packet will be transmitted in clear). This is for privileged socket.

direction ipsec request ...

ipsec means that the matching packets are subject to IPsec processing. ipsec can be followed by one or more request string, which is formatted as below:

```
protocol / mode / src - dst [/level]
```

- protocol is either ah, esp or ipcomp.
- mode is either transport or tunnel.
- src and dst specifies IPsec endpoint. src always means "sending node" and dst always means "receiving node". Therefore, when direction is in, dst is this node and src is the other node (peer). If mode is transport, Both src and dst can be omitted.
- level must be set to one of the following: default, use, require or unique. default means that the kernel should consult the system default policy defined by sysctl(8), such as net.inet.ipsec.esp_trans_deflev. See ipsec(4) regarding the system default. use means that a relevant SA can be used when available, since the kernel may perform IPsec operation against packets when possible. In this case, packets can be transmitted in clear (when SA is not available), or encrypted (when SA is available). require means that a relevant SA is required, since the kernel must perform IPsec operation against packets. unique is the same as require, but adds the restriction that the SA for outbound traffic is used only for this policy. You may need the identifier in order to relate the policy and the SA when you define the SA by manual keying. You can put the decimal number as the identifier after unique like unique: number. number must be between 1 and 32767. If the request string is kept unambiguous, level and slash prior to level can be omitted. However, it is encouraged to specify them explicitly to avoid unintended behaviors. If level is omitted, it will be interpreted as default.

Note that there is a bit difference of specification from setkey(8). In specification by setkey(8), both entrust and bypass are not used. Refer to setkey(8) for detail.

Here are several examples (long lines are wrapped for readability):

```
in discard
  out ipsec esp/transport/require
  in ipsec ah/transport/require
  out ipsec esp/tunnel/10.1.1.2-10.1.1.1/use
  in ipsec ipcomp/transport/use
      esp/transport/use
```
returns a pointer to dynamically allocated region on success, and NULL on errors.

SEE ALSO
ipsec_strerror(3), ipsec(4), setkey(8)

HISTORY
The functions first appeared in WIDE/KAME IPv6 protocol stack kit.

KAME May 5, 1998 KAME

---

ipsec_strerror

IPSEC_STRERROR(3) System Library Functions Manual IPSEC_STRERROR(3)

NAME
ipsec_strerror - error message for IPsec policy manipulation library

SYNOPSIS
#include <netinet6/ipsec.h>

const char * ipsec_strerror();

DESCRIPTION
netinet6/ipsec.h declares

extern int ipsec_errcode;

which is used to pass an error code from IPsec policy manipulation library to an user program. ipsec_strerror() can be used to obtain the error message string for the error code.

The array pointed to is not to be modified by the program. Since ipsec_strerror() uses strerror(3) as underlying function, calling strerror(3) after ipsec_strerror() would make the return value from ipsec_strerror() invalid, or overwritten.

RETURN VALUES
ipsec_strerror() always return a pointer to C string. The C string must not be overwritten by user programs.

SEE ALSO
ipsec_set_policy(3)

HISTORY
ipsec_strerror() first appeared in WIDE/KAME IPv6 protocol stack kit.

BUGS
ipsec_strerror() will return its result which may be overwritten by subsequent calls.

ipsec_errcode is not thread safe.

KAME May 6, 1998 KAME
XX. eCos PPP User Guide

This package provides support for PPP (Point-to-Point Protocol) in the eCos FreeBSD TCP/IP networking stack.
Chapter 52. Features

The eCos PPP implementation provides the following features:

- PPP line protocol including VJ compression.
- LCP, IPCP and CCP control protocols.
- PAP and CHAP authentication.
- CHAT subset connection scripting.
- Modem control line support.
Chapter 53. Using PPP

Before going into detail, let’s look at a simple example of how the eCos PPP package is used. Consider the following example:

```c
static void ppp_up(void)
{
    cyg_ppp_options_t options;
    cyg_ppp_handle_t ppp_handle;

    // Bring up the TCP/IP network
    init_all_network_interfaces();

    // Initialize the options
    cyg_ppp_options_init( &options );

    // Start up PPP
    ppp_handle = cyg_ppp_up( "/dev/ser0", &options );

    // Wait for it to get running
    if( cyg_ppp_wait_up( ppp_handle ) == 0 )
    {
        // Make use of PPP
        use_ppp();

        // Bring PPP link down
        cyg_ppp_down( ppp_handle );

        // Wait for connection to go down.
        cyg_ppp_wait_down( ppp_handle );
    }
}
```

This is a simple example of how to bring up a simple PPP connection to another computer over a directly connected serial line. The other end is assumed to already be running PPP on the line and waiting for a connection.

The first thing this code does is to call `init_all_network_interfaces()` to bring up the TCP/IP stack and initialize any other network interfaces. It then calls `cyg_ppp_options_init()` to initialize the PPP options structure to the defaults. As it happens, the default options are exactly what we want for this example, so we don’t need to make any further changes. We go straight on to bring the PPP interface up by calling `cyg_ppp_up()`. The arguments to this function give the name of the serial device to use, in this case "/dev/ser0", and a pointer to the options.

When `cyg_ppp_up()` returns, it passes back a handle to the PPP connection which is to be used in other calls. The PPP link will not necessarily have been fully initialized at this time. There is a certain amount of negotiation that goes on between the ends of a PPP link before it is ready to pass packets. An application can wait until the link is ready by calling `cyg_ppp_wait_up()`, which returns zero if the link is up and running, or -1 if it has gone down or failed to come up.

After a successful return from `cyg_ppp_wait_up()`, the application may make use of the PPP connection. This is represented here by the call to `use_ppp()` but it may, of course, be accessed by any thread. While the connection is up the application may use the standard socket calls to make or accept network connections and transfer data in the normal way.
Chapter 53. Using PPP

Once the application has finished with the PPP link, it can bring it down by calling \texttt{cyg_ppp_down()}. As with bringing the connection up, this call is asynchronous, it simply informs the PPP subsystem to start bringing the link down. The application can wait for the link to go down fully by calling \texttt{cyg_ppp_wait_down()}.

That example showed how to use PPP to connect to a local peer. PPP is more often used to connect via a modem to a remote server, such as an ISP. The following example shows how this works:

\begin{verbatim}
static char *isp_script[] =
{
  "ABORT" , "BUSY" ,
  "ABORT" , "NO CARRIER" ,
  "ABORT" , "ERROR" ,
  "ATZ" ,
  "OK" , "AT S7=45 S0=0 L1 V1 X4 &C1 E1 Q0" ,
  "OK" , "ATD" CYGPKG_PPP_DEFAULT.Dialup_number ,
  "Login:" , CYGPKG_PPP_AUTH_DEFAULT_USER ,
  "Password:" , CYGPKG_PPP_AUTH_DEFAULT_PASSWD ,
  "Protocol:" , "ppp" ,
  "HELLO" , "\c"
};

static void ppp_up(void)
{
  cyg_ppp_options_t options;
  cyg_ppp_handle_t ppp_handle;

  // Bring up the TCP/IP network
  init_all_network_interfaces();

  // Initialize the options
  cyg_ppp_options_init( &options );

  options.script = isp_script;
  options.modem = 1;

  // Start up PPP
  ppp_handle = cyg_ppp_up( "/dev/ser0", &options );

  // Wait for it to get running
  if( cyg_ppp_wait_up( ppp_handle ) == 0 )
  {
    // Make use of PPP
    use_ppp();

    // Bring PPP link down
    cyg_ppp_down( ppp_handle );

    // Wait for connection to go down.
    cyg_ppp_wait_down( ppp_handle );
  }
}
\end{verbatim}

The majority of this code is exactly the same as the previous example. The main difference is in the setting of a couple of options before calling \texttt{cyg_ppp_up()}. The \textit{script} option is set to point to a CHAT script to manage the setup of the connection. The \textit{modem} option is set to cause the PPP system to make use of the modem control lines.
During the PPP bring-up a call will be made to `cyg_ppp_chat()` to run the CHAT script (see Chapter 56). In the example this script sets up various modem options and then dials a number supplied as part of the PPP package configuration (see Chapter 55). When the connection has been established, the script logs on to the server, using a name and password also supplied by the configuration, and then starts PPP on the remote end. If this script succeeds the PPP connection will be brought up and will then function as expected.

The `modem` option causes the PPP system to make use of the modem control lines. In particular it waits for Carrier Detect to be asserted, and will bring the link down if it is lost. See `cyg_ppp_options_init()` for more details.
Chapter 53. Using PPP
Chapter 54. PPP Interface

cyg_ppp_options_init()

Name

cyg_ppp_options_init — Initialize PPP link options

Synopsis

#include <cyg/ppp/ppp.h>

cyg_int32 cyg_ppp_options_init(cyg_ppp_options_t *options);

Description

This function initializes the PPP options, pointed to by the options parameter, to the default state. Once the defaults have been initialized, application code may adjust them by assigning new values to the the fields of the cyg_ppp_options_t structure.

This function returns zero if the options were initialized successfully. It returns -1 if the options argument is NULL, or the options could not be initialized.

The option fields, their functions and default values are as follows:

debug

If set to 1 this enables the reporting of debug messages from the PPP system. These will be generated using diag_printf() and will appear on the standard debug channel. Note that diag_printf() disabling interrupts during output; this may cause the PPP link device to overrun and miss characters. It is quite possible for this option to cause errors and even make the PPP link fail completely. Consequently, this option should be used with care.

Default value: 0

kdebugflag

This five bit field enables low level debugging messages from the PPP device layer in the TCP/IP stack. As with the debug option, this may result in missed characters and cause errors. The bits of the field have the following meanings:

<table>
<thead>
<tr>
<th>Bit</th>
<th>BSD Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>SC_DEBUG</td>
<td>Enable debug messages</td>
</tr>
</tbody>
</table>
cyg_ppp_options_init()

<table>
<thead>
<tr>
<th>Bit</th>
<th>BSD Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02</td>
<td>SC_LOG_INPKT</td>
<td>Log contents of good packets received</td>
</tr>
<tr>
<td>0x04</td>
<td>SC_LOG_OUTPKT</td>
<td>Log contents of packets sent</td>
</tr>
<tr>
<td>0x08</td>
<td>SC_LOG_RAWIN</td>
<td>Log all characters received</td>
</tr>
<tr>
<td>0x10</td>
<td>SC_LOG_FLUSH</td>
<td>Log all characters flushed</td>
</tr>
</tbody>
</table>

Default value: 0

default_route

If set to 1 this option causes the PPP subsystem to install a default route in the TCP/IP stack’s routing tables using the peer as the gateway. This entry will be removed when the PPP link is broken. If there is already an existing working network connection, such as an ethernet device, then there may already be a default route established. If this is the case, then this option will have no effect.

Default value: 1

modem

If this option is set to 1, then the modem lines will be used during the connection. Specifically, the PPP subsystem will wait until the carrier detect signal is asserted before bringing up the PPP link, and will take the PPP link down if this signal is de-asserted.

Default value: 0

flowctl

This option is used to specify the mechanism used to control data flow across the serial line. It can take one of the following values:

CYG_PPP_FLOWCTL_DEFAULT

The flow control mechanism is not changed and is left at whatever value was set before bringing PPP up. This allows a non-standard flow control mechanism to be used, or for it to be chosen and set by some other means.

CYG_PPP_FLOWCTL_NONE

Flow control is turned off. It is not recommended that this option be used unless the baud rate is set low or the two communicating machines are particularly fast.

CYG_PPP_FLOWCTL_HARDWARE

Use hardware flow control via the RTS/CTS lines. This is the most effective flow control mechanism and should always be used if available. Availability of this mechanism depends on whether the serial device hardware has the ability to control these lines, whether they have been connected to the socket pins and whether the device driver has the necessary support.

CYG_PPP_FLOWCTL_SOFTWARE

Use software flow control by embedding XON/XOFF characters in the data stream. This is somewhat less effective that hardware flow control since it is subject to the propagation time of the serial cable and the latency of the communicating devices. Since it does not rely on any hardware support, this flow control mechanism is always available.

Default value: CYG_PPP_FLOWCTL_HARDWARE
refuse_pap

If this option is set to 1, then the PPP subsystem will not agree to authenticate itself to the peer with PAP. When dialling in to a remote server it is normal to authenticate the client. There are three ways this can be done, using a straightforward login mechanism via the CHAT script, with the Password Authentication Protocol (PAP), or with the Challenge Handshake Authentication Protocol (CHAP). For PAP to work the user and passwd options must be set to the expected values. If they are not, then this option should be set to force CHAP authentication.

Default value: 0

refuse_chap

If this option is set to 1, then the PPP subsystem will not agree to authenticate itself to the peer with CHAP. CHAP authentication will only work if the passwd option has been set to the required CHAP secret for the destination server. Otherwise this option should be disabled.

If both refuse_pap and refuse_chap are set, then either no authentication will be carried out, or it is the responsibility of the chat script to do it. If the peer does not require any authentication, then the setting of these options is irrelevant.

Default value: 0

baud

This option is set to the baud rate at which the serial connection should be run. The default value is the rate at which modems conventionally operate. This field is an instance of the cyg_serial_baud_rate_t enum defined in the serialio.h header and may only take one of the baud rate constants defined in there.

Default value: CYGNUM_SERIAL_BAUD_115200

idle_time_limit

This is the number of seconds that the PPP connection may be idle before it is shut down automatically.

Default value: 60

maxconnect

This causes the connection to terminate when it has been up for this number of seconds. The default value of zero means that the connection will stay up indefinitely, until either end explicitly brings it down, or the link is lost.

Default value: 0

our_address

This is the IP address, in network byte order, to be attached to the local end of the PPP connection. The default value of INADDR_ANY causes the local address to be obtained from the peer.

Default value: INADDR_ANY

his_address

This is the IP address, in network byte order, to be attached to the remote end of the PPP connection. The default value of INADDR_ANY causes the remote address to be obtained from the peer.

Default value: INADDR_ANY

script

This is a pointer to a CHAT script suitable for passing to cyg_ppp_chat(). See Chapter 56 for details of the format and contents of this script.
cyg_ppp_options_init()

Default value: NULL

user

This array contains the user name to be used for PAP authentication. This field is not used for CHAP authentication. By default the value of this option is set from the CYGPKG_PPP_AUTH_DEFAULT_USER configuration option.

Default value: CYGPKG_PPP_AUTH_DEFAULT_USER

passwd

This array contains the password to be used for PAP authentication, or the secret to be used during CHAP authentication. By default the value of this option is set from the CYGPKG_PPP_AUTH_DEFAULT_PASSWD configuration option.

Default value: CYGPKG_PPP_AUTH_DEFAULT_PASSWD
cyg_ppp_up()

Name
cyg_ppp_up — Bring PPP connection up

Synopsis

#include <cyg/ppp/ppp.h>

cyg_ppp_handle_t cyg_ppp_up(char *devnam, const cyg_ppp_options_t *options);

Description

This function starts up a PPP connection. The devnam argument is the name of the device to be used for the connection, typically "/dev/ser0" or "/dev/ser1". The options argument should point to an initialized cyg_ppp_options_t object.

The return value will either be zero, indicating a failure, or a cyg_ppp_handle_t object that may be used as an argument to other PPP functions.

Note: Although the PPP API is designed to permit several simultaneous connections to co-exist, at present only one PPP connection is actually implemented. Any attempt to create a second connection while there is already one open will fail.
cyg_ppp_up()
cyg_ppp_down()

Name

cyg_ppp_down — Bring PPP connection down

Synopsis

#include <cyg/ppp/ppp.h>

cyg_int32 cyg_ppp_down(cyg_ppp_handle_t handle);

Description

This function brings the PPP connection down. The handle argument is the result of a successful call to
cyg_ppp_up(). This function only signals to the PPP subsystem that the link should be brought down. The
link will be terminated asynchronously. If the application needs to wait for the link to terminate, then it should
call cyg_ppp_wait_down() after calling cyg_ppp_down().

The function returns zero if it was able to start the termination of the PPP connection successfully. It will return
-1 if the connection is not running, or if it could not otherwise start the termination.
cyg_ppp_down()
**cyg_ppp_wait_up()**

**Name**

*cyg_ppp_wait_up* — Wait for PPP connection to come up

**Synopsis**

```c
#include <cyg/ppp/ppp.h>

cyg_int32 cyg_ppp_wait_up(cyg_ppp_handle_t handle);
```

**Description**

This function waits until the PPP connection is running and then returns. This is needed because the actual bring up of the connection happens mostly after the call to *cyg_ppp_up()* returns, and may take some time to complete, especially if dialling a remote server.

The result of this call will be zero when the connection is running, or -1 if the connection failed to start for some reason. If the connection is already running when this call is made it will return immediately with a zero result. If the connection is not in the process of coming up, or has failed, or has terminated, then a result of -1 will be returned immediately. Thus this function may also be used to test that the connection is still running at any point.
cyg_ppp_wait_up()
cyg_ppp_wait_down()

Name

`cyg_ppp_wait_down` — Wait for PPP connection to terminate

Synopsis

```c
#include <cyg/ppp/ppp.h>

void cyg_ppp_wait_down(cyg_ppp_handle_t handle);
```

Description

This function waits for the PPP connection to terminate. The link may be terminated with a call to `cyg_ppp_down()`, by the remote end, or by the telephone line being dropped or lost.

This function has no return value. If the PPP connection is not running, or has terminated, it will return. Applications should use `cyg_ppp_wait_up()` to test the link state.
cyg_ppp_wait_down()
Cyg_Ppp_Chat()

Name

Cyg_Ppp_Chat — Execute chat script

Synopsis

#include <cyg/ppp/ppp.h>

Cyg_Int32 Cyg_Ppp_Chat(const char *devname, const char *script[]);

Description

This function implements a subset of the automated conversational scripting as defined by the chat program. The first argument is the name of the serial device to be used, typically "/dev/ser0" or "/dev/ser1". The script argument is a pointer to a zero terminated array of strings that comprise the chat script. See Chapter 53 for an example script, and Chapter 56 for full detail of the script used.

The return value of this function will be zero if the chat script fails for any reason, such as an ABORT or a timeout. If the end of the script is reached, then the return value will be non-zero.

Under normal use this function is called from the PPP subsystem if the cyg_ppp_options_t script field is set to a non-NULL value. This function should only be used directly if the application needs to undertake special processing between running the chat script, and bringing up the PPP connections.
cyg_ppp_chat()
Chapter 55. Installing and Configuring PPP

Including PPP in a Configuration

PPP is contained entirely within a single eCos package. So to include PPP in a configuration all you need to do is add that package.

In the GUI configuration tool use the Build->Packages menu item, find the "PPP Support" package in the left-hand pane and use the Add button to add it to the list of packages in use in the right-hand pane.

In the command-line tool ecosconfig, you can use the following command during the configuration phase to add the PPP package:

$ ecosconfig add ppp

In addition to the PPP package you will also need to have the "Network" package and the "Serial Device Drivers" package in the configuration. The dependencies and requirements of the networking package are such that it is strongly recommended that you start with the net template.

See the eCos User Guide for full details on how to configure and build eCos.

Configuring PPP

The PPP package contains a number of configuration options that may be changed to affect its behaviour.

CYGNUM_PPP_PPPD_THREAD_PRIORITY

The PPP system contains two threads. One is used for receiving data from the link and processing control packets. The other is used to transmit data asynchronously to the link when it cannot be completed synchronously. The receive thread runs at the priority given here, and the transmit thread runs at the next lower priority. The exact priority needed here depends on the importance of the PPP subsystem relative to the rest of the system. The default is to put it in the middle of the priority range to provide reasonable response without impacting genuine high priority threads.

Default value: CYGNUM_KERNEL_SCHED_PRIORITIES/2

CYGPKG_PPP_DEBUG_WARN_ONLY

The runtime debug option enables logging of high level debug messages. Too many of these can interfere with the PPP device and may result in missed messages. This is because these messages are emitted via the diag_printf() mechanism, which disables interrupts while it prints. By default, therefore, we only report errors and warnings, and not all events. Setting this option to zero will enable the logging of all events.

Default value: 1

CYGPKG_PPP_AUTH_DEFAULT_USER

This option gives the default value for the user name used to initialize the user field in the PPP options.

Default value: "eCos"

CYGPKG_PPP_AUTH_DEFAULT_PASSWD

This option gives the default value for the password used to initialize the passwd field in the PPP options.

Default value: "secret"
Chapter 55. Installing and Configuring PPP

CYGPKG_PPP_DEFAULT_DIALUP_NUMBER

This option provides a default dialup number for use in chat scripts. This value is not used anywhere in the PPP package, but is provided to complete the information needed, alongside the user name and password, for accessing a typical dialup server.

Default value: "5551234"

CYGPKG_PPP_PAP

This component enables the inclusion of PAP authentication support.

Default value: 1

CYGPKG_PPP_CHAP

This component enables the inclusion of CHAT authentication support.

Default value: 1

CYGPKG_PPP_COMPRESSION

This component provides control over PPP compression features. WARNING: at present there are problems with this option, and in any case the compression code needs to allocate large amounts of memory. Hence this option is currently disabled and should remain so.

Default value: 0

PPP_BSDCOMP

This option enables inclusion of BSD compression into the PPP protocol.

Default value: 0

PPP_DEFLATE

This option enables inclusion of ZLIB compression into the PPP protocol.

Default value: 0

CYGPKG_PPP_CHAT

This component enables the inclusion of a simple scripting system to bring up PPP connections. It implements a subset of the chat scripting language.

Default value: 1

CYGNUM_PPP_CHAT_ABORTS_MAX

This option defines the maximum number of ABORT strings that the CHAT system will store.

Default value: 10

CYGNUM_PPP_CHAT_ABORTS_SIZE

This option defines the maximum size of each ABORT strings that the chat system will store.

Default value: 20

CYGNUM_PPP_CHAT_STRING_LENGTH

This option defines the maximum size of any expect or reply strings that the chat system will be given.

Default value: 256
CYGPKG_PPP_TEST_DEVICE

This option defines the serial device to be used for PPP test programs.

Default value: "/dev/ser0"

CYGPKG_PPP_TESTS_AUTOMATE

This option enables automated testing features in certain test programs. These programs will interact with a test server at the remote end of the serial link to run a variety of tests in different conditions. Without this option most tests default to running a single test instance and are suitable for being run by hand for debugging purposes.

Default value: 0

CYGDAT_PPP_TEST_BAUD_RATES

This option supplies a list of baud rates at which certain tests will run if the CYGPKG_PPP_TESTS_AUTOMATE option is set.

Default value: "CYGNUM_SERIAL_BAUD_19200,CYGNUM_SERIAL_BAUD_38400,CYGNUM_SERIAL_BAUD_57600,CYGNUM"
Chapter 56. CHAT Scripts

The automated conversational scripting supported by the eCos PPP package is a subset of the scripting language provided by the chat command found on most UNIX and Linux systems.

Unlike the chat command, the eCos cyg_ppp_chat() function takes as a parameter a zero-terminated array of pointers to strings. In most programs this will be defined by means of an initializer for a static array, although there is nothing to stop the application constructing it at runtime. A simple script would be defined like this:

```c
static char *chat_script[] = {
    "ABORT", "BUSY",
    "ABORT", "NO CARRIER",
    "", "ATD5551234",
    "login:" , "ppp",
    "ssword:" , "hithere",
    0
};
```

The following sections have been abstracted from the public domain documentation for the chat command.

Chat Script

A script consists of one or more "expect-send" pairs of strings, separated by spaces, with an optional "subexpect- subsend" string pair, separated by a dash as in the following example:

```c
"login:" , "ppp",
"ssword:" , "hello2u2",
0
```

This script fragment indicates that the cyg_ppp_chat() function should expect the string "login:". If it fails to receive a login prompt within the time interval allotted, it is to send a carriage return to the remote and then expect the string "login:" again. If the first "login:" is received then the carriage return is not generated.

Once it received the login prompt the cyg_ppp_chat() function will send the string "ppp" and then expect the prompt "ssword:". When it receives the prompt for the password, it will send the password "hello2u2".

A carriage return is normally sent following the reply string. It is not expected in the "expect" string unless it is specifically requested by using the "\r" character sequence.

The expect sequence should contain only what is needed to identify the string. It should not contain variable information. It is generally not acceptable to look for time strings, network identification strings, or other variable pieces of data as an expect string.

To help correct for characters which may be corrupted during the initial sequence, look for the string "login:" rather than "login:". It is possible that the leading "I" character may be received in error and you may never find the string even though it was sent by the system. For this reason, scripts look for "login:" rather than "login:" and "password:" rather than "password:".

A very simple script might look like this:

```c
"login:" , "ppp",
"ssword:" , "hello2u2",
0
```
In other words, expect "...login:”, send "ppp”, expect "...ssword:”, send "hello2u2”.

In actual practice, simple scripts are rare. At the very least, you should include sub-expect sequences should the original string not be received. For example, consider the following script:

```plaintext
"login:--login:" , "ppp" ,
"ssword:" , "hello2u2",
0
```

This would be a better script than the simple one used earlier. This would look for the same "login:" prompt, however, if one was not received, a single return sequence is sent and then it will look for "login:" again. Should line noise obscure the first login prompt then sending the empty line will usually generate a login prompt again.

### ABORT Strings

Many modems will report the status of the call as a string. These strings may be CONNECTED or NO CARRIER or BUSY. It is often desirable to terminate the script should the modem fail to connect to the remote. The difficulty is that a script would not know exactly which modem string it may receive. On one attempt, it may receive BUSY while the next time it may receive NO CARRIER.

These "abort" strings may be specified in the script using the ABORT sequence. It is written in the script as in the following example:

```plaintext
"ABORT" , "BUSY" ,
"ABORT" , "NO CARRIER" ,
"OK" , "ATDT5551212",
"CONNECT" , ...
```

This sequence will expect nothing; and then send the string ATZ. The expected response to this is the string OK. When it receives OK, it sends the string ATDT5551212 to dial the telephone. The expected string is CONNECT. If the string CONNECT is received the remainder of the script is executed. However, should the modem find a busy telephone, it will send the string BUSY. This will cause the string to match the abort character sequence. The script will then fail because it found a match to the abort string. If it received the string NO CARRIER, it will abort for the same reason. Either string may be received. Either string will terminate the chat script.

### TIMEOUT

The initial timeout value is 45 seconds. To change the timeout value for the next expect string, the following example may be used:

```plaintext
"" , "ATZ",
"OK" , "ATDT5551212",
"CONNECT" , "\c",
"TIMEOUT" , "10",
"login:--login:" , "ppp",
"TIMEOUT" , "5",
"ssword:" , "hello2u2",
0
```

This will change the timeout to 10 seconds when it expects the login: prompt. The timeout is then changed to 5 seconds when it looks for the password prompt.
The timeout, once changed, remains in effect until it is changed again.

Sending EOT

The special reply string of EOT indicates that the chat program should send an EOT character to the remote. This is normally the End-of-file character sequence. A return character is not sent following the EOT. The EOT sequence may be embedded into the send string using the sequence "\x04" (i.e. a Control-D character).

Escape Sequences

Most standard chat escape sequences can be replaced with standard C string escapes such as \r', \n', \t' etc. Additional escape sequences may be embedded in the expect or reply strings by introducing them with two backslashes.

\c

Suppresses the newline at the end of the reply string. This is the only method to send a string without a trailing return character. It must be at the end of the send string. For example, the sequence "hello\c" will simply send the characters h, e, l, l, o. (not valid in expect strings.)
Chapter 57. PPP Enabled Device Drivers

For PPP to function fully over a serial device, its driver must implement certain features. At present not all eCos serial drivers implement these features. A driver indicates that it supports a certain feature by including an "implements" line in its CDL for the following interfaces:

**CYGINT_IO_SERIAL_FLOW_CONTROL_HW**

This interface indicates that the driver implements hardware flow control using the RTS and CTS lines. When data is being transferred over high speed data lines, it is essential that flow control be used to prevent buffer overrun.

The PPP subsystem functions best with hardware flow control. If this is not available, then it can be configured to use software flow control. Since software flow control is implemented by the device independent part of the serial device infrastructure, it is available for all serial devices. However, this will have an effect on the performance and reliability of the PPP link.

**CYGINT_IO_SERIAL_LINE_STATUS_HW**

This interface indicates that the driver implements a callback interface for indicating the status of various RS232 control lines. Of particular interest here is the ability to detect changes in the Carrier Detect (CD) line. Not all drivers that implement this interface can indicate CD status.

This functionality is only needed if it is important that the link be dropped immediately a telephone connection fails. Without it, a connection will only be dropped after it times out. This may be acceptable in many situations.

At the time of writing, the serial device drivers for the following platforms implement some or all of the required functionality:

- All drivers that use the generic 16x5x driver implement all functions:
  - ARM CerfPDA
  - ARM IQ80321
  - ARM PID
  - ARM IOP310
  - i386 PC
  - MIPS Atlas
  - MIPS Ref4955
  - SH3 SE77x9

- The following drivers implement flow control but either do not support line status callbacks, or do not report CD changes:
  - SH4 SCIF
  - A&M AdderI
  - A&M AdderII
• All other drivers can support software flow control only.
Chapter 58. Testing

Test Programs

There are a number of test programs supplied with the PPP subsystem. By default all of these tests use the device configured by CYGPKG_PPP_TEST_DEVICE as the PPP link device.

PPP_up

This test just brings up the PPP link on CYGPKG_PPP_TEST_DEVICE and waits until the remote end brings it back down. No modem lines are used and the program expects a PPP connection to be waiting on the other end of the line. Typically the remote end will test the link using ping or access the HTTP system monitor if it is present.

If CYGPKG_PPP_TESTS_AUTOMATE is set, then this test attempts to bring PPP up at each of the baud rates specified in CYGDAT_PPP_TEST_BAUD_RATES. If it is not set then it will just bring the connection up at 115200 baud.

PPP_updown

This test brings the PPP link up on CYGPKG_PPP_TESTDEVICE and attempts to ping the remote end of the link. Once the pings have finished, the link is then brought down.

If CYGPKG_PPP_TESTS_AUTOMATE is set, then this test attempts to bring PPP up at each of the baud rates specified in CYGDAT_PPP_TEST_BAUD_RATES. If it is not set then it will just bring the connection up at 115200 baud.

chat

This test does not bring the PPP link up but simply executes a chat script. It expects a server at the remote end of the link to supply the correct responses.

This program expects the test_server.sh script to be running on the remote end and attempts several different tests, expecting a variety of different responses for each.

PPP_auth

This test attempts to bring up the PPP link under a variety of different authentication conditions. This includes checking that both PAP and CHAP authentication work, and that the connection is rejected when the incorrect authentication protocol or secrets are used.

This test expects the test_server.sh script to be running on the remote end. For this test to work the /etc/ppp/pap-secrets file on the remote end should contain the following two lines:

```
eCos       *   secret   *
eCosPAP    *   secretPAP *
```

The /etc/ppp/chap-secrets file should contain:

```
eCos       *   secret   *
eCosCHAP   *   secretCHAP *
```
This test expects the serial test device to be connected to a Hayes compatible modem. The test dials the telephone number given in CYGPKG_PPP_DEFAULT_DIALUP_NUMBER and attempts to log on to an ISP using the user name and password supplied in CYGPKG_PPP_AUTH_DEFAULT_USER and CYGPKG_PPP_AUTH_DEFAULT_PASSWD. Once the PPP connection has been made, the program then attempts to ping a number of well known addresses.

Since this test is designed to interact with an ISP, it does not run within the automated testing system.

tcp_echo

This is a version of the standard network tcp_echo test that brings up the PPP connection before waiting for the tcp_sink and tcp_source programs to connect. It is expected that at least one of these programs will connect via the PPP link. However, if another network interface is present, such as an ethernet device, then one may connect via that interface.

While this test is supported by the test_server.sh script, it runs for such a long time that it should not normally be used during automated testing.

nc_test_slave

This is a version of the standard network nc_test_slave test that brings up the PPP connection before waiting for the nc_test_master program to connect. It is expected that the master will connect via the PPP link.

While this test is supported by the test_server.sh script, it runs for such a long time that it should not normally be used during automated testing.

Test Script

The PPP package additionally contains a shell script (test_server.sh) that may be used to operate the remote end of a PPP test link.

The script may be invoked with the following arguments:

```
--dev=<devname>
This mandatory option gives the name of the device to be used for the PPP link. Typically "/dev/ttyS0" or "/dev/ttyS1".

--myip=<ipaddress>
This mandatory option gives the IP address to be attached to this end of the PPP link.

--hisip=<ipaddress>
This mandatory option gives the IP address to be attached to the remote (test target) end of the PPP link.

--baud=<baud_rate>
This option gives the baud rate at which the PPP link is to be run. If absent then the link will run at the value set for --redboot-baud.

--redboot
If this option is present then the script will look for a "RedBoot>" prompt between test runs. This is necessary if the serial device being used for testing is also used by RedBoot.
```
Chapter 58. Testing

--redboot-baud=<baud_rate>

This option gives the baud rate at which the search for the RedBoot prompt will be made. If absent then the link will run at 38400 baud.

--debug

If this option is present, then the script will print out some additional debug messages while it runs.

This script operates as follows: If the --redboot option is set it sets the device baud rate to the RedBoot baud rate and waits until a "RedBoot>" prompt is encountered. It then sets the baud rate to the value given by the --baud option and reads lines from the device until a recognizable test announce string is read. It then executes an appropriate set of commands to satisfy the test. This usually means bringing up the PPP link by running pppd and maybe executing various commands. It then either terminates the link itself, or waits for the target to terminate it. It then goes back to looking for another test announce string. If a string of the form "BAUD:XXX" is received then the baud rate is changed depending on the XXX value. If a "FINISH" string is received it returns to waiting for a "RedBoot>" prompt. The script repeats this process until it is terminated with a signal.
Chapter 58. Testing
XXI. Ethernet Device Drivers
Chapter 59. Generic Ethernet Device Driver

Generic Ethernet API

This section provides a simple description of how to write a low-level, hardware dependent ethernet driver.

There is a high-level driver (which is only code — with no state of its own) that is part of the stack. There will be one or more low-level drivers tied to the actual network hardware. Each of these drivers contains one or more driver instances. The intent is that the low-level drivers know nothing of the details of the stack that will be using them. Thus, the same driver can be used by the eCos supported TCP/IP stack, RedBoot, or any other, with no changes.

A driver instance is contained within a struct eth_drv_sc:

```c
struct eth_hwr_funs {
    // Initialize hardware (including startup)
    void (*start)(struct eth_drv_sc *sc,
                  unsigned char *enaddr,
                  int flags);
    // Shut down hardware
    void (*stop)(struct eth_drv_sc *sc);
    // Device control (ioctl pass-thru)
    int (*control)(struct eth_drv_sc *sc,
                   unsigned long key,
                   void *data,
                   int data_length);
    // Query - can a packet be sent?
    int (*can_send)(struct eth_drv_sc *sc);
    // Send a packet of data
    void (*send)(struct eth_drv_sc *sc,
                 struct eth_drv_sg *sg_list,
                 int sg_len,
                 int total_len,
                 unsigned long key);
    // Receive [unload] a packet of data
    void (*recv)(struct eth_drv_sc *sc,
                 struct eth_drv_sg *sg_list,
                 int sg_len);
    // Deliver data to/from device from/to stack memory space
    // (moves lots of memcpy())s out of DSRs into thread)
    void (*deliver)(struct eth_drv_sc *sc);
    // Poll for interrupts/device service
    void (*poll)(struct eth_drv_sc *sc);
    // Get interrupt information from hardware driver
    int (*int_vector)(struct eth_drv_sc *sc);
    // Logical driver interface
    struct eth_drv_funs *eth_drv, *eth_drv_old;
};

struct eth_drv_sc {
    struct eth_hwr_funs *funs;
    void *driver_private;
    const char *dev_name;
    int state;
    struct arpcom *ac_arpcom; /* ethernet common */
};
```
Note: If you have two instances of the same hardware, you only need one struct eth_hwr_funs shared between them.

There is another structure which is used to communicate with the rest of the stack:

```c
struct eth_drv_funs {
    // Logical driver - initialization
    void (*init)(struct eth_drv_sc *sc, unsigned char *enaddr);
    // Logical driver - incoming packet notifier
    void (*recv)(struct eth_drv_sc *sc, int total_len);
    // Logical driver - outgoing packet notifier
    void (*tx_done)(struct eth_drv_sc *sc, CYG_ADDRESS key, int status);
};
```

Your driver does not create an instance of this structure. It is provided for driver code to use in the eth_drv member of the function record. Its usage is described below in the Section called Upper Layer Functions.

One more function completes the API with which your driver communicates with the rest of the stack:

```c
extern void eth_drv_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data);
```

This function is designed so that it can be registered as the DSR for your interrupt handler. It will awaken the “Network Delivery Thread” to call your deliver routine. See the Section called Deliver function.

You create an instance of struct eth_drv_sc using the ETH_DRV_SC() macro which sets up the structure, including the prototypes for the functions, etc. By doing things this way, if the internal design of the ethernet drivers changes (e.g. we need to add a new low-level implementation function), existing drivers will no longer compile until updated. This is much better than to have all of the definitions in the low-level drivers themselves and have them be (quietly) broken if the interfaces change.

The “magic” which gets the drivers started (and indeed, linked) is similar to what is used for the I/O subsystem. This is done using the NETDEVTAB_ENTRY() macro, which defines an initialization function and the basic data structures for the low-level driver.

```c
typedef struct cyg_netdevtab_entry {
    const char *name;
    bool (*init)(struct cyg_netdevtab_entry *tab);
    void *device_instance;
    unsigned long status;
} cyg_netdevtab_entry_t;
```

The device_instance entry here would point to the struct eth_drv_sc entry previously defined. This allows the network driver setup to work with any class of driver, not just ethernet drivers. In the future, there will surely be serial PPP drivers, etc. These will use the NETDEVTAB_ENTRY() setup to create the basic driver, but they will most likely be built on top of other high-level device driver layers.

To instantiate itself, and connect it to the system, a hardware driver will have a template (boilerplate) which looks something like this:

```c
#include <cyg/infra/cyg_type.h>
#include <cyg/hal/hal_arch.h>
```
Chapter 59. Generic Ethernet Device Driver

#include <cyg/infra/diag.h>
#include <cyg/hal/drv_api.h>
#include <cyg/io/eth/netdev.h>
#include <cyg/io/eth/eth_drv.h>

ETH_DRV_SC(DRV_sc, 
  0, // No driver specific data needed 
  "eth0", // Name for this interface 
  HRDWR_start, 
  HRDWR_stop, 
  HRDWR_control, 
  HRDWR_can_send 
  HRDWR_send, 
  HRDWR_recv, 
  HRDWR_deliver, 
  HRDWR_poll, 
  HRDWR_int_vector
}

NETDEVTAB_ENTRY(DRV_netdev, 
  "DRV", 
  DRV_HRDWR_Init, 
  &DRV_sc);

This, along with the referenced functions, completely define the driver.

Note: If one needed the same low-level driver to handle multiple similar hardware interfaces, you would need multiple invocations of the ETH_DRV_SC() / NETDEVTAB_ENTRY() macros. You would add a pointer to some instance specific data, e.g. containing base addresses, interrupt numbers, etc, where the

0, // No driver specific data

is currently.

Review of the functions

Now a brief review of the functions. This discussion will use generic names for the functions — your driver should use hardware-specific names to maintain uniqueness against any other drivers.

Init function

static bool DRV_HDWR_Init(struct cyg_netdevtab_entry *tab)

This function is called as part of system initialization. Its primary function is to decide if the hardware (as indicated via tab->device_instance) is working and if the interface needs to be made available in the system. If this is the case, this function needs to finish with a call to the ethernet driver function:

struct eth_drv_sc *sc = (struct eth_drv_sc *)tab->device_instance;
....initialize code....
// Initialize upper level driver
(sc->funs->eth_drv->init)( sc, unsigned char *enaddr );
where `enaddr` is a pointer to the ethernet station address for this unit, to inform the stack of this device’s readiness and availability.

**Note:** The ethernet station address (ESA) is supposed to be a world-unique, 48 bit address for this particular ethernet interface. Typically it is provided by the board/hardware manufacturer in ROM.

In many packages it is possible for the ESA to be set from RedBoot, (perhaps from ‘fconfig’ data), hard-coded from CDL, or from an EPROM. A driver should choose a run-time specified ESA (e.g. from RedBoot) preferentially, otherwise (in order) it should use a CDL specified ESA if one has been set, otherwise an EPROM set ESA, or otherwise fail. See the `cl/cs8900a` ethernet driver for an example.

### Start function

```c
static void HRDWR_start(struct eth_drv_sc *sc, unsigned char *enaddr, int flags)
```

This function is called, perhaps much later than system initialization time, when the system (an application) is ready for the interface to become active. The purpose of this function is to set up the hardware interface to start accepting packets from the network and be able to send packets out. The receiver hardware should not be enabled prior to this call.

**Note:** This function will be called whenever the up/down state of the logical interface changes, e.g. when the IP address changes, or when promiscuous mode is selected by means of an `ioctl()` call in the application. This may occur more than once, so this function needs to be prepared for that case.

**Note:** In future, the `flags` field (currently unused) may be used to tell the function how to start up, e.g. whether interrupts will be used, alternate means of selecting promiscuous mode etc.

### Stop function

```c
static void HRDWR_stop(struct eth_drv_sc *sc)
```

This function is the inverse of “start.” It should shut down the hardware, disable the receiver, and keep it from interacting with the physical network.

### Control function

```c
static int HRDWR_control(
    struct eth_drv_sc *sc, unsigned long key,
    void *data, int len)
```
This function is used to perform low-level “control” operations on the interface. These operations would typically be initiated via `ioctl()` calls in the BSD stack, and would be anything that might require the hardware setup to change (i.e. cannot be performed totally by the platform-independent layers).

The `key` parameter selects the operation, and the `data` and `len` params point describe, as required, some data for the operation in question.

**Available Operations:**

**ETH_DRV_SET_MAC_ADDRESS**

This operation sets the ethernet station address (ESA or MAC) for the device. Normally this address is kept in non-volatile memory and is unique in the world. This function must at least set the interface to use the new address. It may also update the NVM as appropriate.

**ETH_DRV_GET_IF_STATS_UD**
**ETH_DRV_GET_IF_STATS**

These acquire a set of statistical counters from the interface, and write the information into the memory pointed to by `data`. The “UD” variant explicitly instructs the driver to acquire up-to-date values. This is a separate option because doing so may take some time, depending on the hardware.

The definition of the data structure is in `cyg/io/eth/eth_drv_stats.h`.

This call is typically made by SNMP, see Chapter 61.

**ETH_DRV_SET_MC_LIST**

This entry instructs the device to set up multicast packet filtering to receive only packets addressed to the multicast ESAs in the list pointed to by `data`.

The format of the data is a 32-bit count of the ESAs in the list, followed by packed bytes which are the ESAs themselves, thus:

```c
#define ETH_DRV_MAX_MC 8
struct eth_drv_mc_list {
    int len;
    unsigned char addrs[ETH_DRV_MAX_MC][ETHER_ADDR_LEN];
};
```

**ETH_DRV_SET_MC_ALL**

This entry instructs the device to receive all multicast packets, and delete any explicit filtering which had been set up.

This function should return zero if the specified operation was completed successfully. It should return non-zero if the operation could not be performed, for any reason.

**Can-send function**

```c
static int HRDWR_can_send(struct eth_drv_sc *sc)
```

This function is called to determine if it is possible to start the transmission of a packet on the interface. Some interfaces will allow multiple packets to be “queued” and this function allows for the highest possible utilization of that mode.

Return the number of packets which could be accepted at this time, zero implies that the interface is saturated/busy.
Send function

```c
struct eth_drv_sg {
    CYG_ADDRESS buf;
    CYG_ADDRWORD len;
};

static void
HRDWR_send(
    struct eth_drv_sc *sc,
    struct eth_drv_sg *sg_list, int sg_len,
    int total_len, unsigned long key)
```

This function is used to send a packet of data to the network. It is the responsibility of this function to somehow hand the data over to the hardware interface. This will most likely require copying, but just the address/length values could be used by smart hardware.

**Note:** All data in/out of the driver is specified via a “scatter-gather” list. This is just an array of address/length pairs which describe sections of data to move (in the order given by the array), as in the struct eth_drv_sg defined above and pointed to by `sg_list`.

Once the data has been successfully sent by the interface (or if an error occurs), the driver should call `(sc->funs->eth_drv->tx_done)()` (see the Section called Callback Tx-Done function) using the specified `key`. Only then will the upper layers release the resources for that packet and start another transmission.

**Note:** In future, this function may be extended so that the data need not be copied by having the function return a “disposition” code (done, send pending, etc). At this point, you should move the data to some “safe” location before returning.

Deliver function

```c
static void
HRDWR_deliver(struct eth_drv_sc *sc)
```

This function is called from the “Network Delivery Thread” in order to let the device driver do the time-consuming work associated with receiving a packet — usually copying the entire packet from the hardware or a special memory location into the network stack’s memory.

After handling any outstanding incoming packets or pending transmission status, it can unmask the device’s interrupts, and free any relevant resources so it can process further packets.

It will be called when the interrupt handler for the network device has called

```c
eth_drv_dsr( vector, count, (cyg_addrword_t)sc );
```

to alert the system that “something requires attention.” This `eth_drv_dsr()` call must occur from within the interrupt handler’s DSR (not the ISR) or actually be the DSR, whenever it is determined that the device needs attention from the foreground. The third parameter (data in the prototype of `eth_drv_dsr()`) must be a valid struct `eth_drv_sc` pointer sc.
The reason for this slightly convoluted train of events is to keep the DSR (and ISR) execution time as short as possible, so that other activities of higher priority than network servicing are not denied the CPU by network traffic.

To deliver a newly-received packet into the network stack, the deliver routine must call

\[(sc->funs->eth_drv->recv)(sc, len);\]

which will in turn call the receive function, which we talk about next. See also the Section called Callback Receive function below.

**Receive function**

```c
static void
HRDWR_recv(
    struct eth_drv_sc *sc,
    struct eth_drv_sg *sg_list, int sg_len)

This function is a call back, only invoked after the upper-level function

\[(sc->funs->eth_drv->recv)(struct eth_drv_sc *sc, int total_len)\]

has been called itself from your deliver function when it knows that a packet of data is available on the interface. The \[(sc->funs->eth_drv->recv)()\] function then arranges network buffers and structures for the data and then calls \[HRDWR_recv()\] to actually move the data from the interface.

A scatter-gather list (struct eth_drv_sg) is used once more, just like in the send case.

**Poll function**

```c
static void
HRDWR_poll(struct eth_drv_sc *sc)

This function is used when in a non-interrupt driven system, e.g. when interrupts are completely disabled. This allows the driver time to check whether anything needs doing either for transmission, or to check if anything has been received, or if any other processing needs doing.

It is perfectly correct and acceptable for the poll function to look like this:

```c
static void
HRDWR_poll(struct eth_drv_sc *sc)
{
    my_interrupt_ISR(sc);
    HRDWR_deliver(struct eth_drv_sc *sc);
}
```

provided that both the ISR and the deliver functions are idempotent and harmless if called when there is no attention needed by the hardware. Some devices might not need a call to the ISR here if the deliver function contains all the “intelligence.”
Chapter 59. Generic Ethernet Device Driver

Interrupt-vector function

static int
HRDWR_int_vector(struct eth_drv_sc *sc)

This function returns the interrupt vector number used for receive interrupts. This is so that the common GDB stubs can detect when to check for incoming “CTRL-C” packets (used to asynchronously halt the application) when debugging over ethernet. The GDB stubs need to know which interrupt the ethernet device uses so that they can mask or unmask that interrupt as required.

Upper Layer Functions

Upper layer functions are called by drivers to deliver received packets or transmission completion status back up into the network stack.

These functions are defined by the hardware independent upper layers of the networking driver support. They are present to hide the interfaces to the actual networking stack so that the hardware drivers may be used by different network stack implementations without change.

These functions require a pointer to a struct eth_drv_sc which describes the interface at a logical level. It is assumed that the low level hardware driver will keep track of this pointer so it may be passed “up” as appropriate.

Callback Init function

void (sc->funs->eth_drv->init)(
   struct eth_drv_sc *sc, unsigned char *enaddr)

This function establishes the device at initialization time. It should be called once per device instance only, from the initialization function, if all is well (see the Section called Init function). The hardware should be totally initialized (not “started”) when this function is called.

Callback Tx-Done function

void (sc->funs->eth_drv->tx_done)(
   struct eth_drv_sc *sc,
   unsigned long key, int status)

This function is called when a packet completes transmission on the interface. The key value must be one of the keys provided to HRDWR_send() above. The value status should be non-zero (details currently undefined) to indicate that an error occurred during the transmission, and zero if all was well.

It should be called from the deliver function (see the Section called Deliver function) or poll function (see the Section called Poll function).

Callback Receive function

void (sc->funs->eth_drv->recv)(struct eth_drv_sc *sc, int len)
This function is called to indicate that a packet of length \( \text{len} \) has arrived at the interface. The callback \( \text{HRDWR}_\text{recv() \_} \) function described above will be used to actually unload the data from the interface into buffers used by the device independent layers.

It should be called from the deliver function (see the Section called Deliver function) or poll function (see the Section called Poll function).

### Calling graph for Transmission and Reception

It may be worth clarifying further the flow of control in the transmit and receive cases, where the hardware driver does use interrupts and so DSRs to tell the “foreground” when something asynchronous has occurred.

#### Transmission

1. Some foreground task such as the application, SNMP “daemon”, DHCP management thread or whatever, calls into network stack to send a packet, or the stack decides to send a packet in response to incoming traffic such as a “ping” or ARP request.
2. The driver calls the \( \text{HRDWR}_\text{can\_send()} \) function in the hardware driver.
3. \( \text{HRDWR}_\text{can\_send()} \) returns the number of available "slots" in which it can store a pending transmit packet. If it cannot send at this time, the packet is queued outside the hardware driver for later; in this case, the hardware is already busy transmitting, so expect an interrupt as described below for completion of the packet currently outgoing.
4. If it can send right now, \( \text{HRDWR}_\text{send()} \) is called. \( \text{HRDWR}_\text{send()} \) copies the data into special hardware buffers, or instructs the hardware to “send that.” It also remembers the key that is associated with this tx request.
5. These calls return … time passes …
6. Asynchronously, the hardware makes an interrupt to say “transmit is done.” The ISR quietens the interrupt source in the hardware and requests that the associated DSR be run.
7. The DSR calls (or \( \text{is} \)) the \( \text{eth\_drv\_dsr()} \) function in the generic driver.
8. \( \text{eth\_drv\_dsr()} \) in the generic driver awakens the “Network Delivery Thread” which calls the deliver function \( \text{HRDWR}_\text{deliver()} \) in the driver.
9. The deliver function realizes that a transmit request has completed, and calls the callback tx-done function \( \{\text{sc->funs->eth\_drv->tx\_done}()\} \) with the same key that it remembered for this tx.
10. The callback tx-done function uses the key to find the resources associated with this transmit request; thus the stack knows that the transmit has completed and its resources can be freed.
11. The callback tx-done function also enquires whether \( \text{HRDWR}_\text{can\_send()} \) now says “yes, we can send” and if so, dequeues a further transmit request which may have been queued as described above. If so, then \( \text{HRDWR}_\text{send()} \) copies the data into the hardware buffers, or instructs the hardware to “send that” and remembers the new key, as above. These calls then all return to the “Network Delivery Thread” which then sleeps, awaiting the next asynchronous event.
12. All done …


**Receive**

1. Asynchronously, the hardware makes an interrupt to say “there is ready data in a receive buffer.” The ISR quietens the interrupt source in the hardware and requests that the associated DSR be run.

2. The DSR calls (or is) the `eth_drv_dsr()` function in the generic driver.

3. `eth_drv_dsr()` in the generic driver awakens the “Network Delivery Thread” which calls the deliver function `HRDWR_deliver()` in the driver.

4. The deliver function realizes that there is data ready and calls the callback receive function `(sc->funs->eth_drv->recv)()` to tell it how many bytes to prepare for.

5. The callback receive function allocates memory within the stack (eg. MBUFs in BSD/Unix style stacks) and prepares a set of scatter-gather buffers that can accommodate the packet.

6. It then calls back into the hardware driver routine `HRDWR_recv()`. `HRDWR_recv()` must copy the data from the hardware’s buffers into the scatter-gather buffers provided, and return.

7. The network stack now has the data in-hand, and does with it what it will. This might include recursive calls to transmit a response packet. When this all is done, these calls return, and the “Network Delivery Thread” sleeps once more, awaiting the next asynchronous event.
XXII. Ethernet PHY Device Support
Chapter 60. Ethernet PHY Device Support

Ethernet PHY Device API

Modern ethernet subsystems are often separated into two pieces, the media access controller (sometimes known as a MAC) and the physical device or line interface (often referred to as a PHY). In this case, the MAC handles generating and parsing physical frames and the PHY handles how this data is actually moved to/from the wire. The MAC and PHY communicate via a special protocol, known as MII. This MII protocol can handle control over the PHY which allows for selection of such transmission criteria as line speed, duplex mode, etc.

In most cases, ethernet drivers only need to bother with the PHY during system initialization. Since the details of the PHY are separate from the MAC, there are different drivers for each. The drivers for the PHY are described by a set of exported functions which are commonly used by the MAC. The primary use of these functions currently is to initialize the PHY and determine the status of the line connection.

The connection between the MAC and the PHY differs from MAC to MAC, so the actual routines to manipulate this data channel are a property of the MAC instance. Furthermore, there are many PHY devices each with their own internal operations. A complete MAC/PHY driver setup will be comprised of the MAC MII access functions and the PHY internal driver.

A driver instance is contained within a eth_phy_access_t:

```c
#define PHY_BIT_LEVEL_ACCESS_TYPE 0
#define PHY_REG_LEVEL_ACCESS_TYPE 1

typedef struct {
    int ops_type; // 0 => bit level, 1 => register level
    bool init_done;
    void (*init)(void);
    void (*reset)(void);
    union {
        struct {
            void (*set_data)(int);
            int (*get_data)(void);
            void (*set_clock)(int);
            void (*set_dir)(int);
        } bit_level_ops;
        struct {
            void (*put_reg)(int reg, int unit, unsigned short data);
            bool (*get_reg)(int reg, int unit, unsigned short *data);
        } reg_level_ops;
    } ops;
    int phy_addr;
    struct _eth_phy_dev_entry *dev; // Chip access functions
} eth_phy_access_t;

struct _eth_phy_dev_entry {
    char *name;
    unsigned long id;
    bool (*stat)(eth_phy_access_t *f, int *stat);
};
```

The `dev` element points to the PHY specific support functions. Currently, the only function which must be defined is `stat()`.

The MAC-MII-PHY interface is a narrow connection, with commands and status moving between the MAC and PHY using a bit-serial protocol. Some MAC devices contain the intelligence to run this protocol, exposing
a mechanism to access PHY registers one at a time. Other MAC devices may only provide access to the MII data lines (or even still, this may be considered completely separate from the MAC). In these cases, the PHY support layer must handle the serial protocol. The choice between the access methods is in the `ops_type` field. If it has the value `PHY_BIT_LEVEL_ACCESS_TYPE`, then the PHY device layer will run the protocol, using the access functions `set_data()`, `get_data()`, `set_clock()`, `set_dir()` are used to control the MII signals and run the protocol. If `ops_type` has the value `PHY_REG_LEVEL_ACCESS_TYPE`, then the routines `put_reg()`, and `get_reg()` are used to access the PHY registers.

Two additional functions may be defined. These are `init()`, and `reset()`. The purpose of these functions is for gross-level management of the MII interface. The `init()` function will be called once, at system initialization time. It should do whatever operations are necessary to prepare the MII channel. In the case of `PHY_BIT_LEVEL_ACCESS_TYPE` devices, `init()` should prepare the signals for use, i.e. set up the appropriate parallel port registers, etc. The `reset()` function may be called by a driver to cause the PHY device to be reset to a known state. Not all drivers will require this and this function may not even be possible, so it’s use and behavior is somewhat target specific.

Currently, the only function required of device specific drivers is `stat()`. This routine should query appropriate registers in the PHY and return a status bitmap indicating the state of the physical connection. In the case where the PHY can auto-negotiate a line speed and condition, this information may be useful to the MAC to indicate what speed it should provide data, etc. The status bitmask contains these bits:

```c
#define ETH_PHY_STAT_LINK 0x0001 // Link up/down
#define ETH_PHY_STAT_100MB 0x0002 // Connection is 100Mb/10Mb
#define ETH_PHY_STAT_FDX 0x0004 // Connection is full/half duplex
```

Note: the usage here is that if the bit is set, then the condition exists. For example, if the `ETH_PHY_STAT_LINK` is set, then a physical link has been established.
XXIII. SNMP
Chapter 61. SNMP for eCos

Version

This is a port of UCD-SNMP-4.1.2

Originally this document said: See http://ucd-snmp.ucdavis.edu/ for details. And send them a postcard.

The project has since been renamed “net-snmp” and re-homed at http://net-snmp.sourceforge.net/ (http://net-snmp.sourceforge.net/) where various new releases (of the original, not eCos ports) are available.

The original source base from which we worked to create the eCos port is available from various archive sites such as ftp://ftp.freesnmp.com/mirrors/net-snmp/ (ftp://ftp.freesnmp.com/mirrors/net-snmp/) or ftp://sunsite.cnlab-switch.ch/mirror/ucd-snmp/ (ftp://sunsite.cnlab-switch.ch/mirror/ucd-snmp/) generally with this filename and details:

ucd-snmp-4.1.2.tar.gz... Nov 2 2000 1164k (ftp://ftp.freesnmp.com/mirrors/net-snmp/ucd-snmp-4.1.2.tar.gz)

SNMP packages in the eCos source repository

The SNMP/eCos package consists of two eCos packages; the SNMP library and the SNMP agent.

The sources are arranged this way partly for consistency with the original release from UCD, and so as to accommodate possible future use of the SNMP library without having an agent present. That could be used to build an eCos-based SNMP client application.

The library contains support code for talking SNMP over the net - the SNMP protocol itself - and a MIB file parser (ASN-1) which is not used in the agent case.

The agent contains the application specific handler files to get information about the system into the SNMP world, together with the SNMP agent thread (snmpd in UNIX terms).

MIBs supported

The standard set in MIB-II, together with the Ether-Like MIB, are supported by default. The MIB files used to compile the handlers in the agent and to “drive” the testing (snmpwalk et al under LINUX) are those acquired from that same UCD distribution.

These are the supported MIBs; all are below mib2 == 1.3.6.1.2:

```plaintext
system      { mib2  1 }
interfaces  { mib2  2 }
            [ address-translation “at” { mib2  3 } is deprecated ]
ip          { mib2  4 }
icmp        { mib2  5 }
tcp         { mib2  6 }
udp         { mib2  7 }
            [ exterior gateway protocol “egp” { mib2  8 } not supported ]
            [ cmot { mib2  9 } is “historic”, just a placeholder ]
dot3        { mib2 10 7 } == { transmission 7 } “EtherLike MIB”
snmp        { mib2 11 }
```

507
Chapter 61. SNMP for eCos

On inclusion of SNMPv3 support packages, the following MIBs are added to the default set of MIBs enumerated above:

- **snmpEngine**:  
  
  snmpEngine { snmpFrameworkMIBObjects 1 }  
  SNMP-FRAMEWORK-MIB, as described in RFC-2571 for support of SNMPv3 framework.

- **usmStats**:  
  
  usmStats { usmMIBObjects 1 }  
  SNMP-USER-BASED-SM-MIB, as specified in RFC-2574 for support of user based security model in SNMPv3 management domains.

- **usmUser**:  
  
  usmUser { usmMIBObjects 2 }  
  specified in RFC-2574 for support of user based security model in SNMPv3 management domains.

Changes to eCos sources

Small changes have been made in three areas:

1. Various hardware-specific ethernet drivers.
2. The generic ethernet device driver.
3. The OpenBSD TCP/IP networking package.

These changes were made in order to export information about the driver and the network that the SNMP agent must report. The changes were trivial in the case of the network stack, since it was already SNMP-friendly. The generic ethernet device driver was re-organized to have an extensive header file and to add a couple of APIs to extract statistics that the hardware-specific device drivers keep within themselves.

There may be a performance hit for recording that data; disabling a config option named something like CYGDBG_DEVS_ETH_xxxx_xxxx_KEEP_STATISTICS depending on the specific device driver will prevent that.

Not all platform ethernet device drivers export complete SNMP statistical information; if the exported information is missing, SNMP will report zero values for such data (in the dot3 MIB).

The interface chipset has an ID which is an OID; not all the latest greatest devices are listed in the available database, so new chipsets may need to be added to the client MIB, if not defined in those from UCD.

Starting the SNMP Agent

A routine to instantiate and start the SNMP agent thread in the default configuration is provided in PACKAGES/net/snmp/agent/VERSION/src/snmptask.c

It starts the snmpd thread at priority CYGPKG_NET_THREAD_PRIORITY+1 by default, i.e. one step less important than the TCP/IP stack service thread. It also statically creates and uses a very large stack of around 100 KiloBytes. To use that convenience function, this code fragment may be copied (in plain C).

```c
#ifdef CYGPKG_SNMPAGENT
{  
  extern void cyg_net_snmp_init(void);  
  cyg_net_snmp_init();
}
#endif
```

In case you need to perform initialization, for example setting up SNMPv3 security features, when the snmp agent starts and every time it restarts, you can register a callback function by simply writing the global variable:

...
externC void (*snmpd_reinit_function)( void );

with a suitable function pointer.

The entry point to the SNMP agent is

externC void snmpd( void (*initfunc)( void ) );

so you can of course easily start it in a thread of your choice at another priority instead if required, after performing whatever other initialization your SNMP MIBs need. A larger than default stacksize is required. The initfunc parameter is the callback function mentioned above — a NULL parameter there is safe and obviously means no callback is registered.

Note that if you call snmpd(); yourself and do not call cyg_net_snmp_init(); then that routine, global variable, and the default large stack will not be used. This is the recommended way control such features from your application; create and start the thread yourself at the appropriate moment.

Other APIs from the snmpd module are available, specifically:

void SnmpdShutDown(int a);

which causes the snmpd to restart itself — including the callback to your init function — as soon as possible.

The parameter a is ignored. It is there because in snmpd’s “natural environment” this routine is a UNIX signal handler.

The helper functions in the network stack for managing DHCP leases will call SnmpdShutDown() when necessary, for example if network interfaces go down and/or come up again.

Configuring eCos

To use the SNMP agent, the SNMP library and agent packages must be included in your configuration. To incorporate the stack into your configuration select the SNMP library and SNMP agent packages in the eCos Configuration Tool, or at the command line type:

$ ecosconfig add snmplib snmpagent

After adding the networking, common ethernet device drivers, snmp library and snmp agent packages, there is no configuration required. However there are a number of configuration options that can be set such as some details for the System MIB, and disabling SNMPv3 support (see below).

Starting the SNMP agent is not integrated into network tests other than snmpping below, nor is it started automatically in normal eCos startup - it is up to the application to start the agent when it is ready, at least after the network interfaces are both 'up'.

Version usage (v1, v2 or v3)

The default build supports all three versions of the SNMP protocol, but without any dispatcher functionality (rfc 2571, section 3.1.1.2). This has the following implications:

1. There is no community authentication for v1 and v2c.
2. Security provided by v3 can be bypassed by using v1/v2c protocol.

To provide the dispatcher with rfc 2571 type functionality, it is required to set up security models and access profiles. This can be provided in the normal Unix style by writing the required configurations in snmpd.conf
file. Application code may setup profiles in `snmpd.conf` and optionally set the environment variable `SNMPCONFPATH` to point to the file if it is not in the usual location. The whole concept works in the usual way as with the standard UCD-SNMP distribution.

### Traps

The support of the `trapsink` command in the `snmpd.conf` file is not tested and there may be problems for it working as expected. Moreover, in systems that do not have filesystem support, there is no way to configure a trap-session in the conventional way.

For reasons mentioned above, applications need to initialize their own trap sessions and pass it the details of trap-sink. The following is a small sample for initializing a v1 trap session:

```c
typedef struct trap {
    unsigned char ip [4];
    unsigned int port;
    unsigned char community [256];
} trap;

trap trapsink;
unsigned char sink [16];
...
...

if (trapsink.ip != 0) {
    sprintf (sink, "%d.%d.%d.%d",
             trapsink[0], trapsink[1], trapsink[2], trapsink[3]);
    if (create_trap_session (sink,
                             trapsink.port,
                             (char *)trapsink.community,
                             SNMP_VERSION_1,
                             SNMP_MSG_TRAP) == 0) {
        log_error ("Creation of trap session failed \n");
    }
}
```

### `snmpd.conf` file

Using `snmpd.conf` requires the inclusion of one of the file-system packages (eg. CYGPKG_RAMFS) and CYGPKG_FILEIO. With these two packages included, the SNMP sub-system will read the `snmpd.conf` file from the location specified in `SNMPCONFPATH`, or the standard builtin locations, and use these profiles. Only the profiles specified in the `ACCESS-CONTROL` section of `snmpd.conf` file have been tested and shown to work. Other profiles which have been implemented in UCD-SNMP-4.1.2’s `snmpd.conf` may not work because the sole purpose of adding support for the `snmpd.conf` file has been to set up `ACCESS-CONTROL` models.

At startup, the SNMP module tries to look for file `snmp.conf`. If this file is not available, the module successively looks for files `snmpd.conf`, `snmp.local.conf` and `snmpd.local.conf` at the locations pointed to by `SNMPCONFPATH` environment variable. In case `SNMPCONFPATH` is not defined, the search sequence is carried out in default directories. The default directories are `/usr/share/snmp`, `/usr/local/share/snmp` and `$HOME/.snmp`. The configurations read from these files are used to control both, SNMP applications and the SNMP agent; in the usual UNIX fashion.

The inclusion of `snmpd.conf` support is enabled by default when suitable filesystems and FILEIO packages are active.
Test cases

Currently only one test program is provided which uses SNMP.

"snmpping" in the SNMP agent package runs the ping test from the TCPIP package, with the snmpd running also. This allows you to interrogate it using host tools of your choice. It supports MIBs as documented above, so eg. `snmpwalk <hostname> public dot3` under Linux/UNIX should have the desired effect.

For serious testing, you should increase the length of time the test runs by setting `CYGNUM_SNMPAGENT_TESTS_ITERATIONS` to something big (e.g., 999999). Build the test (make -C net/snmp/agent/current tests) and run it on the target.

Then start several jobs, some for pinging the board (to make the stats change) and some for interrogating the snmpd. Set $IP to whatever IP address the board has:

```
# in a root shell, for flood ping
while(1)
do
ping -f -c 3001 $IP
sleep 5
ping -c 32 -s 2345 $IP
end

# have more than one of these going at once
setenv MIBS all
while(1)
snmpwalk -OS $IP public
date
end
```

Leave to run for a couple of days or so to test stability.

The test program can also test snmpd.conf support. It tries to build a minimal snmpd.conf file on a RAM filesystem and passes it to the snmp sub-system. With this profile on target, the following snmp[cmd] (cmd=walk, get, set) should work :

```
  snmp[cmd] -v1 $IP crux $OID
  snmp[cmd] -v2 $IP crux $OID
  snmp[cmd] -v3 $IP -u root -L noAuthNoPriv $OID
  snmp[cmd] -v3 $IP -u root -L authNoPriv -A MD5 -a md5passwd $OID
```

The following commands would however fail since they violate the access model :

```
  snmp[cmd] $IP public $OID
  snmp[cmd] -v1 $IP public $OID
  snmp[cmd] -v2c $IP public $OID
  snmp[cmd] -v3 $IP -u no_user -L noAuthNoPriv $OID
  snmp[cmd] -v3 $IP -u root -L authNoPriv -A MD5 -a badpasswd $OID
```

SNMP clients and package use

SNMP clients may use these packages, but this usage is currently untested: the reason why this port to eCos exists is to acquire the SNMP agent. The fact that that the SNMP API (for clients) exists is a side-effect. See the standard man page SNMP_API(3) for details. There are further caveats below about client-side use of the SNMP library.

All of the SNMP header files are installed beneath .../include/ucd-snmp in the install tree. The SNMP code itself assumes that directory is on its include path, so we recommend that client code does the same. Further,
Chapter 61. SNMP for eCos

like the TCP/IP stack, compiling SNMP code requires definition of \_KERNEL and \_ECOS, and additionally
IN\_UCD\_SNMP\_SOURCE.

Therefore, add all of these to your compile lines if you wish to include SNMP header files:

-D\_KERNEL
-D\_ECOS
-D\_IN\_UCD\_SNMP\_SOURCE=1
-I$(PREFIX)/include/ucd-snmp

Unimplemented features

Currently, the filesystem and persistent storage areas are left undone, to be implemented by the application.

The SNMP library package is intended to support client and agent code alike. It therefore contains lots of
assumptions about the presence of persistent storage ie. a filesystem. Currently, by default, eCos has no such
thing, so those areas have been simply commented out and made to return empty lists or say “no data here.”

Specifically the following files have omitted/unimplemented code:

PACKAGES/net/snmp/lib/VERSION/src/parse.c
contains code to enumerate MIB files discovered in the system MIB directories (“/usr/share/snmp/mibs”),
and read them all in, building data structures that are used by client programs to interrogate an agent. This is
not required in an agent, so the routine which enumerates the directories returns an empty list.

PACKAGES/net/snmp/lib/VERSION/src/read_config.c contains two systems:

The first tries to read the configuration file as described in the snmpd.conf file section and the second system
contains code to record persistent data as files in a directory (typically /var/ucd-snmp) thus preserving the
state permanently.

The first part is partially implemented to support multiple profiles and enables dispatcher functionality as
discussed in the Section called Version usage (v1, v2 or v3). The second part is not supported at all in the
default implementation. As required, a cleaner interface to permit application code to manage persistent data
will be developed in consultation with customers.

MIB Compiler

In the directory /snmp/agent/VERSION/utils/mib2c, there are the following files:

README-eCos notes about running with a nonstandard
perl path.
README.mib2c the README from UCD; full instructions on
using mib2c
mib2c the perl program
mib2c.conf a configuration file altered to include the
eCos/UCD
mib2c.conf-ORIG copyright and better #include paths; and
the ORIGINAL.
mib2c.storage.conf other config files, not modified.
mib2c.vartypes.conf

mib2c is provided BUT it requires the SNMP perl package SNMP-3.1.0, and that in turn requires perl
nsPerl5.005_03 (part of Red Hat Linux from 6.0, April 1999).
These are available from the CPAN (“the Comprehensive Perl Archive Network”) as usual; http://www.cpan.org/ and links from there. Specifically:

- PERL itself: http://people.netscape.com/kristian/nsPerl/
- SNMP.pl http://www.cpan.org/modules/01modules.index.html
- http://www.cpan.org/authors/id/G/GS/GSM/SNMP.tar.gz

(note that the .tar.gz files are not browsable)

For documentation on the files produced, see the documentation available at http://ucd-snmp.ucdavis.edu/ in general, and file AGENT.txt in particular.

It is likely that the output of mib2c will be further customized depending on eCos customer needs; it’s easy to do this by editing the mib2c.conf file to add or remove whatever you need with the resulting C sources.

The UCD autoconf-style configuration does not apply to eCos. So if you add a completely new MIB to the agent, and support it using mib2c so that the my_new_mib.c file contains a init_my_new_mib() routine to register the MIB handler, you will also need to edit a couple of control files; these claim to be auto-generated, but in the eCos release, they’re not, don’t worry.

PACKAGES/net/snmp/agent/VERSION/include/mib_module_includes.h contains a number of lines like

```c
#include "mibgroup/mibII/interfaces.h"
```

so add your new MIB thus:

```c
#include "mibgroup/mibII/my_new_mib.h"
```

PACKAGES/net/snmp/agent/VERSION/include/mib_module_inits.h contains a number of lines like

```c
init_interfaces();
init_dot3();
```

and so on; add your new MIB as follows:

```c
init_my_new_mib();
```

and this should work correctly.

### snmpd.conf

```c
NAME
share/snmp/snmpd.conf - configuration file for the ucd-snmp SNMP agent.
```

DESCRIPTION
snmpd.conf is the configuration file which defines how the ucd-snmp SNMP agent operates. These files may contain any of the directives found in the DIRECTIVES section below. This file is not required for the agent to operate and report mib entries.

PLEASE READ FIRST
First, make sure you have read the snmp_config(5) manual page that describes how the ucd-snmp configuration files operate, where they are located and how they all work together.

EXTENSIBLE-MIB
The ucd-snmp SNMP agent reports much of its information through queries to the 1.3.6.1.4.1.2021 section of the mib tree. Every mib in this section has the following table entries in it.

.1 -- index
   This is the table’s index numbers for each of the DIRECTIVES listed below.

.2 -- name
   The name of the given table entry. This should be unique, but is not required to be.

.100 -- errorFlag
   This is a flag returning either the integer value 1 or 0 if an error is detected for this table entry.

.101 -- errorMsg
   This is a DISPLAY-STRING describing any error triggering the errorFlag above.

.102 -- errorFix
   If this entry is SNMPset to the integer value of 1 AND the errorFlag defined above is indeed a 1, a program or script will get executed with the table entry name from above as the argument. The program to be executed is configured in the config.h file at compile time.

Directives
proc NAME
proc NAME MAX
proc NAME MAX MIN

Checks to see if the NAME’ed processes are running on the agent’s machine. An error flag (1) and a description message are then passed to the 1.3.6.1.4.1.2021.2.100 and 1.3.6.1.4.1.2021.2.101 mib tables (respectively) if the NAME’ed program is not found in the process table as reported by "/bin/ps -e".

If MAX and MIN are not specified, MAX is assumed to be infinity and MIN is assumed to be 1.

If MAX is specified but MIN is not specified, MIN
is assumed to be 0.

procfix NAME PROG ARGS

This registers a command that knows how to fix errors with the given process NAME. When 1.3.6.1.4.1.2021.2.102 for a given NAMEd program is set to the integer value of 1, this command will be called. It defaults to a compiled value set using the PROCFIXCMD definition in the config.h file.

exec NAME PROG ARGS

exec MIBNUM NAME PROG ARGS

If MIBNUM is not specified, the agent executes the named PROG with arguments of ARGS and returns the exit status and the first line of the STDOUT output of the PROG program to queries of the 1.3.6.1.4.1.2021.8.100 and 1.3.6.1.4.1.2021.8.101 mib tables (respectively). All STDOUT output beyond the first line is silently truncated.

If MIBNUM is specified, it acts as above but returns the exit status to MIBNUM.100.0 and the entire STDOUT output to the table MIBNUM.101 in a mib table. In this case, the MIBNUM.101 mib contains the entire STDOUT output, one mib table entry per line of output (ie, the first line is output as MIBNUM.101.1, the second at MIBNUM.101.2, etc...).

Note: The MIBNUM must be specified in dotted-integer notation and can not be specified as ".iso.org.dod.internet..." (should instead be

Note: The agent caches the exit status and STDOUT of the executed program for 30 seconds after the initial query. This is to increase speed and maintain consistency of information for consecutive table queries. The cache can be flushed by a snmp-set request of integer(1) to 1.3.6.1.4.1.2021.100.VER-CLEARCACHE.

execfix NAME PROG ARGS

This registers a command that knows how to fix errors with the given exec or sh NAME. When 1.3.6.1.4.1.2021.8.102 for a given NAMEd entry is set to the integer value of 1, this command will be called. It defaults to a compiled value set using the EXECFIXCMD definition in the config.h file.

disk PATH

disk PATH [ MINSPACE | MINPERCENT% ]

Checks the named disks mounted at PATH for available disk space. If the disk space is less than MINSPACE (kB) if specified or less than MINPERCENT (%) if a % sign is specified, or DEFDISKMINIMUMSPACE (kB) if not specified, the associated
entry in the 1.3.6.1.4.1.2021.9.100 mib table will be set to (1) and a descriptive error message will be returned to queries of 1.3.6.1.4.1.2021.9.101.

load MAX1

load MAX1 MAX5

load MAX1 MAX5 MAX15

Checks the load average of the machine and returns an error flag (1), and a text-string error message to queries of 1.3.6.1.4.1.2021.10.100 and 1.3.6.1.4.1.2021.10.101 (respectively) when the 1-minute, 5-minute, or 15-minute averages exceed the associated maximum values. If any of the MAX1, MAX5, or MAX15 values are unspecified, they default to a value of DEFMAXLOADAVE.

file FILE [MAXSIZE]

Monitors file sizes and makes sure they don't grow beyond a certain size. MAXSIZE defaults to infinite if not specified, and only monitors the size without reporting errors about it.

Errors

Any errors in obtaining the above information are reported via the 1.3.6.1.4.1.2021.101.100 flag and the 1.3.6.1.4.1.2021.101.101 text-string description.

SMUX SUB-AGENTS

To enable and SMUX based sub-agent, such as gated, use the smuxpeer configuration entry

smuxpeer OID PASS

For gated a sensible entry might be

.1.3.6.1.4.1.4.1.3 secret

ACCESS CONTROL

snmpd supports the View-Based Access Control Model (vacm) as defined in RFC 2275. To this end, it recognizes the following keywords in the configuration file: com2sec, group, access, and view as well as some easier-to-use wrapper directives: rocommunity, rwcommunity, rouser, rwuser.

rocommunity COMMUNITY [SOURCE] [OID]

rwcommunity COMMUNITY [SOURCE] [OID]

These create read-only and read-write communities that can be used to access the agent. They are a quick method of using the following com2sec, group, access, and view directive lines. They are not as efficient either, as groups aren't created so the tables are possibly larger. In other words: don't use these if you have complex situations to set up.

The format of the SOURCE is token is described in the com2sec directive section below. The OID token restricts access for that community to everything
below that given OID.

rouser USER [noauth|auth|priv] [OID]

rwuser USER [noauth|auth|priv] [OID]
Creates a SNMPv3 USM user in the VACM access configuration tables. Again, its more efficient (and powerful) to use the combined com2sec, group, access, and view directives instead.

The minimum level of authentication and privacy the user must use is specified by the first token (which defaults to "auth"). The OID parameter restricts access for that user to everything below the given OID.

com2sec NAME SOURCE COMMUNITY
This directive specifies the mapping from a source/community pair to a security name. SOURCE can be a hostname, a subnet, or the word "default". A subnet can be specified as IP/MASK or IP/BITS. The first source/community combination that matches the incoming packet is selected.

group NAME MODEL SECURITY
This directive defines the mapping from security-model/securityname to group. MODEL is one of v1, v2c, or usm.

access NAME CONTEXT MODEL LEVEL PREFIX READ WRITE NOTIFY
The access directive maps from group/security model/security level to a view. MODEL is one of any, v1, v2c, or usm. LEVEL is one of noauth, auth, or priv. PREFIX specifies how CONTEXT should be matched against the context of the incoming pdu, either exact or prefix. READ, WRITE and NOTIFY specifies the view to be used for the corresponding access. For v1 or v2c access, LEVEL will be noauth, and CONTEXT will be empty.

view NAME TYPE SUBTREE [MASK]
The defines the named view. TYPE is either included or excluded. MASK is a list of hex octets, separated by "." or ":". The MASK defaults to "ff" if not specified.

The reason for the mask is, that it allows you to control access to one row in a table, in a relatively simple way. As an example, as an ISP you might consider giving each customer access to his or her own interface:

view cust1 included interfaces.ifTable.ifEntry.ifIndex.1 ff.a0
view cust2 included interfaces.ifTable.ifEntry.ifIndex.2 ff.a0

(interfaces.ifTable.ifEntry.ifIndex.1 == 1.3.6.1.2.1.2.2.1.1.1, ff.a0 == 11111111.10100000, which nicely covers up and including the row index, but lets the user vary the field of the row)

VACM Examples:

```bash
# sec.name source community
```
Chapter 61. SNMP for eCos

com2sec local localhost private
com2sec mynet 10.10.10.0/24 public
com2sec public default public

# sec.model sec.name
group mygroup v1 mynet
group mygroup v2c mynet
group mygroup usm mynet
group local v1 local
group local v2c local
group local usm local
group public v1 public
group public v2c public
group public usm public

# incl/excl subtree mask
view all included .1 80
view system included system fe
view mib2 included .iso.org.dod.internet.mgmt.mib-2 fc

# context sec.model sec.level prefix read write notify
access mygroup "" any noauth exact mib2 none none
access public "" any noauth exact system none none
access local "" any noauth exact all all all

Default VACM model
The default configuration of the agent, as shipped, is functionally equivalent to the following entries:
com2sec public default public
group public v1 public
group public v2c public
group public usm public
view all included .1
access public "" any noauth exact all none none

SNMPv3 CONFIGURATION

engineID STRING
The snmpd agent needs to be configured with an engineID to be able to respond to SNMPv3 messages. With this configuration file line, the engineID will be configured from STRING. The default value of the engineID is configured with the first IP address found for the hostname of the machine.

createUser username (MD5|SHA) authpassphrase [DES] [priv-passphrase]
This directive should be placed into the "/var/ucd-snmp/snmpd.conf" file instead of the other normal locations. The reason is that the information is read from the file and then the line is removed (eliminating the storage of the master password for that user) and replaced with the key that is derived from it. This key is a localized key, so that if it is stolen it can not be used to access other agents. If the password is stolen, however, it can be.

MD5 and SHA are the authentication types to use, but you must have built the package with openssl installed in order to use SHA. The only privacy protocol currently supported is DES. If the pri-
vacy passphrase is not specified, it is assumed to be the same as the authentication passphrase. Note that the users created will be useless unless they are also added to the VACM access control tables described above.

Warning: the minimum pass phrase length is 8 characters.

SNMPv3 users can be created at runtime using the snmpusm command.

SETTING SYSTEM INFORMATION

syslocation STRING

syscontact STRING

Sets the system location and the system contact for the agent. This information is reported by the 'system' table in the mibII tree.

authtrapenable NUMBER

Setting authtrapenable to 1 enables generation of authentication failure traps. The default value is 2 (disable).

trapcommunity STRING

This defines the default community string to be used when sending traps. Note that this command must be used prior to any of the following three commands that are intended use this community string.

trapsink HOST [COMMUNITY [PORT]]

trap2sink HOST [COMMUNITY [PORT]]

informsink HOST [COMMUNITY [PORT]]

These commands define the hosts to receive traps (and/or inform notifications). The daemon sends a Cold Start trap when it starts up. If enabled, it also sends traps on authentication failures. Multiple trapsink, trap2sink and informsink lines may be specified to specify multiple destinations. Use trap2sink to send SNMPv2 traps and informsink to send inform notifications. If COMMUNITY is not specified, the string from a preceding trapcommunity directive will be used. If PORT is not specified, the well known SNMP trap port (162) will be used.

PASS-THROUGH CONTROL

pass MIBOID EXEC

Passes entire control of MIBOID to the EXEC program. The EXEC program is called in one of the following three ways:

EXEC -g MIBOID

EXEC -n MIBOID
These call lines match to SNMP get and get-next requests. It is expected that the EXEC program will take the arguments passed to it and return the appropriate response through its stdout.

The first line of stdout should be the mib OID of the returning value. The second line should be the TYPE of value returned, where TYPE is one of the text strings: string, integer, unsigned, objectid, timeticks, ipaddress, counter, or gauge. The third line of stdout should be the VALUE corresponding with the returned TYPE.

For instance, if a script was to return the value integer value "42" when a request for .1.3.6.1.4.100 was requested, the script should return the following 3 lines:

```
.1.3.6.1.4.100
integer
42
```

To indicate that the script is unable to comply with the request due to an end-of-mib condition or an invalid request, simple exit and return no output to stdout at all. A snmp error will be generated corresponding to the SNMP NO-SUCH-NAME response.

EXEC -s MIBOID TYPE VALUE

For SNMP set requests, the above call method is used. The TYPE passed to the EXEC program is one of the text strings: integer, counter, gauge, timeticks, ipaddress, objid, or string, indicating the type of value passed in the next argument.

Return nothing to stdout, and the set will assumed to have been successful. Otherwise, return one of the following error strings to signal an error: not-writable, or wrong-type and the appropriate error response will be generated instead.

Note: By default, the only community allowed to write (ie snmpset) to your script will be the "private" community, or community #2 if defined differently by the "community" token discussed above. Which communities are allowed write access are controlled by the RWRITE definition in the snmplib/snmp_impl.h source file.

EXAMPLE

See the EXAMPLE.CONF file in the top level source directory for a more detailed example of how the above information is used in real examples.
RE-READING snmpd.conf and snmpd.local.conf

The ucd-snmp agent can be forced to re-read its configuration files. It can be told to do so by one of two ways:

1. An snmpset of integer(1) to 1.3.6.1.4.1.2021.100.VERUPDATECONFIG.

2. A "kill -HUP" signal sent to the snmpd agent process.

FILES

share/snmp/snmpd.conf

SEE ALSO

snmp_config(5), snmpd(1), EXAMPLE.conf, read_config(3).

27 Jan 2000

SNMPD.CONF(5)
Chapter 61. SNMP for eCos
XXIV. Embedded HTTP Server
Chapter 62. Embedded HTTP Server

Introduction

The eCos HTTPD package provides a simple HTTP server for use with applications in eCos. This server is specifically aimed at the remote control and monitoring requirements of embedded applications. For this reason the emphasis is on dynamically generated content, simple forms handling and a basic CGI interface. It is not intended to be a general purpose server for delivering arbitrary web content. For these purposes a port of the GoAhead web server is available from www.goahead.com.

This server is also capable of serving content using IPv6 when the eCos configuration contains IPv6.

Server Organization

The server consists of one or more threads running in parallel to any application threads and which serve web pages to clients. Apart from defining content, the application does not need to do anything to start the HTTP server.

The HTTP server is, by default, started by a static constructor. This simply creates an initial thread and sets it running. Since this is called before the scheduler is started, nothing will happen until the application calls cyg_scheduler_start(). The server thread can also be started explicitly by the application, see the CYGNUM_HTTPD_SERVER_AUTO_START option for details.

When the thread gets to run it first optionally delays for some period of time. This is to allow the application to perform any initialization free of any interference from the HTTP server. When the thread does finally run it creates a socket, binds it to the HTTP server port, and puts it into listen mode. It will then create any additional HTTPD server threads that have been configured before becoming a server thread itself.

Each HTTPD server thread simply waits for a connection to be made to the server port. When the connection is made it reads the HTTP request and extracts the filename being accessed. If the request also contains form data, this is also preserved. The filename is then looked up in a table.

Each table entry contains a filename pattern string, a pointer to a handler function, and a user defined argument for the function. Table entries are defined using the same link-time table building mechanism used to generate device tables. This is all handled by the CYG_HTTPD_TABLE_ENTRY() macro which has the following format:

```c
#include <cyg/httpd/httpd.h>

CYG_HTTPD_TABLE_ENTRY( __name, __pattern, __handler, __arg )
```

The __name argument is a variable name for the table entry since C does not allow us to define anonymous data structures. This name should be chosen so that it is unique and does not pollute the name space. The __pattern argument is the match pattern. The __handler argument is a pointer to the handler function and __arg the user defined value.

The link-time table building means that several different pieces of code can define server table entries, and so long as the patterns do not clash they can be totally oblivious of each other. However, note also that this mechanism does not guarantee the order in which entries appear, this depends on the order of object files in the link, which could vary from one build to the next. So any tricky pattern matching that relies on this may not always work.

A request filename matches an entry in the table if either it exactly matches the pattern string, or if the pattern ends in an asterisk, and it matches everything up to that point. So for example the pattern
"/monitor/threads.html" will only match that exact filename, but the pattern "/monitor/thread-*" will match "/monitor/thread-0040.html", "/monitor/thread-0100.html" and any other filename starting with "/monitor/thread-".

When a pattern is matched, the handler function is called. It has the following prototype:

```c
cyg_bool cyg_httpd_handler(FILE *client, char *filename, char *formdata, void *arg);
```

The `client` argument is the TCP connection to the client: anything output through this stream will be returned to the browser. The `filename` argument is the filename from the HTTP request and the `formdata` argument is any form response data, or NULL if none was sent. The `arg` argument is the user defined value from the table entry.

The handler is entirely responsible for generating the response to the client, both HTTP header and content. If the handler decides that it does not want to generate a response it can return `false`, in which case the table scan is resumed for another match. If no match is found, or no handler returns true, then a default response page is generated indicating that the requested page cannot be found.

Finally, the server thread closes the connection to the client and loops back to accept a new connection.

## Server Configuration

The HTTP server has a number of configuration options:

### CYGNUM_HTTPD_SERVER_PORT

This option defines the TCP port that the server will listen on. It defaults to the standard HTTP port number 80. It may be changed to a different number if, for example, another HTTP server is using the main HTTP port.

### CYGDAT_HTTPD_SERVER_ID

This is the string that is reported to the client in the "Server:" field of the HTTP header.

### CYGNUM_HTTPD_THREAD_COUNT

The HTTP server can be configured to use more than one thread to service HTTP requests. If you expect to serve complex pages with many images or other components that are fetched separately, or if any pages may take a long time to send, then it may be useful to increase the number of server threads. For most uses, however, the connection queuing in the TCP/IP stack and the speed with which each page is generated, means that a single thread is usually adequate.

### CYGNUM_HTTPD_THREAD_PRIORITY

The HTTP server threads can be run at any priority. The exact priority depends on the importance of the server relative to the rest of the system. The default is to put them in the middle of the priority range to provide reasonable response without impacting genuine high priority threads.
Chapter 62. Embedded HTTP Server

**CYGNUM_HTTPD_THREAD_STACK_SIZE**

This is the amount of stack to be allocated for each of the HTTPD threads. The actual stack size allocated will be this value plus the values of `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HTTPD_SERVER_BUFFER_SIZE`.

**CYGNUM_HTTPD_SERVER_BUFFER_SIZE**

This defines the size of the buffer used to receive the first line of each HTTP request. If you expect to use particularly long URLs or have very complex forms, this should be increased.

**CYGNUM_HTTPD_SERVER_AUTO_START**

This option causes the HTTP Daemon to be started automatically during system initialization. If this option is not set then the application must start the daemon explicitly by calling `cyg_httpd_startup()`. This option is set by default.

**CYGNUM_HTTPD_SERVER_DELAY**

This defines the number of system clock ticks that the HTTP server will wait before initializing itself and spawning any extra server threads. This is to give the application a chance to initialize properly without any interference from the HTTPD.

Support Functions and Macros

The emphasis of this server is on dynamically generated content, rather than fetching it from a filesystem. To do this the handler functions make calls to `fprintf()` and `fputs()`. Such handler functions would end up a mass of print calls, with the actual structure of the HTML page hidden in the format strings and arguments, making maintenance and debugging very difficult. Such an approach would also result in the definition of many, often only slightly different, format strings, leading to unnecessary bloat.

In an effort to expose the structure of the HTML in the structure of the C code, and to maximize the sharing of string constants, the `cyg/httpd/httpd.h` header file defines a set of helper functions and macros. Most of these are wrappers for predefined print calls on the `client` stream passed to the handler function. For examples of their use, see the System Monitor example.

**Note:** All arguments to macros are pointers to strings, unless otherwise stated. In general, wherever a function or macro has an `attr` or `__attr` parameter, then the contents of this string will be inserted into the tag being defined as HTML attributes. If it is a NULL or empty string it will be ignored.

HTTP Support

```c
void cyg_http_start( FILE *client, char *content_type, int content_length );
void cyg_http_finish( FILE *client );
#define html_begin(__client)
#define html_end( __client )
```

527
The function `cyg_http_start()` generates a simple HTTP response header containing the value of `CYGDAT_HTTPD_SERVER_ID` in the "Server" field, and the values of `content_type` and `content_length` in the "Content-type" and "Content-length" field respectively. The function `cyg_http_finish()` just adds an extra newline to the end of the output and then flushes it to force the data out to the client.

The macro `html_begin()` generates an HTTP header with a "text/html" content type followed by an opening "<html>" tag. `html_end()` generates a closing "</html>" tag and calls `cyg_http_finish()`.

### General HTML Support

```c
void cyg_html_tag_begin( FILE *client, char *tag, char *attr );
void cyg_html_tag_end( FILE *client, char *tag );
#define html_tag_begin( __client, __tag, __attr )
#define html_tag_end( __client, __tag )
#define html_head( __client, __title, __meta )
#define html_body_begin( __client, __attr )
#define html_body_end( __client )
#define html_heading( __client, __level, __heading )
#define html_para_begin( __client, __attr )
#define html_url( __client, __text, __link )
#define html_image( __client, __source, __alt, __attr )
```

The function `cyg_html_tag_begin()` generates an opening tag with the given name. The function `cyg_html_tag_end()` generates a closing tag with the given name. The macros `html_tag_begin()` and `html_tag_end()` are just wrappers for these functions.

The macro `html_head()` generates an HTML header section with `__title` as the title. The `__meta` argument defines any meta tags that will be inserted into the header. `html_body_begin()` and `html_body_end()` generate HTML body begin and end tags.

`html_heading()` generates a complete HTML header where `__level` is a numerical level, between 1 and 6, and `__heading` is the heading text. `html_para_begin()` generates a paragraph break.

`html_url()` inserts a URL where `__text` is the displayed text and `__link` is the URL of the linked page. `html_image()` inserts an image tag where `__source` is the URL of the image to be included and `__alt` is the alternative text for when the image is not displayed.

### Table Support

```c
#define html_table_begin( __client, __attr )
#define html_table_end( __client )
#define html_table_header( __client, __content, __attr )
#define html_table_row_begin( __client, __attr )
#define html_table_row_end( __client )
#define html_table_data_begin( __client, __attr )
#define html_table_data_end( __client )
```

`html_table_begin()` starts a table and `html_table_end()` end it. `html_table_header()` generates a simple table column header containing the string `__content`.

`html_table_row_begin()` and `html_table_row_end()` begin and end a table row, and similarly `html_table_data_begin()` and `html_table_data_end()` begin and end a table entry.
## Forms Support

```
define html_form_begin( __client, __url, __attr )
define html_form_end( __client )
define html_form_input( __client, __type, __name, __value, __attr )
define html_form_input_radio( __client, __name, __value, __checked )
define html_form_input_checkbox( __client, __name, __value, __checked )
define html_form_input_hidden( __client, __name, __value )
define html_form_select_begin( __client, __name, __attr )
define html_form_option( __client, __value, __label, __selected )
define html_form_select_end( __client )
void cyg_formdata_parse( char *data, char *list[], int size );
char *cyg_formlist_find( char *list[], char *name );
```

`html_form_begin()` begins a form, the `__url` argument is the value for the `action` attribute.

`html_form_end()` ends the form.

`html_form_input()` defines a general form input element with the given type, name and value.

`html_form_input_radio` creates a radio button with the given name and value; the `__checked` argument is a boolean expression that is used to determine whether the `checked` attribute is added to the tag. Similarly `html_form_input_checkbox()` defines a checkbox element. `html_form_input_hidden()` defines a hidden form element with the given name and value.

`html_form_select_begin()` begins a multiple choice menu with the given name. `html_form_select_end()` end it. `html_form_option()` defines a menu entry with the given value and label; the `__selected` argument is a boolean expression controlling whether the selected attribute is added to the tag.

`cyg_formdata_parse()` converts a form response string into an NULL-terminated array of “name=value” entries. The `data` argument is the string as passed to the handler function; note that this string is not copied and will be updated in place to form the list entries. `list` is a pointer to an array of character pointers, and is `size` elements long. `cyg_formlist_find()` searches a list generated by `cyg_formdata_parse()` and returns a pointer to the value part of the string whose name part matches `name`; if there is no match it will return NULL.

## Predefined Handlers

```
cyg_bool cyg_httpd_send_html( FILE *client, char *filename, char *request, void *arg );

typedef struct
{
    char *content_type;
    cyg_uint32 content_length;
    cyg_uint8 *data;
} cyg_httpd_data;
#define CYG_HTTPD_DATA( __name, __type, __length, __data )
cyg_bool cyg_httpd_send_data( FILE *client, char *filename, char *request, void *arg );
```

The HTTP server defines a couple of predefined handers to make it easier to deliver simple, static content.

`cyg_httpd_send_html()` takes a NULL-terminated string as the argument and sends it to the client with an HTTP header indicating that it is HTML. The following is an example of its use:

```c
char cyg_html_message[] = "<head><title>Welcome</title></head>
""<body><h2>Welcome to my Web Page</h2></body>
"
```

`CYG_HTTPD_TABLE_ENTRY` defines a table entry with the given filename and URL.
cyg_httpd_send_html,
cyg_html_message );

cyg_httpd_send_data() Sends arbitrary data to the client. The argument is a pointer to a cyg_httpd_data structure that defines the content type and length of the data, and a pointer to the data itself. The CYG_HTTPD_DATA() macro automates the definition of the structure. Here is a typical example of its use:

static cyg_uint8 ecos_logo_gif[] = {
    ...
};

CYG_HTTPD_DATA( cyg_monitor_ecos_logo_data,
    "image/gif",
    sizeof(ecos_logo_gif),
    ecos_logo_gif );

CYG_HTTPD_TABLE_ENTRY( cyg_monitor_ecos_logo,
    "/monitor/ecos.gif",
    cyg_httpd_send_data,
    &cyg_monitor_ecos_logo_data );

System Monitor

Included in the HTTPD package is a simple System Monitor that is intended to act as a test and an example of how to produce servers. It is also hoped that it might be of some use in and of itself.

The System Monitor is intended to work in the background of any application. Adding the network stack and the HTTPD package to any configuration will enable the monitor by default. It may be disabled by disabling the CYGPKG_HTTPD_MONITOR option.

The monitor is intended to be simple and self-explanatory in use. It consists of four main pages. The thread monitor page presents a table of all current threads showing such things as id, state, priority, name and stack dimensions. Clicking on the thread ID will link to a thread edit page where the thread’s state and priority may be manipulated. The interrupt monitor just shows a table of the current interrupts and indicates which are active. The memory monitor shows a 256 byte page of memory, with controls to change the base address and display element size. Note: Accessing invalid memory locations can cause memory exceptions and the program to crash. The network monitor page shows information extracted from the active network interfaces and protocols. Finally, if kernel instrumentation is enabled, the instrumentation page provides some controls over the instrumentation mechanism, and displays the instrumentation buffer.
XXV. FTP Client for eCos TCP/IP Stack

The ftpclient package provides an FTP (File Transfer Protocol) client for use with the TCP/IP stack in eCos. It supports both IPv4 and IPv6 and will use the DNS client, when its part of the eCos configuration.
Chapter 63. FTP Client Features

FTP Client API

This package implements an FTP client. The API is in include file `install/include/ftpclient.h` and it can be used thus:

```c
#include <network.h>
#include <ftpclient.h>
```

It looks like this:

```c
ftp_get
int ftp_get(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            char * buf,
            unsigned buf_size,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to retrieve a file from a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ‘/’ file separators, not ‘\’. The file is placed into `buf`. `buf` has maximum size `buf_size`. If the file is bigger than this, the transfer fails and FTP_TOOBIG is returned. Other error codes listed in the header can also be returned. If the transfer is successful the number of bytes received is returned.

```c
ftp_put
int ftp_put(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            char * buf,
            unsigned buf_size,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to send a file to a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ‘/’ file separators, not ‘\’. The contents of `buf` are placed into the file on the server. If an error occurs one of the codes listed will be returned. If the transfer is successful zero is returned.

```c
ftpclient_printf
void ftpclient_printf(unsigned error, const char * fmt, ...);
```

`ftp_get()` and `ftp_put` take a pointer to a function to use for printing out diagnostic and error messages. This is a sample implementation which can be used if you don’t want to implement the function yourself. `error` will be true when the message to print is an error message. Otherwise the message is diagnostic, eg. the commands sent and received from the server.
XXVI. Simple Network Time Protocol Client

The SNTP package provides implementation of a client for RFC 2030, the Simple Network Time Protocol (SNTP). The client listens for broadcasts or IPv6 multicasts from an NTP server and uses the information received to set the system clock. It can also be configured to send SNTP time requests to specific NTP servers using SNTP’s unicast mode.
Chapter 64. The SNTP Client

Starting the SNTP client

The sntp client is implemented as a thread which listens for NTP broadcasts and IPv6 multicasts, and optionally sends SNTP unicast requests to specific NTP servers. This thread may be automatically started by the system if it receives a list of (S)NTP servers from the DHCP server and unicast mode is enabled. Otherwise it must be started by the user application. The header file `cyg/sntp/sntp.h` declares the function to be called. The thread is then started by calling the function:

```c
void cyg_sntp_start(void);
```

It is safe to call this function multiple times. Once started, the thread will run forever.

What it does

The SNTP client listens for NTP IPv4 broadcasts from any NTP servers, or IPv6 multicasts using the address fe00:0X::101, where X can be 2 (Link Local), 5 (Site-Local) or 0xe (Global). Such packets contain a timestamp indicating the current time. The packet also contains information about where the server is in the hierarchy of time servers. A server at the root of the time server tree normally has an atomic clock. Such a server is said to be at stratum 0. A time server which is synchronised to a stratum 0 server is said to be at stratum 1 etc. The client will accept any NTP packets from servers using version 3 or 4 of the protocol. When receiving packets from multiple servers, it will use the packets from the server with the lowest stratum. However, if there are no packets from this server for 10 minutes and another server is sending packets, the client will change servers.

If SNTP unicast mode is enabled via the CYGPKG_NET_SNTP_UNICAST option, the SNTP client can additionally be configured with a list of specific NTP servers to query. The general algorithm is as follows: if the system clock has not yet been set via an NTP time update, then the client will send out NTP requests every 30 seconds to all configured NTP servers. Once an NTP time update has been received, the client will send out additional NTP requests every 30 minutes in order to update the system clock. These requests are resent every 30 seconds until a response is received.

The system clock in eCos is accurate to 1 second. The SNTP client will change the system clock when the time difference with the received timestamp is greater than 2 seconds. The change is made as a step.

Configuring the unicast list of NTP servers

If SNTP unicast mode is enabled via the CYGPKG_NET_SNTP_UNICAST option, the SNTP client can be configured with a list of NTP servers to contact for time updates.

By default, this list is configured with NTP server information received from DHCP. The number of NTP servers that are extracted from DHCP can be configured with the CYGOPT_NET_SNTP_UNICAST_MAXDHCP option. This option can also be used to disable DHCP usage entirely.

The list of NTP servers can be manually configured with the following API function. Note that manual configuration will override any servers that were automatically configured by DHCP. But later reconfigurations by DHCP will override manual configurations. Hence it is not recommended to manually configure servers when CYGOPT_NET_SNTP_UNICAST is enabled.
#include <cyg/sntp/sntp.h>

void cyg_sntp_set_servers(struct sockaddr *server_list, cyg_uint32 num_servers);

This function takes an array of sockaddr structures specifying the IP address and UDP port of each NTP server to query. Currently, both IPv4 and IPv6 sockaddr structures are supported. The num_servers argument specifies how many sockaddr’s are contained in the array. The server_list array must be maintained by the caller. Once the array is registered with this function, it must not be modified by the caller until it is replaced or unregistered by another call to this function.

Calling this function with a server_list of NULL and a num_servers value of 0 unregisters any previously configured server_list array.

Finally, note that if this function is called with a non-empty server list, it will implicitly start the SNTP client if it has not already been started (i.e. it will call cyg_sntp_start()).

**Warning: timestamp wrap around**

The timestamp in the NTP packet is a 32bit integer which represents the number of seconds after 00:00 01/01/1900. This 32bit number will wrap around at 06:28:16 Feb 7 2036. At this point in time, the eCos time will jump back to around 00:00:00 Jan 1 1900 when the next NTP packet is received.

YOU HAVE BEEN WARNED!

**The SNTP test program**

The SNTP package contains a simple test program. Testing an SNTP client is not easy, so the test program should be considered as more a proof of concept. It shows that an NTP packet has been received, and is accurate to within a few days.

The test program starts the network interfaces using the standard call. It then starts the SNTP thread. A loop is then entered printing the current system time every second for two minutes. When the client receives an NTP packet the time will jump from 1970 to hopefully the present day. Once the two minutes have expired, two simple tests are made. If the time is still less than 5 minutes since 00:00:00 01/01/1970 the test fails. This indicates no NTP messages have been received. Check that the server is actually sending packet, using the correct port (123), correct IPv6 multicast address, and at a sufficiently frequent rate that the target has a chance to receive a message within the 2 minute interval. If all this is correct, assume the target is broken.

The second test is that the current system time is compared with the build time as reported by the CPP macro __DATE__. If the build date is in the future relative to the system time, the test fails. If the build date is more than 90 days in the past relative to the system time the test also fails. If such failures are seen, use walk-clock time to verify the time printed during the test. If this seems correct check the build date for the test. This is printed at startup. If all else fails check that the computer used to build the test has the correct time.

If SNTP unicast mode is enabled, the above tests are run twice. The first time, the SNTP client is configured with NTP server addresses from DHCP. The second time, unicast mode is disabled and only multicasts are listened for. Note that the unicast test is partially bogus in the sense that any multicast packet received will also make the unicast test pass. To reduce the chance of this happening the test will wait for a sorter time for replies. This is not ideal, but it is the best that can be done with an automated test.
XXVII. Another Tiny HTTP Server for eCos

This package provides an extensible, small footprint, full featured HTTP server for eCos. Many of these features can be disabled via the configuration tool, thus reducing the footprint of the server. The server has been written for the FreeBSD network stack.
Chapter 65. The ATHTTP Server

Features

This ATHTTP implementation provides the following features:

- GET, POST and HEAD Methods
- File system Access
- Callbacks to C functions
- MIME type support
- CGI mechanism through the OBJLOADER package or through a simple tcl interpreter
- Basic and Digest (MD5) Authentication
- Directory Listing
- Extendable Internal Resources

Ecos tables are used extensively through the server to provide a high degree of customization.

Starting the server

In order to start the web server, the user needs to call the function:

cyg_httpd_start();

in the application code. The server initialization code spawns a new thread which calls init_all_network_interfaces() to initialize the TCP/IP stack and then starts the daemon. The function is safe to call multiple times.

MIME types

The server has an internal table with all the recognized mime types. Each time a file or an internal resource is sent out by the server, its extension is searched in this table and if a match is found, the associated MIME type is then sent out in the header. The server already provides entries for the following standard file extensions: 'html', 'htm', 'gif', 'jpg', 'css', 'js', 'png' and the user is responsible for adding any further entry. The syntax for adding an entry is the following:

CYG_HTTPD_MIME_TABLE_ENTRY(entry_label, extension_string, mime_type_string);

entry table : an identifier unique to this entry
extension string : a string containing the extension for this entry
type string : the mime string. The strings for many more mime types is included in a file in the "doc" directory.

The following is an example of how to add the Adobe Portable Document Format pdf MIME type to the table:
CYG_HTTPD_MIME_TABLE_ENTRY(hal_pdf_entry, "pdf", "application/pdf");

**MIME Types for Chunked Frames**

For chunked frames, which are generally used inside C language callbacks, there is no file name to match an extension to, and thus the extension to be used must be passed in the `cyg_httpd_start_chunked()` call. The server will then scan the MIME table to find a MIME type to match the extension. For example, to start a chunked transfer of an `html` file, the following call is used:

```c
cyg_httpd_start_chunked("html");
```

In any event, it is the responsibility of the user to make sure that a match to all used extensions is found in the table search. Failing this, the default MIME type specified in the `CYGDAT_NET_ATHTTPD_DEFAULT_MIME_TYPE` string is returned.

**C language callback functions**

The server allows the association of particular URLs to C language callback functions. eCos tables are used to define the association between a URL and its corresponding callback. The syntax of the macro to add callback entries to the table is:

```c
CYG_HTTPD_HANDLER_TABLE_ENTRY(entry_label, url_string, callback);
```

- **entry table**: an identifier unique to this entry.
- **url_string**: a string with the extension url that will be appended to the default directory.
- **callback**: a function with a prototype:

  ```c
  cyg_int32 callback_function(CYG_HTTPD_STATE*);
  ```

  Return value is ignored - just return 0.

**CYG_HTTPD_STATE** is a pointer to a structure that contains, among others, a buffer (outbuffer) that can be used to send data out. The definitions of the structure is in http.h.

The following is an example of how to add a callback to a function `myForm()` whenever the URL `/myform.cgi` is requested:

```c
CYG_HTTPD_HANDLER_TABLE_ENTRY(hal_cb_entry, "/myform.cgi", myForm);
```

and somewhere in the source tree there is a function:

```c
int myForm(CYG_HTTPD_STATE* p)
{
    cyg_httpd_start_chunked("html");
    strcpy(p->outbuffer, "eCos Web Server");
    cyg_httpd_write_chunked(p->outbuffer, strlen(p->outbuffer));
    cyg_httpd_end_chunked();
}
```
This function also shows the correct method of using the chunked frames API inside a C language callback and also shows the use of outbuffer to collect data to send out.

Chunked frames are useful when the size of the frame is not known upfront. In this case it possible to send a response in chunks of various sizes, and terminate it with a null chunk (See RFC 2616 for details). To use chunked frames, the `cyg_httpd_start_chunked()` function is used. The prototype is the following:

```c
ssize_t cyg_httpd_start_chunked(char *);
```

The only parameter is the `extension` to use in the search for the MIME type. For most files this will be "html" or "htm" and it will be searched in the MIME table for an appropriate MIME type that will be sent along in the header. The function returns the number of bytes sent out.

The chunked frame must be terminated by a call to `cyg_httpd_end_chunked()`:

```c
void cyg_httpd_end_chunked();
```

In between these two calls, the user can call the function `cyg_httpd_write_chunked()` to send out data any number of times. It is important that `cyg_httpd_write_chunked()` be the only function used to send data out for chunked frames. This guarantees that proper formatting of the response is respected. The prototype for the function is:

```c
ssize_t cyg_httpd_write_chunked(char* p, int len);
```

The `char*` points to the data to send out, the `int` is the length of the data to send.

In the case in which the size of the data is known upfront, the callback can instead create the header with a call to `cyg_httpd_create_std_header()` with the following prototype:

```c
void cyg_httpd_create_std_header(char *ext, int len);
```

extension : the extension used in the search of the MIME type
len : length of the data to send out

and use `cyg_httpd_write()` to send data out to the client. The prototype of `cyg_httpd_write()` is the same as `cyg_httpd_write_chunked()`

### CGI

The web server allows writing of pseudo-CGI programs. This is helpful in order to modify the functionality of the server without having to recompile it and reflash it.

One way to implement CGI is, of course, the C language callback mechanism described above: This assumes, of course, that all the callbacks are written by compile time and cannot be modified later on. Another way to perform the same functionality is the use of a library in the form of an object file. These object files reside in the file system and are loaded, executed and unloaded on demand.

Yet a third way is the use of a scripting language. Since full fledged implementation of the most popular scripting languages such as Python or Perl are too large for most embedded systems, a slim down implementation of tcl was chosen for this server. Most of the tcl functionality is still there, and makes writing cgi a lot easier.

In order to limit the footprint of the operating system support for both the objloader and the tcl script for dealing with cgi files can be independently selected out. Tcl support in particular increases the memory requirements considerably.
Chapter 65. The ATHTTP Server

CGI via objloader

In order to use the cgi mechanism the CYGPKG_OBJLOADER must be included when building the operating system. This will enable the proper option in the configuration tool and if selected, the necessary code will be compiled in the eCos kernel. The user will then have to compile the necessary libraries and place them in the file system under a directory defined by CYGDAT_NET_ATHTTPD_SERVEROPT_CGIDIR. When a request is made, the web server checks if the root directory of the requested URL is inside the CYGDAT_NET_ATHTTPD_SERVEROPT_CGIDIR directory. If so, the server assumes that the user requested a cgi file and looks into the directory to see if a library by the same name is present, and if so load it and tries to execute a function inside the library with the following prototype:

```c
void exec_cgi(CYG_HTTPD_STATE *)
```

The pointer `CYG_HTTPD_STATE*` gives access to the socket data: The user will use this pointer to access the 'outbuffer' and use it to copy data to send data out.

When using the OBJLOADER package within the HTTP server a number of functions are automatically added to the externals table of the OBJLOADER package. These functions are likely to be used inside the library and the relocator need to have a pointer to them. In order to add more functions, see the OBJLOADER documentation. The complete list of the functions automatically added is:

- `cyg_httpd_start_chunked()`
- `cyg_httpd_write_chunked()`
- `cyg_httpd_end_chunked()`
- `cyg_httpd_write()`
- `cyg_httpd_find_form_variable()`
- `cyg_httpd_find_ires()`
- `cyg_httpd_send_ires()`
- `diag_printf()`
- `cyg_httpd_format_header()`
- `cyg_httpd_find_mime_string()`

Every time the web client issues a GET or POST request for a file with an extension of '.o' in the /cgi-bin directory (or whatever path the user chooses to hold the libraries) then the library by that name is loaded, run and when the execution is over, it is dumped from memory. The library must be compiled separately, using the same toolchain used to compile the server and then added to the file system.

In order to reduce the footprint of the server, CGI through OBJLOADER can be compiled out by unchecking CYGOPT_NET_ATHTTPD_USE_CGIBIN_OBJLOADER in the configuration tool.

CGI via the simple tcl interpreter

A small tcl interpreter has been added to the web server, and it can be used to write simple cgi scripts. The interpreter is admittedly very minimal, and it is only useful for very simple applications, but it is an excellent starting point for further development.

In order for the scripting language to be useful, it has to access the form variables passed on during the GET or POST request. Because of this, all form variables registered with the CYG_HTTPD_FVAR_TABLE_ENTRY() macro are accessible via tcl. For example, if we have registered a form variable called foo, and during the GET request we are defining foo as being "1":

```tcl
```
GET /myForm.cgi?foo=1

then tcl will be able to access the variable foo as $foo. The data in the body of a POST request is also accessible through the use of the variable $post_data. This is useful if the data is not in "multipart/form-data" and tcl has to perform any type of processing on the data itself.

In order to send back a response to the client a few functions have been added to the interpreter. These functions are:

**start_chunked**

```tcl
start_chunked "extension";
```

"extension" is a string used to search the table of the mime types. For example, to send back to the client an HTML file, we can use: start_chunked "html";

**write_chunked**

```tcl
write_chunked content;
```

content is a string to send back to the client.

**end_chunked**

```tcl
end_chunked;
```

No parameters. Send back an end of frame to the client.

**tcl hello world example**

The following example demonstrates how to send a log file in the file /ram/log to a web client. It replaces newline characters with `<br>` so that it is formatted on the browser correctly.

```tcl
start_chunked "html";

set fp [aio.open "/ram/log" r];
$fp seek 0 end;
set fsize [$fp tell];
$fp seek 0 start;
set data "abcxxx";
set data [$fp read $fsize];
$fp close;
set data [string map {
  
  <br>

} $data];

set datax ""
append datax "<html><body>" $data "</body></html>";

write_chunked $datax;
end_chunked;
```

The above file should exist on a filesystem on the embedded target within its `cgi-bin` directory, for example as /cgi-bin/hello.tcl. Thereafter it may be accessed at the URL `http://TARGET_NAME/cgi-bin/hello.tcl`. 

545
Chapter 65. The ATHTTP Server

Authentication

The server supports both Basic (base64) and Digest (MD5) authentication, although they have not been tested with all clients. In this implementation, the contents of certain directories of the file system can be protected, such that the user will be required to issue a username/password to access the content of the directory.

To protect a directory with a basic authentication, there is a specific macro:

`CYG_HTTPD_AUTH_TABLE_ENTRY(entry, path, domain, un, pw, mode)`

- `entry`: an identifier unique to this entry.
- `path`: the path to the directory whose content must be authenticated before it is sent out.
- `domain`: a domain identifier for this directory.
- `un`: username for authentication.
- `pw`: password for authentication.
- `mode`: `CYG_HTTPD_AUTH_BASIC` for base64 encoding or `CYG_HTTPD_AUTH_DIGEST` for MD5 encoding.

For example, to require basic authentication of the content of directory "/ecos/" with a username of "foo" and password "bar", the following is used:

```
CYG_HTTPD_AUTH_TABLE_ENTRY(hal_domain1_entry, 
    "/ecos/",  "ecos_domain", 
    "foo",  "bar", 
    CYG_HTTPD_AUTH_BASIC);
```

Any request for a file in the directory /ecos/ will now trigger a credential check. These credentials, once provided, are automatically sent by the client for every request within the particular domain.

It must be noticed that the path name set in the macro is relative to the HTML document directory, `CYGDAT_NET_HTTPD_SERVEROPT_HTMLDIR` and it is the first part of the path provided by the client request (including the leading slash).

In order to reduce the footprint of the server, authentication is not enabled by default, and so the option `CYGOPT_NET_ATHTTPD_USE_AUTH` must be used to enable support for basic and digest authentication.

The MD5 digest authentication support is implemented using the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Derivative works with MD5 digest authentication included must be identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. See the file md5.c within this package for license details.

Directory Listing

If the user issues a "GET" request with a URL terminating in a slash, the server will try to locate one of the following index files in the directory, choosing one in the following order:

- index.html
- index.htm
- default.html
- home.html

If any of these files are found, its contents are sent back to the client. If no such file is found the server uses the user-provided index file name (if any is specified with the `CYGDAT_NET_ATHTTPD_ALTERNATE_HOME` setting. Failing all this a directory listing is sent.
Trailing slash redirection for directory names is supported.

In order to reduce the footprint of the server, directory listing can be disabled by unchecking CYGOPT_NET_ATHTTPD_USE_DIRLIST. The savings are substantial since directory listing also makes use of a few internal resources (gif files) which are also compiled out.

**Form Variables**

The server will automatically try to parse form variables when a form is submitted in the following cases:

- In a GET request, when the URL is followed by a question mark sign
- In a POST request, when the the 'Content-Type' header line is set to 'application/x-www-form-urlencoded'

The variable names to look for during the parsing are held in an eCos table. In order to take advantage of this feature, the user first adds the variable names to the table, which also requires providing a buffer where the parsed value will eventually be stored. The values will then be available in the buffers during the processing of the request, presumably in the body of a c language callback or CGI script.

For example, if the user wants two form variables, "foo" and "bar", to be parsed automatically, those variable names must be added to the table with the following macro:

```c
CYG_HTTPD_FVAR_TABLE_ENTRY(entry, name, buffp, bufflen)
```

- entry : an identifier unique to this entry.
- name : name of the form variable
- buffp : a pointer to a buffer of characters where to store the value of the form variable.
- bufflen : The length of the buffer. Must include a trailing string terminator.

or, in the specific instance mentioned above:

```c
#define HTML_VAR_LEN 20
char var_foo[HTML_VAR_LEN];
char var_bar[HTML_VAR_LEN];
CYG_HTTPD_FVAR_TABLE_ENTRY(hal_form_entry_foo, "foo", var_foo, HTML_VAR_LEN);
CYG_HTTPD_FVAR_TABLE_ENTRY(hal_form_entry_bar, "bar", var_bar, HTML_VAR_LEN);
```

and after the GET or POST submissions, the list will contain the value for "foo" and "bar" (if they were found in the form data.) It is the responsibility of the user to make sure that the buffer is large enough to hold all the data parsed (including the string terminator). The parser will write only up to the length of the buffer minus one (the last being the terminator) and discard any additional data.

The values parsed are likely going to be used in c language callback, or in CGI files. In a c language callback the user can directly access the pointers of individual variables for further processing, keeping in mind that the parsing always result in a string of characters to be produced, and any conversion (e.g. from strings to integer) must be performed within the callback. In a TCL script the user can just access a variable by its name. For example, in the case of the variables ‘foo’ and ‘bar’ shown above, it is possible to do something like ‘write_chunked “You wrote $foo”.’ The data that was sent in the body of a POST request is accessible in through a variable called ‘post_data’. In CGI functions implemented using the objloader the pointers to the variables cannot be accessed directly, since the library will likely not know their location in memory. The proper way to access them is by using the cyg_httpd_find_form_variable() function from within the library:

```c
char* cyg_httpd_find_form_variable(char* name)
```
name : name of the form variable to look up

returns a pointer to the buffer, or 0 if the variable was not found.

When using the OBJLOADER package within the web server, an entry for the cyg_httpd_find_form_variable() function is automatically added to the externals table the OBJLOADER for relocation. See the OBLOADER paragraph of the ATHTTP user’s guide for the full list of the exported functions.

In order to avoid stale data, all the buffers in the table are cleared before running the parser and thus any variable in the list that was not assigned a new value during the request will be an empty string.

Internal Resources

When the server does not use a file system the user must be responsible to provide a C language callback function for each URL that will be requested by the client. This means locating the data and sending it out using either cyg_httpd_write() or cyg_httpd_write_chunked().

In order to simplify this process the server allows registering any number of URLs as internal resources, by providing the URL name, the pointer to the resource data and its size. When a URL is requested the server will look it up among all internal resources, and if found, it will send out the resource.

Internal resource can also be used along with a file system. In this case the file system is searched first, and if a file is found, it is sent. If a file is not found, the internal resources are searched and if a match if found it is sent.

The drawback of this approach is, of course, that all these resources are going to add to the size of the operating system image, and thus it should be used only when memory is not a major constraint of the design.

As always, to provide this type of customization, ecos tables are used. The format for adding a new resource to the internal table is the following:

CYG_HTTPD_IRES_TABLE_ENTRY(entry, name, buffp, len)

entry : an identifier unique to this entry.
name : name of the URL including leading ‘/’
buffp : a pointer to a buffer of characters where to store the value of the form variable.
len : size of the array

As an example, if the user wants to provide his own web page by hardcoding it in the application code, here is how he would do it:

#define MY_OWN_HOME_PAGE "eCos RTOS"
CYG_HTTPD_IRES_TABLE_ENTRY(cyg_httpd_ires_home, 
"/index.html", 
MY_OWN_HOME_PAGE, 
9);

The extension of the file name determines the MIME type to be used for internal resources.

When using directory listing you are implicitly making use of internal resources. The small icons that appear to the left of file names and directories are internal resources. Unchecking CYGOPT_NET_HTTP_USE_DIRLIST will prevent the addition of these files.

In order to use internal resources, a generic file must first be turned into a c language array, which is then compiled in the application code. To create this array you can use the tcl script that comes with the ecos distribution at packages/fs/rom/current/support/file2.tcl.
XXVIII. Memory Allocation
Chapter 66. eCos Memory Pools

eCos Memory pools

There are three sorts of memory pools. A variable size memory pool is for allocating blocks of any size. A fixed size memory pool, has the block size specified when the pool is created and only provides blocks of that size. Both of these pools must be explicitly created. The third type is the traditional heap which can be accessed using malloc and friends.

Variable Size Allocation Pools

Name

cyg_mempool_var_create, cyg_mempool_var_delete, cyg_mempool_var_alloc,
cyg_mempool_var_timed_alloc, cyg_mempool_var_try_alloc,
cyg_mempool_var_free, cyg_mempool_var_waiting, cyg_mempool_var_get_info

Synopsis

#include <cyg/kernel/kapi.h>

void cyg_mempool_var_create(void* base, cyg_uint32 size, cyg_handle_t* varpool, cyg_mempool_var* var);
void cyg_mempool_var_delete(cyg_handle_t varpool);
void* cyg_mempool_var_alloc(cyg_handle_t varpool, cyg_uint32 size);
void* cyg_mempool_var_timed_alloc(cyg_handle_t varpool, cyg_uint32 size, cyg_tick_count_t abstime);
void* cyg_mempool_var_try_alloc(cyg_handle_t varpool, cyg_uint32 size);
void cyg_mempool_var_free(cyg_handle_t varpool, void* p);
cyg_bool_t cyg_mempool_var_waiting(cyg_handle_t varpool);
void cyg_mempool_var_get_info(cyg_handle_t varpool, cyg_mempool_info* info);

Description

The variable size memory pool functions are used for allocating blocks of any size. Before memory can be allocated the pool must first be created by calling cyg_mempool_var_create(). The parameter base is a point to the bottom of the memory area to be used by the pool and size is the size of the memory area in bytes. It also takes a pointer to a cyg_mempool_var data structure which is typically statically allocated, and may be part of a larger data structure. It should be noted that some memory is take from the pool for book keeping purposes. If a memory pool is no longer required and there are not threads waiting to allocate memory from it, it can be destroyed with cyg_mempool_var_delete().

Memory can be allocated from the pool using a number of functions. They all take the parameters varpool which indicates which pool should be used and the size which indicates who big a memory area should be allocated. cyg_mempool_var_alloc() will block until the memory becomes available. cyg_mempool_tryalloc() will
try not block if no memory is available and will return NULL. Otherwise a pointer to the allocated memory will be returned. \texttt{cyg_mempool_var_timed_alloc()} will block if memory is not available and wait for memory to become available until the \texttt{abstime} is reached. It will then return NULL.

Allocated memory can be freed using the function \texttt{cyg_mempool_var_free()}. Lastly it is possible to query information about the pool using the function \texttt{cyg_mempool_var_get_info()}. This takes a pointer to the structure \texttt{cyg_mempool_info} which is:

```c
typedef struct {
    cyg_int32 totalmem;
    cyg_int32 freemem;
    void *base;
    cyg_int32 size;
    cyg_int32 blocksize;
    cyg_int32 maxfree;
} cyg_mempool_info;
```
**Fixed Size Allocation Pools**

**Name**

cyg_mempool_fix_create, cyg_mempool_fix_delete, cyg_mempool_fix_alloc,
cyg_mempool_fix_timed_alloc, cyg_mempool_fix_try_alloc,
cyg_mempool_fix_free, cyg_mempool_fix_waiting, cyg_mempool_fix_get_info

**Synopsis**

```c
#include <cyg/kernel/kapi.h>

void cyg_mempool_fix_create(void* base, cyg_uint32 size, cyg_uint32 blocksize,
cyg_handle_t* fixpool, cyg_mempool_fix* fix);
void cyg_mempool_fix_delete(cyg_handle_t fixpool);
void* cyg_mempool_fix_alloc(cyg_handle_t fixpool);
void* cyg_mempool_fix_timed_alloc(cyg_handle_t fixpool, cyg_tick_count_t abstime);
void* cyg_mempool_fix_try_alloc(cyg_handle_t fixpool);
void cyg_mempool_fix_free(cyg_handle_t fixpool, void* p);
cyg_bool_t cyg_mempool_fix_waiting(cyg_handle_t fixpool);
void cyg_mempool_fix_get_info(cyg_handle_t fixpool, cyg_mempool_info* info);
```

**Description**

The fixed size memory pool functions are used for allocating blocks of the same size. The allocation and free functions are more efficient than the variable size pools, but are naturally limited to being only able to allocate blocks of a sized size. Before memory can be allocated the pool must first be created by calling `cyg_mempool_fix_create()` . The parameter `base` is a point to the bottom of the memory area to be used by the pool and `size` is the size of the memory area in bytes. `blocksize` indicates the size of each allocation in bytes. The function also takes a pointer to a `cyg_mempool_fix` data structure which is typically statically allocated, and may be part of a larger data structure. It should be noted that some memory is take from the pool for book keeping purposes. If a memory pool is no longer required and there are not threads waiting to allocate memory from it, it can be destroyed with `cyg_mempool_fix_delete()`.

Memory can be allocated from the pool using a number of functions. They all take the parameter `fixpool` which indicates which pool should be used. `cyg_mempool_fix_alloc()` will block until the memory becomes available. `cyg_mempool_fix_alloc()` will try not block if no memory is available and will return NULL. Otherwise a pointer to the allocated memory will be returned. `cyg_mempool_fix_timed_alloc()` will block if memory is not available and wait for memory to become available until the time `abstime` is reached. It will then return NULL.

Allocated memory can be freed using the function `cyg_mempool_fix_free()`.

Lastly it is possible to query information about the pool using the function `cyg_mempool_fix_get_info()`. This takes a pointer to the structure `cyg_mempool_info` which is:

```c
typedef struct {
    cyg_int32 totalmem;
    cyg_int32 freemem;
    void* base;
```
Fixed Size Allocation Pools

cyg_int32 size;
cyg_int32 blocksize;
cyg_int32 maxfree;
} cyg_mempool_info;
stdlib malloc Pools

Name

malloc, calloc, free, realloc, mallinfo — stdlib malloc pool

Synopsis

#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
struct mallinfo mallinfo(void);

Description

eCos provides the standard library functions used for allocating memory from the heap. malloc() allocates a block of memory of size bytes. calloc() performs the same, but also sets the memory to zero. The function free() returns a block to the pool. realloc resizes a block of memory. Lastly, mallinfo() returns information about the heap, as described by the structure mallinfo:

struct mallinfo {
    int arena;  /* total size of memory arena */
    int ordblks; /* number of ordinary memory blocks */
    int smblks; /* number of small memory blocks */
    int hblks;  /* number of mmapped regions */
    int hblkhld; /* total space in mmapped regions */
    int usmblks; /* space used by small memory blocks */
    int fsmblks; /* space available for small memory blocks */
    int uordblks; /* space used by ordinary memory blocks */
    int fordblks; /* space free for ordinary blocks */
    int keepcost; /* top-most, releasable (via malloc_trim) space */
    int maxfree; /* (NON-STANDARD EXTENSION) size of largest free block */
};
stdlib malloc Pools
XXIX. CRC Algorithms

The CRC package provides implementation of CRC algorithms. This includes the POSIX CRC calculation which produces the same result as the cksum command on Linux, another 32 bit CRC by Gary S. Brown and a 16bit CRC. The CRC used for Ethernet FCS is also implemented.
Chapter 67. CRC Functions

CRC API

The package implements a number of CRC functions as described below. The API to these functions is in the include file cyg/crc/crc.h.

**cyg_posix_crc32**

This function implements a 32 bit CRC which is compliant to the POSIX 1008.2 Standard. This is the same as the Linux cksum program.

```
cyg_uint32 cyg_posix_crc32(unsigned char *s, int len);
```

The CRC calculation is run over the data pointed to by `s`, of length `len`. The CRC is returned as an unsigned long.

**cyg_crc32**

These functions implement a 32 bit CRC by Gary S. Brown. They use the polynomial \(X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X^1+X^0\).

```
cyg_uint32 cyg_crc32(unsigned char *s, int len);
cyg_uint32 cyg_crc32_accumulate(cyg_uint32 crc, unsigned char *s, int len);
```

The CRC calculation is run over the data pointed to by `s`, of length `len`. The CRC is returned as an unsigned long.

The CRC can be calculated over data separated into multiple buffers by using the function `cyg_crc32_accumulate()`. The parameter `crc` should be the result from the previous CRC calculation.

**cyg_ether_crc32**

These functions implement the 32 bit CRC used by the Ethernet FCS word.

```
cyg_uint32 cyg_ether_crc32(unsigned char *s, int len);
cyg_uint32 cyg_ether_crc32_accumulate(cyg_uint32 crc, unsigned char *s, int len);
```

The CRC calculation is run over the data pointed to by `s`, of length `len`. The CRC is returned as an unsigned long.

The CRC can be calculated over data separated into multiple buffers by using the function `cyg_ether_crc32_accumulate()`. The parameter `crc` should be the result from the previous CRC calculation.

**cyg_crc16**

This function implements a 16 bit CRC. It uses the polynomial \(x^{16}+x^{12}+x^5+1\).

```
cyg_uint16 cyg_crc16(unsigned char *s, int len);
```
Chapter 67. CRC Functions

The CRC calculation is run over the data pointed to by $s$, of length $\text{len}$. The CRC is returned as an unsigned short.
XXX. CPU load measurements

The cpuload package provides a way to estimate the cpuload. It gives an estimated percentage load for the last 100 milliseconds, 1 second and 10 seconds.
Chapter 68. CPU Load Measurements

CPU Load API

The package allows the CPU load to be estimated. The measurement code must first be calibrated to the target it is running on. Once this has been performed the measurement process can be started. This is a continuous process, so always providing the most up to data measurements. The process can be stopped at any time if required. Once the process is active, the results can be retrieved.

Note that if the target/processor performs any power saving actions, such as reducing the clock speed, or halting until the next interrupt etc., these will interfere with the CPU load measurement. Under these conditions the measurement results are undefined. The synthetic target is one such system. See the implementation details at the foot of this page for further information.

SMP systems are not supported, only uniprocessor system.

The API for load measuring functions can be found in the file cyg/cpuload/cpuload.h.

**cyg_cpuload_calibrate**

This function is used to calibrate the cpu load measurement code. It makes a measurement to determine the CPU properties while idle.

```c
void cyg_cpuload_calibrate(cyg_uint32 *calibration);
```

The function returns the calibration value at the location pointed to by `calibration`.

This function is quite unusual. For it to work correctly a few conditions must be met. The function makes use of the two highest thread priorities. No other threads must be using these priorities while the function is being used. The kernel scheduler must be started and not disabled. The function takes 100ms to complete during which time no other threads will be run.

**cyg_cpuload_create**

This function starts the CPU load measurements.

```c
void cyg_cpuload_create(cyg_cpuload_t *cpuload, 
                        cyg_uint32 calibrate, 
                        cyg_handle_t *handle);
```

The measurement process is started and a handle to it is returned in `*handle`. This handle is used to access the results and the stop the measurement process.

**cyg_cpuload_delete**

This function stops the measurement process.

```c
void cyg_cpuload_delete(cyg_handle_t handle);
```

`handle` should be the value returned by the create function.
Chapter 68. CPU Load Measurements

**cyg_cpuload_get**

This function returns the latest measurements.

```c
void cyg_cpuload_get(cyg_handle_t handle,
    cyg_uint32 *average_point1s,
    cyg_uint32 *average_1s,
    cyg_uint32 *average_10s);
```

*handle* should be the value returned by the create function. The load measurements for the last 100ms, 1s and 10s are returned in *average_point1s*, *average_1s* and *average_10s* respectively.

**Implementation details**

This section gives a few details of how the measurements are made. This should help to understand what the results mean.

When there are no other threads runnable, eCos will execute the idle thread. This thread is always runnable and uses the lowest thread priority. The idle thread does little. It is an endless loop which increments the variable, *idle_thread_loops* and executes the macro `HAL_IDLE_THREAD_ACTION`. The cpu load measurement code makes use of the variable. It periodically examines the value of the variable and sees how much it has changed. The idler the system, the more it will have incremented. From this it is simple to determine the load of the system.

The function `cyg_cpuload_calibrate` executes the idle thread for 100ms to determine how much *idle_thread_loops* is incremented on a system idle for 100ms. `cyg_cpuload_create` starts an alarm which every 100ms calls an alarm function. This function looks at the difference in *idle_thread_loops* since the last invocation of the alarm function and so calculated how idle or busy the system has been. The structure `cyg_cpuload` is updated during the alarm functions with the new results. The 100ms result is simply the result from the last measurement period. A simple filter is used to average the load over a period of time, namely 1s and 10s. Due to rounding errors, the 1s and 10s value will probably never reach 100% on a fully loaded system, but 99% is often seen.

As stated above, clever power management code will interfere with these measurements. The basic assumption is that the idle thread will be executed un-hindered and under the same conditions as when the calibration function was executed. If the CPU clock rate is reduced, the idle thread counter will be incremented less and so the CPU load measurements will give values too high. If the CPU is halted entirely, 100% cpu load will be measured.
XXXI. gprof Profiling Support
Profiling

Name

CYGPKG_PROFILE_GPROF — eCos Support for the gprof profiling tool

Description

The GNU gprof tool provides profiling support. After a test run it can be used to find where the application spent most of its time, and that information can then be used to guide optimization effort. Typical gprof output will look something like this:

```
Each sample counts as 0.003003 seconds.

% cumulative self self total
  time seconds seconds calls us/call us/call name
14.15 1.45 1.45 120000 12.05 12.05 Proc_7
11.55 2.63 1.18 120000 9.84 9.84 Func_1
  8.04 3.45 0.82 main
7.60 4.22 0.78 40000 19.41 86.75 Proc_1
  6.89 4.93 0.70 40000 17.60 28.99 Proc_6
  6.77 5.62 0.69 40000 17.31 27.14 Func_2
  6.62 6.30 0.68 40000 16.92 16.92 Proc_8
  5.94 6.90 0.61 strcmp
  5.58 7.47 0.57 40000 14.26 26.31 Proc_3
  5.01 7.99 0.51 40000 12.79 12.79 Proc_4
  4.46 8.44 0.46 40000 11.39 11.39 Proc_5
  3.68 8.82 0.38 40000  9.40  9.40 Proc_5
  3.32 9.16 0.34 40000  8.48  8.48 Proc_2
...
```

This output is known as the flat profile. The data is obtained by having a hardware timer generate regular interrupts. The interrupt handler stores the program counter of the interrupted code. gprof performs a statistical analysis of the resulting data and works out where the time was spent.

gprof can also provide information about the call graph, for example:

```
index % time self children called name
...
[2]  34.0 0.78 2.69 40000/40000 main [1]

0.78 0.46 40000/40000 Proc_6 [5]
0.57 0.48 40000/40000 Proc_3 [7]
0.48 0.00 40000/120000 Proc_7 [3]
```

This shows that function Proc_1 was called only from main, and Proc_1 in turn called three other functions. Callgraph information is obtained only if the application code is compiled with the -pg option. This causes the compiler to insert extra code into each compiled function, specifically a call to mcount, and the implementation of mcount stores away the data for subsequent processing by gprof.
Profiling

Caution

There are a number of reasons why the output will not be 100% accurate. Collecting the flat profile typically involves timer interrupts so any code that runs with interrupts disabled will not appear. The current host-side gprof implementation maps program counter values onto symbols using a bin mechanism. When a bin spans the end of one function and the start of the next gprof may report the wrong function. This is especially likely on architectures with single-byte instructions such as an x86. When examining gprof output it may prove useful to look at a linker map or program disassembly.

The eCos profiling package requires some additional support from the HAL packages, and this may not be available on all platforms:

1. There must be an implementation of the profiling timer. Typically this is provided by the variant or platform HAL using one of the hardware timers. If there is no implementation then the configuration tools will report an unresolved conflict related to CYGINT_PROFILE_HAL_TIMER and profiling is not possible. Some implementations overload the system clock, which means that profiling is only possible in configurations containing the eCos kernel and CYGVAR_KERNEL_COUNTERS_CLOCK.

2. There should be a hardware-specific implementation of mcount, which in turn will call the generic functionality provided by this package. It is still possible to do some profiling without mcount but the resulting data will be less useful. To check whether or not mcount is available, look at the current value of the CDL interface CYGINT_PROFILE_HAL_MCOUNT in the graphical configuration tool or in an ecos.ecc save file.

This document only describes the eCos profiling support. Full details of gprof functionality and output formats can be found in the gprof documentation. However it should be noted that that documentation describes some functionality which cannot be implemented using current versions of the gcc compiler: the section on annotated source listings is not relevant, and neither are associated command line options like -A and -y.

Building Applications for Profiling

To perform application profiling the gprof package CYGPKG_PROFILE_GPROF must first be added to the eCos configuration. On the command line this can be achieved using:

$ ecosconfig add profile_gprof
$ ecosconfig tree
$ make

Alternatively the same steps can be performed using the graphical configuration tool.

If the HAL packages implement mcount for the target platform then usually application code should be compiled with -pg. Optionally eCos itself can also be compiled with this option by modifying the configuration option CYGBLD_GLOBAL_CFLAGS. Compiling with -pg is optional but gives more complete profiling data.

Note: The profiling package itself must not be compiled with -pg because that could lead to infinite recursion when doing mcount processing. This is handled automatically by the package's CDL.

Profiling does not happen automatically. Instead it must be started explicitly by the application, using a call to profile_on. A typical example would be:

#include <pkgconf/system.h>
#ifdef CYGPKG_PROFILE_GPROF

#include <pkgconf/system.h>
#ifdef CYGPKG_PROFILE_GPROF

The `profile_on` takes four arguments:

- **Start address**
- **End address**

These specify the range of addresses that will be profiled. Usually profiling should cover the entire application. On most targets the linker script will export symbols `_stext` and `_etext` corresponding to the beginning and end of code, so these can be used as the addresses. It is possible to perform profiling on a subset of the code if that code is located contiguously in memory.

- **Bucket size**

`profile_on` divides the range of addresses into a number of buckets of this size. It then allocates a single array of 16-bit counters with one entry for each bucket. When the profiling timer interrupts the interrupt handler will examine the program counter of the interrupted code and, assuming it is within the range of valid addresses, find the containing bucket and increment the appropriate counter.

The size of the array counters is determined by the range of addresses being profiled and by the bucket size. For a bucket size of 16, one counter is needed for every 16 bytes of code. For an application with say 512K of code that means dynamically allocating a 64K array. If the target hardware is low on memory then this may be unacceptable, and the requirements can be reduced by increasing the bucket size. However this will affect the accuracy of the results and gprof is more likely to report the wrong function. It also increases the risk of a counter overflow.

For the sake of run-time efficiency the bucket size must be a power of 2, and it will be adjusted if necessary.

- **Time interval**

The final argument specifies the interval between profile timer interrupts, in units of microseconds. Increasing the interrupt frequency gives more accurate profiling results, but at the cost of higher run-time overheads and a greater risk of a counter overflow. The HAL package may modify this interval because of hardware restrictions, and the generated profile data will contain the actual interval that was used. Usually it is a good idea to use an interval that is not a simple fraction of the system clock, typically 10000 microseconds. Otherwise there is a risk that the profiling timer will disproportionally sample code that runs only in response to the system clock.

`profile_on` can be invoked multiple times, and on subsequent invocations, it will delete profiling data and allocate a fresh profiling range.

Profiling can be turned off using the function `profile_off`:

```c
void profile_off(void);
```
Profiling

This will also reset any existing profile data.

If the eCos configuration includes a TCP/IP stack and if a tftp daemon will be used to extract the data from the target then the call to `profile_on` should happen after the network is up. `profile_on` will attempt to start a tftp daemon thread, and this will fail if networking has not yet been enabled.

```c
int main(int argc, char** argv)
{
    ...
    init_all_network_interfaces();
    ...
#ifdef CYGPKG_PROFILE_GPROF
    {
        extern char _stext[], _etext[];
        profile_on(_stext, _etext, 16, 3000);
    }
#endif
    ...
}
```

The application can then be linked and run as usual.

When `gprof` is used for native development rather than for embedded targets the profiling data will automatically be written out to a file `gmon.out` when the program exits. This is not possible on an embedded target because the code has no direct access to the host’s file system. Instead the `gmon.out` file has to be extracted from the target as described below. `gprof` can then be invoked normally:

```
$ gprof dhrystone
Flat profile:

Each sample counts as 0.003003 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.15</td>
<td>1.45</td>
<td>1.45</td>
<td>120000</td>
<td>12.05</td>
</tr>
<tr>
<td>11.55</td>
<td>2.63</td>
<td>1.18</td>
<td>120000</td>
<td>9.84</td>
</tr>
<tr>
<td>8.04</td>
<td>3.45</td>
<td>0.82</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...
```

If `gmon.out` does not contain call graph data, either because `mcount` is not supported or because this functionality was explicitly disabled, then the `-no-graph` must be used.

```
$ gprof --no-graph dhrystone
Flat profile:
```
Profiling

Each sample counts as 0.003003 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.15</td>
<td>1.45</td>
<td>1.45</td>
<td>Proc_7</td>
<td></td>
</tr>
<tr>
<td>11.55</td>
<td>2.63</td>
<td>1.18</td>
<td>Func_1</td>
<td></td>
</tr>
<tr>
<td>8.04</td>
<td>3.45</td>
<td>0.82</td>
<td>main</td>
<td></td>
</tr>
</tbody>
</table>

Extracting the Data

By default gprof expects to find the profiling data in a file gmon.out in the current directory. This package provides two ways of extracting data: a gdb macro or tftp transfers. Using tftp is faster but requires a TCP/IP stack on the target. It also consumes some additional target-side resources, including an extra tftp daemon thread and its stack. The gdb macro can be used even when the eCos configuration does not include a TCP/IP stack. However it is much slower, typically taking tens of seconds to retrieve all the data for a non-trivial application.

The gdb macro is called gprof_dump, and can be found in the file gprof.gdb in the host subdirectory of this package. A typical way of using this macro is:

```
(gdb) source <repo>/services/profile/gprof/<version>/host/gprof.gdb
(gdb) gprof_dump
```

This macro can be used any time after the call to profile_on. It will store the profiling data accumulated so far to the file gmon.out in the current directory, and then reset all counts. gprof uses only a 16 bit counter for every bucket of code. These counters can easily saturate if the profiling run goes on for a long time, or if the application code spends nearly all its time in just a few tight inner loops. The counters will not actually wrap around back to zero, instead they will stick at 0xFFFF, but this will still affect the accuracy of the gprof output. Hence it is desirable to reset the counters once the profiling data has been extracted.

The file gprof.gdb contains two other macros which may prove useful. gprof_fetch extracts the profiling data and generates the file gmon.out, but does not reset the counters. gprof_reset only resets the counters, without extracting the data or overwriting gmon.out.

If the configuration includes a TCP/IP stack then the profiling data can be extracted using tftp instead. There are two relevant configuration options. CYGPKG_PROFILE_TFTP controls whether or not tftp is supported. It is enabled by default if the configuration includes a TCP/IP stack, but can be disabled to save target-side resources. CYGNUM_PROFILE_TFTP_PORT controls the UDP port which will be used. This port cannot be shared with other tftp daemons. If neither application code nor any other package (for example the gcov test coverage package) provides a tftp service then the default port can be used. Otherwise it will be necessary to assign unique ports to each daemon.

If enabled the tftp daemon will be started automatically by profile_on. This should only happen once the network is up and running, typically after the call to init_all_network_interfaces.

The data can then be retrieved using a standard tftp client. There are a number of such clients available with very different interfaces, but a typical session might look something like this:

```
$ tftp
tftp> connect 10.1.1.134
tftp> binary
tftp> get gmon.out
Received 64712 bytes in 0.9 seconds
tftp> quit
```
The address 10.1.1.134 should be replaced with the target’s IP address. Extracting the profiling data by tftp will automatically reset the counters.

**Configuration Options**

This package contains a number of configuration options. Two of these, CYGPKG_PROFILE_TFTP and CYGNUM_PROFILE_TFTP_PORT, related to support for tftp transfers and have already been described.

Support for collecting the call graph data via mcount is optional and can be controlled via CYGPKG_PROFILE_CALLGRAPH. This option will only be active if the HAL provides the underlying mcount support and implements CYGINTEGR_PROFILE_HAL_MCOUNT. The call graph data allows gprof to produce more useful output, but at the cost of extra run-time and memory overheads. If this option is disabled then the -pg compiler flag should not be used.

If CYGPKG_PROFILE_CALLGRAPH is enabled then there are two further options which can be used to control memory requirements. Collecting the data requires two blocks of memory, a simple hash table and an array of arc records. The mcount code uses the program counter address to index into the hash table, giving the first element of a singly linked list. The array of arc records contains the various linked lists for each hash slot. The required number of arc records depends on the number of function calls in the application. For example if a function Proc_7 is called from three different places in the application then three arc records will be needed.

CYGNUM_PROFILE_CALLGRAPH_HASH_SHIFT controls the size of the hash table. The default value of 8 means that the program counter is shifted right by eight places to give a hash table index. Hence each hash table slot corresponds to 256 bytes of code, and for an application with say 512K of code profile_on will dynamically allocate an 8K hash table. Increasing the shift size reduces the memory requirement, but means that each hash table slot will correspond to more code and hence mcount will need to traverse a longer linked list of arc records.

CYGNUM_PROFILE_CALLGRAPH_ARC_PERCENTAGE controls how much memory profile_on will allocate for the arc records. This uses a simple heuristic, a percentage of the overall code size. By default the amount of arc record space allocated will be 5% of the code size, so for a 512K executable that requires approximately 26K. This default should suffice for most applications. In exceptional cases it may be insufficient and a diagnostic will be generated when the profiling data is extracted.

**Implementing the HAL Support**

The profiling package requires HAL support: A function hal_enable_profile_timer and an implementation of mcount. The profile timer is required. Typically it will be implemented by the variant or platform HAL using a spare hardware timer, and that HAL package will also implement the CDL interface CYGINTEGR_PROFILE_HAL_TIMER. Support for mcount is optional but very desirable. Typically it will be implemented by the architectural HAL, which will also implement the CDL interface CYGINTEGR_PROFILE_HAL_MCOUNT.

```c
#include <pkgconf/system.h>
#ifdef CYGPKG_PROFILE_GPROF
  #include <cyg/profile/profile.h>
#endif

int hal_enable_profile_timer(int resolution) {
  ...
  return actual_resolution;
}
```
This function takes a single argument, a time interval in microseconds. It should arrange for a timer interrupt to
go off after every interval. The timer VSR or ISR should then determine the program counter of the interrupted
code and register this with the profiling package:

\[
...\nonumber
\text{__profile\_hit(interrupted\_pc);}
...\nonumber
\]

The exact details of how this is achieved, especially obtaining the interrupted PC, are left to the HAL imple-
mentor. The HAL is allowed to modify the requested time interval because of hardware constraints, and should
return the interval that is actually used.

\text{mcount} \text{ can be more difficult. The calls to } \text{mcount} \text{ are generated internally by the compiler and the details}
depend on the target architecture. In fact \text{mcount} \text{ may not use the standard calling conventions at all. Typically}
implementing \text{mcount} \text{ requires looking at the code that is actually generated, and possibly at the sources of the}
appropriate compiler back end.\text{The HAL mcount function should call into the profiling package using standard calling conventions:}

\[
...\nonumber
\text{__profile\_mcount((CYG\_ADDRWORD) caller\_pc, (CYG\_ADDRWORD) callee\_pc);}
...\nonumber
\]

If \text{mcount} \text{ was invoked because } \text{main} \text{ called } \text{Proc\_1} \text{ then the caller pc should be an address inside } \text{main},
typically corresponding to the return location, and the callee pc should be an address inside \text{Proc\_1}, usually
near the start of the function.

For some targets the compiler does additional work, for example automatically allocating a per-function word
of memory to eliminate the need for the hash table. This is too target-specific and hence cannot easily be used
by the generic profiling package.
Profiling
XXXII. eCos Power Management Support
Profiling
Introduction

Name

Introduction — eCos support for Power Management

Introduction

The eCos Power Management package provides a framework for incorporating power management facilities in an embedded application. However its functionality is deliberately limited.

1. The package does not contain any support for controlling the current power mode of any given processor, device or board. Instead it is the responsibility of the appropriate HAL or device driver package to implement such support, by implementing power controllers. The power management package groups these power controllers together and provides an interface for manipulating them.

2. The package does not contain any power management policy support. Specifically, including this package in an application does not by itself ever cause the system to go into low-power mode. Instead it is the responsibility of a separate policy module, provided by higher-level application code or by some other package, to decide when it would be appropriate to switch from one power mode to another. The power management package then provides the mechanisms for making it happen.

Including Power Management

The power management package is never included automatically in an eCos configuration: it is not part of any target specification or of any template. Instead it must be added explicitly to a configuration if the intended application requires power management functionality. When using the command-line ecosconfig tool this can be achieved using a command such as:

$ ecosconfig add power

The generic eCos user documentation should be consulted for more information on how to use the various tools. The functionality provided by the power management package is defined in the header file cyg/power/power.h. This header file can be used by both C and C++ code.

Power Modes

There are four defined modes of operation:

active

The system is fully operational, and power consumption is expected to be high.

idle

There has been little or no activity for a short period of time. It is up to the policy module to determine what constitutes a short period of time, but typically it will be some tenths of a second or some small number of seconds. A possible action when entering idle mode is to reduce the system’s clock speed, thus reducing the power drawn by the cpu.
Note that typically this power mode is not entered automatically whenever the idle thread starts running. Instead it is entered when the policy module discovers that for a certain period of time the system has been spending most of its time in the idle thread. Theoretically it is possible to implement a policy module that would cause a switch to idle mode as soon as the idle thread starts running, but that could result in a great many power mode changes for no immediate benefit.

The system has been idle for a significant period of time, perhaps some tens of seconds. It is desirable to shut down any hardware that is drawing a significant amount of power, for example a screen backlight.

The system is powered down. Power consumption should be minimized. Some special action may be needed before the system comes back up, for example the user may need to press a specific button.

The exact transitions that will happen are decided by the policy module. One policy module might include transitions from active to idle, from idle to sleep, from sleep to off, and from any of idle, sleep or off directly back to active. Another policy module might only use the active and off states, bypassing the intermediate ones.

Power Controllers

The power management package operates primarily on power controllers. The main functionality provided by a power controller is to switch the power mode for some part of the system, for example the lcd display or the cpu. A power controller consists primarily of a function which will be invoked to switch the power mode for the part of the overall system being controlled, plus some auxiliary data. A typical system will include a number of different power controllers:

1. Usually there will be one power controller `power_controller_cpu` associated with the processor or with the target platform, and provided by the corresponding HAL package. It is this controller which is responsible for switching off the system when entering the off mode, which makes it somewhat special: attempting to switch off the cpu before other devices like the lcd display does not make sense because the cpu would no longer be executing any instructions for the latter operation. Therefore this power controller has to be invoked last when switching to a lower-power mode, and similarly when switching back to a higher-power mode it will be invoked first.

   It should be noted that providing power management support is not a hard requirement when porting eCos to a new processor or platform, and many eCos ports predate the availability of power management support. Therefore for any given platform it is distinctly possible that `power_controller_cpu` is not yet provided, and if full power management functionality is desired then the appropriate HAL package would have to be extended first. System developers should examine the relevant HAL documentation and sources to determine what is actually available.

2. Some or all of the device drivers will supply their own power controllers, as part of the device driver package. It is not required that all device drivers provide power controllers. In some cases, especially for devices that are integrated with the processor, `power_controller_cpu` will take care of the integrated devices as a side effect. In other cases the hardware may not provide any functionality that allows power consumption to be controlled. For any given device driver it is also possible that no power controller exists either because it was not required when the driver was written, or because the driver predates the availability of power management. Again the relevant documentation and sources should be consulted for further information.

3. There may be power controllers which are not associated directly with any specific hardware. For example a TCP/IP stack could provide a power controller so that it gets informed when the system has been
reactivated: by looking at the system clock it can determine for how long the system has been switched off; using this information it can then recover from expired dhcp leases, or even to shut down any stream connections that may have become invalid (although arguably the stack should have refused to go to off mode while there were open connections).

**Basic Operation**

By default the Power Management package creates a thread during initialization. It is also possible for the package to be used without such a thread, for example in configurations which do not include a full kernel, and this alternative is described below. When a separate thread is used the stacksize and priority for this thread can be controlled by configuration options CYGNUM_POWER_THREAD_STACKSIZE and CYGNUM_POWER_THREAD_PRIORITY. Typically the thread will just wait on a semaphore internal to the package, and will do nothing until some other part of the system requests a change to the power mode.

At some point the policy module will decide that the system should move into a lower-power mode, for example from active to idle. This is achieved by calling the function `power_set_mode`, provided by the power management package and declared in cyg/power/power.h, with a single argument, `PowerMode_Idle`. This function manipulates some internal state and posts the semaphore, thus waking up the power management thread. Note that the function returns before the mode change has completed, and in fact depending on thread priorities this return may happen before any power controller has been invoked.

When the power management thread wakes up it examines the internal state to figure out what it should be doing. In this case it is supposed to change the global power mode, so it will iterate over all the power controllers requesting each one to switch to the idle mode. It is up to each power controller to handle this request appropriately. Optionally the thread will invoke a callback function after processing each power controller, so that higher-level code such as the policy module can more easily keep track of the actual state of each controller. Once the thread has iterated through all the power controllers it will again wait on the internal semaphore for the next request to arrive.

**Note:** At present the power management thread always runs at a single priority, which defaults to a low priority. A possible future enhancement would be to support two separate priorities. When switching to a lower-powered mode the thread would run at a low priority as before, thus allowing other threads to run and get a chance to cancel this mode change. When switching to a higher-powered mode the thread would run at a high priority. This could be especially important when moving out of the off state: for example it would ensure that all device drivers get a chance to wake up before ordinary application threads get to run again and possibly attempt I/O operations.

Although usually calls to `power_set_mode` will come from just one place in the policy module, this is not a hard requirement. It is possible for multiple threads to call this function, with no need for any synchronization. If the power management thread is in the middle of performing a mode change and a new request comes in, the thread will detect this, abort the current operation, and start iterating through the power controllers again with the new mode. This check happens between every power controller invocation. Usefully this makes it possible for power controllers themselves to manipulate power modes: a power controller is invoked to change mode; for some reason it determines that the new mode is inappropriate; it calls `power_set_mode` to move the system back to another mode; when the power controller returns this event will be detected; the power management thread will abort the current mode change, and start the new one.

In addition to changing the power mode for the system as a whole, individual controllers can be manipulated using the function `power_set_controller_mode`. For example, while the system as a whole might be in active mode certain devices might be kept in sleep mode until they are explicitly activated. It is possible to mix concurrent calls to `power_set_mode` and `power_set_controller_mode`, and when a power controller is invoked
it may use \texttt{power\_set\_controller\_mode} to request further changes to its own or to another controller’s mode as required.

There are some scenarios where the power management package should not use its own thread. One scenario is if the configuration is specifically for a single-threaded application such as RedBoot. Another scenario is if the policy module already involves a separate thread: it may make more sense if the various power management operations are synchronous with respect to the calling thread. The use of a separate thread inside the power management package is controlled by the configuration option \texttt{CYGPKG\_POWER\_THREAD}, which is active only if the kernel package is present and enabled by default.

If no separate power management thread is used then obviously the implementations of \texttt{power\_set\_mode} and \texttt{power\_set\_controller\_mode} will be somewhat different: instead of waking up a separate thread to do the work, these functions will now manipulate the power controllers directly. If the system does still involve multiple threads then only one thread may call \texttt{power\_set\_mode} or \texttt{power\_set\_controller\_mode} at a time: the power management package will not provide any synchronization, that must happen at a higher level. However when a power controller is invoked it can still call these functions as required.
Power Management Information

Name

Obtaining Power Management Information — finding out about the various power controllers in the system

Synopsis

```c
#include <cyg/power/power.h>

extern PowerController __POWER__[], __POWER_END__;
extern PowerController power_controller_cpu;
extern cyg_handle_t power_thread_handle;
PowerMode power_get_mode (void);
PowerMode power_get_desired_mode (void);
PowerMode power_get_controller_mode (PowerController* controller);
PowerMode power_get_controller_desired_mode (PowerController* controller);
const char* power_get_controller_id (PowerController* controller);
```

Accessing Power Controllers

All the power controllers in a system are held in a table, filled in at link-time. The symbols __POWER__ and __POWER_END__ can be used to iterate through this table, for example:

```c
PowerController* controller;
for (controller = &(__POWER__[0]); controller != &(__POWER_END__); controller++) {
    ...
}
```

Each controller has an associated priority, controlling the order in which they appear in the table. Typically a software-only component such as a TCP/IP stack would use a small number for the priority, so that it appears near the start of the table, whereas a device driver would be nearer the back of the table. When switching to a lower-powered mode the power management package will iterate through this table from front to back, thus ensuring that for example the TCP/IP stack gets a chance to shut down before the underlying ethernet or other hardware that the stack depends on. Similarly when switching to a higher-powered mode the power management package will iterate through this table from back to front.

In most systems there will be one special controller, power_controller_cpu, which should be provided by one of the architectural, variant or platform HAL packages. This controller will always be the last entry in the table. It is responsible for the final power down operation when switching to off mode. Other packages such as device drivers may or may not declare variable identifiers for their power controllers, allowing those controllers to be accessed by name as well as by their entries in the global table.
Global Power Modes

The function `power_get_mode` can be called at any time to determine the current power mode for the system as a whole. The return value will be one of `PowerMode_Active`, `PowerMode_Idle`, `PowerMode_Sleep` or `PowerMode_Off`. In normal circumstances it is unlikely that `PowerMode_Off` would be returned since that mode generally means that the cpu is no longer running.

The function `power_get_desired_mode` returns the power mode that the system should be running at. Most of the time this will be the same value as returned by `power_get_mode`. However a different value may be returned when in the middle of changing power modes. For example, if the current thread runs at a higher priority than the power management thread then the latter may have been pre-empted in the middle of a mode change: `power_get_mode` will return the mode the system was running at before the mode change started, and `power_get_desired_mode` will return the mode the system should end up in when the mode change completes, barring further calls to `power_set_mode`.

Individual Controller Power Modes

The power management package keeps track of the current and desired modes for each power controller, as well as the modes for the system as a whole. The function `power_get_controller_mode` takes a single argument, a pointer to a power controller, and returns the power mode that controller is currently running at. Similarly `power_get_controller_desired_mode` returns the power mode that controller should be running at. Most of the time the current and desired modes for a given controller will be the same, and will also be the same as the global power mode. However if the power management thread is pre-empted in the middle of a mode change then some of the controllers will have been updated to the desired global mode, whereas others will still be at the old mode. The power management package also provides functionality for manipulating individual controllers, and for detaching controllers from global mode changes.

Power Controller Identification

In some scenarios the power management package will run completely automated, and there is no need to identify individual power controllers. Any form of identification such as a string description would serve no purpose, but would still consume memory in the final system. In other scenarios it may be very desirable to provide some means of identification. For example, while still debugging it may be useful to see a simple string when printing the contents of a power controller structure. Alternatively, if the application is expected to provide some sort of user interface that gives control over which parts of the system are enabled or disabled, a string identifier for each controller would be useful. To cope with these scenarios the power management package provides a configuration option `CYGIMP_POWER_PROVIDE_STRINGS`. When enabled, each power controller will contain a pointer to a constant string which can be accessed via a function `power_get_controller_id`. When disabled the system will not contain these strings, and the function will not be provided. The following code illustrates how to use this function.

```c
#include <stdio.h>
#include <pkgconf/system.h>
ifndef CYGPKG_POWER
#error The power management package is not present.
endif
#include <pkgconf/power.h>
ifndef CYGIMP_POWER_PROVIDE_STRINGS
#error Power controller identifiers are not available.
endif
#include <cyg/power/power.h>

static const char *
```
mode_to_string(PowerMode mode)
{
    const char* result;
    switch(mode) {
        case PowerMode_Active : result = "active"; break;
        case PowerMode_Idle   : result = "idle"; break;
        case PowerMode_Sleep  : result = "sleep"; break;
        case PowerMode_Off    : result = "off"; break;
        default : result = "<unknown>"; break;
    }
    return result;
}

int main(int argc, char** argv)
{
    PowerController* controller;

    for (controller = &(__POWER__[0]);
         controller != &(__POWER_END__);
         controller++) {
        printf("Controller @ %p: %s, %s\n", controller,
               power_get_controller_id(controller),
               mode_to_string(power_get_controller_mode(controller)));
    }
    return 0;
}

The Power Management Thread

If the power management package is configured to use a separate thread then a handle for that thread is made available to higher-level code via the variable power_thread_handle. This handle can be used for a variety of purposes, including manipulating that thread’s priority.
Power Management Information
Changing Power Modes

Name
Changing Power Modes — reducing or increasing power consumption as needed

Synopsis

```c
#include <cyg/power/power.h>
void power_set_mode ( PowerMode new_mode );
void power_set_controller_mode ( PowerController* controller , PowerMode new_mode );
void power_set_controller_mode_now ( PowerController* controller , PowerMode new_mode );
```

Changing the Global Power Mode

The primary functionality supported by the power management package is to change the system’s global power mode. This is achieved by calling the function `power_set_mode` with a single argument, which should be one of `PowerMode_Active`, `PowerMode_Idle`, `PowerMode_Sleep` or `PowerMode_Off`. Typically this function will only be invoked in certain scenarios:

1. A typical system will contain a policy module which is primarily responsible for initiating power mode changes, and a thread inside the power management package. The policy module will call `power_set_mode`, which has the effect of manipulating some internal state in the power management package and waking up its thread. When this thread gets scheduled to run (its priority is controlled by a configuration option), it will iterate over the power controllers and invoke each controller to change its power mode. There is support for a callback function, and for detached power controllers.

2. After a call to `power_set_mode` but before the power management thread has had a chance to iterate over all the controllers, or even before the thread has been rescheduled at all, the policy module may decide that a different power mode would be more appropriate for the current situation and calls `power_set_mode` again. This has the effect of aborting the previous mode change, followed by the power management thread iterating over the power controllers again for the new mode.

3. If there is no single policy module responsible for power mode changes, any code can call `power_set_mode`. If there are multiple calls in quick succession, earlier calls will be aborted and the system should end up in the power mode corresponding to the last call.

4. As a special case, it is possible for a power controller to call `power_set_mode` when invoked by the power management thread. For example a power controller could decide that it is inappropriate for the system to go to sleep because the device it is associated with is still busy. The effect is as if the policy module had called `power_set_mode` again before the mode change had completed.

If the power management package has been configured not to use a separate thread then obviously the behaviour is somewhat different. The call to `power_set_mode` will now iterate over the various power controllers immediately, rather than leaving this to a separate thread, and the whole mode change completes before `power_set_mode` returns. If some other thread or a DSR calls `power_set_mode` concurrently the behaviour of the system is undefined. However, it is still legal for a power controller to call `power_set_mode`; effectively this is a recursive call; it is detected by the system, and internal state is updated; the recursive `power_set_mode` call
now returns, and when the power controller returns back to the original `power_set_mode` call it detects what has happened, aborts the previous mode change, and starts a new mode change as requested by the controller.

`power_set_mode` is normally invoked from thread context. If a separate power management thread is used it can be invoked safely from DSR context. If the system is configured not to use such a thread, it may or may not be safe to invoke this function from DSR context: essentially the function just iterates through the various power controllers, and the documentation or source code of each controller present in the current system will have to be examined to determine whether or not this can happen safely in DSR context. `power_set_mode` should never be invoked from ISR context.

**Manipulating an Individual Power Controller**

In some cases it is desirable to set the power mode of an individual controller separately from the mode for the system as a whole. For example if a device is not currently being used then the associated power controller could be set to `PowerMode_Off`, even while the system as a whole is still active. This can be achieved by calling the function `power_set_controller_mode`. It takes two arguments: the first identifies a particular controller; the second specifies the desired new power mode for that controller. The function operates in much the same way as `power_set_mode`, for example if a separate power management thread is being used then `power_set_controller_mode` operates by manipulating some internal state and waking up that thread. The limitations are also much the same as for `power_set_mode`, so for example `power_set_controller_mode` should not be invoked from inside ISRs.

Manipulating individual controllers is often used in conjunction with the function `power_set_controller_attached`, allowing the policy module to specify which controllers are affected by global mode changes.

**Direct Manipulation of a Power Controller**

In exceptional circumstances it may be necessary to invoke a power controller directly, bypassing the power management thread and higher-level functionality such as callback functions. The function `power_set_controller_mode_now` allows this. It takes two arguments, a controller and a mode, just like `power_set_controller_mode`.

Use of `power_set_controller_mode_now` is dangerous. For example no attempt is made to synchronise with any other power mode changes that might be happening concurrently. A possible use is when the system gets woken up out of sleep mode: depending on the hardware, on which power controllers are present, and on the application code it may be necessary to wake up some power controllers immediately before the system as a whole is ready to run again.
Support for Policy Modules

Name
Support for Policy Modules — closer integration with higher-level code

Synopsis

```c
#include <cyg/power/power.h>
void power_set_policy_callback ( void (*)(PowerController*, PowerMode, PowerMode,
PowerMode, PowerMode) callback );
void (*)(PowerController*, PowerMode, PowerMode, PowerMode, PowerMode)
power_get_policy_callback (void);
CYG_ADDRWORD power_get_controller_policy_data ( PowerController* controller );
void power_set_controller_policy_data ( PowerController* controller, CYG_ADDRWORD
data );
```

Policy Callbacks

The use of a separate thread to perform power mode changes in typical configurations can cause problems for some policy modules. Specifically, the policy module can request a mode change for the system as a whole or for an individual controller, but it does not know when the power management thread actually gets scheduled to run again and carry out the request. Although it would be possible for the policy module to perform some sort of polling, in general that is undesirable.

To avoid such problems the policy module can install a callback function using `power_set_policy_callback`. The current callback function can be retrieved using `power_get_policy_callback`. If a callback function has been installed then it will be called by the power management package whenever a power controller has been invoked to perform a mode change. The callback will be called in the context of the power management thread, so usually it will have to make use of thread synchronisation primitives to interact with the main policy module.

It is passed five arguments:

1. The power controller that has just been invoked to perform a mode change.
2. The mode this controller was running at before the invocation.
3. The current mode this controller is now running at.
4. The desired mode before the power controller was invoked. Usually this will be the same as the current mode, unless the controller has decided for some reason that this was inappropriate.
5. The current desired mode. This will differ from the previous argument only if there has was another call to `power_set_mode` or `power_set_controller_mode` while the power controller was being invoked, probably by the power controller itself.

A simple example of a policy callback function would be:

```c
static void
power_callback(
    PowerController* controller,
    PowerMode old_mode,
    PowerMode new_mode,
    PowerMode old_desired_mode,
```
Support for Policy Modules

```c
powerMode new_desired_mode) {
    printf("Power mode change: %s, %s -> %d
",
           power_get_controller_id(controller),
           mode_to_string(old_mode),
           mode_to_string(new_mode));
    CYG_UNUSED_PARAM(PowerMode, old_desired_mode);
    CYG_UNUSED_PARAM(PowerMode, new_desired_mode);
}
```

```c
int main(int argc, char** argv) {
    ...,
    power_set_policy_callback(&power_callback);
    ...
}
```

If `power_set_controller_mode_now` is used to manipulate an individual controller the policy callback will not be invoked. This function may get called from any context including DSRs, and even if there is already a call to the policy callback happening in some other context, so invoking the callback would usually be unsafe.

If the power management package has not been configured to use a separate thread then `power_set_mode` and `power_set_controller_mode` will manipulate the power controllers immediately and invoke the policy callback afterwards. Therefore the policy callback will typically run in the same context as the main policy module.

**Policy-specific Controller Data**

Some policy modules may want to associate some additional data with each power controller. This could be achieved by for example maintaining a hash table or similar data structure, but for convenience the power management package allows higher-level code, typically the policy module, to store and retrieve one word of data in each power controller. The function `power_set_controller_policy_data` takes two arguments, a pointer to a power controller and a `CYG_ADDRWORD` of data: by appropriate use of casts this word could be an integer or a pointer to some data structure. The matching function `power_get_controller_policy_data` retrieves the word previously installed, and can be cast back to an integer or pointer. The default value for the policy data is 0.

For example the following code fragment stores a simple index value in each power controller. This could then be retrieved by the policy callback.

```c
unsigned int i = 0;
PowerController* controller;

for (controller = &(__POWER__[0]);
     controller != &(__POWER_END__);
     controller++) {
    power_set_controller_policy_data(controller, (CYG_ADDRWORD) i++);
}
```

Not all policy modules will require per-controller data. The configuration option `CYGIMP_POWER_PROVIDE_POLICY_DATA` can be used to control this functionality, thus avoiding wasting a small amount of memory inside each power controller structure.
Attached and Detached Controllers

Name

Attached and Detached Controllers — control which power controllers are affected by global changes

Synopsis

```c
#include <cyg/power/power.h>
cyg_bool power_get_controller_attached ( PowerController* controller );
void power_set_controller_attached ( PowerController* controller , cyg_bool new_state );
```

Detaching Power Controllers

By default the global operation `power_set_mode` affects all power controllers. There may be circumstances when this is not desirable. For example if a particular device is not currently being used then it can be left switched off: the rest of the system could be moving between active, idle and sleep modes, but there is no point in invoking the power controller for the unused device. To support this the power management package supports the concept of attached and detached controllers. By default all controllers are attached, and hence will be affected by global mode changes. A specific controller can be detached using the function `power_set_controller_attached`. This function takes two arguments, one to specify a particular controller and another to specify the desired new state. `power_get_controller_attached` can be used to determine whether or not a specific controller is currently attached.

The attached or detached state of a controller only affects what happens during a global mode change, in other words following a call to `power_set_mode`. It is still possible to manipulate a detached controller using `power_set_controller_mode` or `power_set_controller_mode_now`. 
Implementing a Power Controller

Name
Implementing a Power Controller — adding power management support to device drivers and other packages

Implementing a Power Controller

A system will have some number of power controllers. Usually there will be one power controller for the cpu, power_controller_cpu, typically provided by one of the HAL packages and responsible for managing the processor itself and associated critical components such as memory. Some or all of the device drivers will provide power controllers, allowing the power consumption of the associated devices to be controlled. There may be some arbitrary number of other controllers present in the system. The power management package does not impose any restrictions on the number or nature of the power controllers in the system, other than insisting that at most one power_controller_cpu be provided.

Each power controller involves a single data structure of type PowerController, defined in the header file cyg/power/power.h. These data structures should all be placed in the table __POWER__, so that the power management package and other code can easily locate all the controllers in the system. This table is constructed at link-time, avoiding code-size or run-time overheads. To facilitate this the package provides two macros which should be used to define a power controller, POWER_CONTROLLER() and POWER_CONTROLLER_CPU().

The macro POWER_CONTROLLER takes four arguments:

1. A variable name. This can be used to access the power controller directly, as well as via the table.

2. A priority. The table of power controllers is sorted, such that power controllers with a numerically lower priority come earlier in the table. The special controller power_controller_cpu always comes at the end of the table. When moving from a high-power mode to a lower-powered mode, the power management package iterates through the table from front to back. When moving to a higher-powered mode the reverse direction is used. The intention is that the power controller for a software-only package such as a TCP/IP stack should appear near the start of the table, whereas the controllers for the ethernet and similar devices would be near the end of the table. Hence when the policy module initiates a mode change to a lower-powered mode the TCP/IP stack gets a chance to cancel this mode change, before the devices it depends on are powered down. Similarly when moving to a higher-powered mode the devices will be re-activated before any software that depends on those devices.

The header file cyg/power/power.h defines three priorities PowerPri_Early, PowerPri_Typical and PowerPri_Late. For most controllers one of these priorities, possibly with a small number added or subtracted, will give sufficient control. If an application developer is uncertain about the relative priorities of the various controllers, a simple test program that iterates over the table will quickly eliminate any confusion.

3. A constant string identifier. If the system has been configured without support for such identifiers (CYGIMP_POWER_PROVIDE_STRINGS) then this identifier will be discarded at compile-time. Otherwise it will be made available to higher-level code using the function power_get_controller_id.

4. A function pointer. This will be invoked to perform actual mode changes, as described below.

A typical example of the use of the POWER_CONTROLLER macro would be as follows:

```c
#include <pkgconf/system.h>

#ifndef CYGPKG_POWER

```
Implementing a Power Controller

```c
#include <cyg/power/power.h>

static void
xyzzy_device_power_mode_change(
    PowerController* controller,
    PowerMode desired_mode,
    PowerModeChange change)
{
    // Do the work
}

static POWER_CONTROLLER(xyzzy_power_controller, \
    PowerPri_Late, \
    "xyzzy device", \
    &xyzzy_device_power_mode_change);
#endif
```

This creates a variable `xyzzy_power_controller`, which is a power controller data structure that will end up near the end of the table of power controllers. Higher-level code can iterate through this table and report the string "xyzzy device" to the user. Whenever there is a mode change operation that affects this controller, the function `xyzzy_device_power_mode_change` will be invoked. The variable is declared static so this controller cannot be manipulated by name in any other code. Alternatively, if the variable had not been declared static other code could manipulate this controller by name as well as through the table, especially if the package for the xyzzy device driver explicitly declared this variable in an exported header file. Obviously exporting the variable involves a slight risk of a name clash at link time.

The above code explicitly checks for the presence of the power management package before including that package’s header file or providing any related functionality. Since power management functionality is optional, such checks are recommended.

The macro `POWER_CONTROLLER_CPU` only takes two arguments, a string identifier and a mode change function pointer. This macro always instantiates a variable `power_controller_cpu` so there is no need to provide a variable name. The resulting power controller structure always appears at the end of the table, so there is no need to specify a priority. Typical usage of the `POWER_CONTROLLER_CPU` macro would be:

```c
static void
wumpus_processor_power_mode_change(
    PowerController* controller,
    PowerMode desired_mode,
    PowerModeChange change)
{
    // Do the work
}

POWER_CONTROLLER_CPU("wumpus processor", \
    &wumpus_processor_power_mode_change);
```

This defines a power controller structure `power_controller_cpu`. It should not be declared static since higher-level code may well want to manipulate the cpu’s power mode directly, and the variable is declared by the power management package’s header file.

Some care has to be taken to ensure that the power controllers actually end up in the final executable. If a power controller variable ends up in an ordinary library and is never referenced directly then typically the linker will believe that the variable is not needed and it will not end up in the executable. For eCos packages this can be achieved in the CDL, by specifying that the containing source file should end up in `libextras.a` rather than the default `libtarget.a`:

```
cdl_package CYGPKG_HAL_WUMPUS_ARCH {
    ...
```
If the file `data.c` instantiates a power controller this is now guaranteed to end up in the final executable, as intended. Typically HAL and device driver packages will already have some data that must not be eliminated by the linker, so they will already contain a file that gets built into `libextras.a`. For power controllers defined inside application code it is important that the power controllers end up in `.o` object files rather than in `.a` library archive files.

All the real work of a power controller is done by the mode change function. If the power management package has been configured to use a separate thread then this mode change function will be invoked by that thread (except for the special case of `power_set_controller_mode_now`). If no separate thread is used then the mode change function will be invoked directly by `power_set_mode` or `power_set_controller_mode`.

The mode change function will be invoked with three arguments. The first argument identifies the power controller. Usually this argument is not actually required since a given mode change function will only ever be invoked for a single power controller. For example, `xyzzy_device_power_mode_change` will only ever be used in conjunction with `xyzzy_power_controller`. However there may be some packages which contain multiple controllers, all of which can share a single mode change function, and in that case it is essential to identify the specific controller. The second argument specifies the mode the controller should switch to, if possible: it will be one of `PowerMode_Active`, `PowerMode_Idle`, `PowerMode_Sleep` or `PowerMode_Off`. The final argument will be one of `PowerModeChange_Controller`, `PowerModeChange_ControllerNow`, or `PowerModeChange_Global`, and identifies the call that caused this invocation. For example, if the mode change function was invoked because of a call to `power_set_mode` then this argument will be `PowerModeChange_Global`. It is up to each controller to decide how to interpret this final argument. A typical controller might reject a global request to switch to off mode if the associated device is still busy, but if the request was aimed specifically at this controller then it could instead abort any current I/O operations and switch off the device.

The PowerController data structure contains one field, `mode`, that needs to be updated by the power mode change function. At all times it should indicate the current mode for this controller. When a mode change is requested the desired mode is passed as the second argument. The exact operation of the power mode change function depends very much on what is being controlled and the current circumstances, but some guidelines are possible:

1. If the request can be satisfied without obvious detriment, do so and update the `mode` field. Reducing the power consumption of a device that is not currently being used is generally harmless.
2. If a request is a no-op, for example if the system is switching from idle to sleep mode and the controller does not distinguish between these modes, simply act as if the request was satisfied.
3. If a request is felt to be unsafe, for example shutting down a device that is still in use, then the controller may decide to reject this request. This is especially true if the request was a global mode change as opposed to one intended specifically for this controller: in the latter case the policy module should be given due deference. There are a number of ways in which a request can be rejected:
   a. If the request cannot be satisfied immediately but may be feasible in a short while, leave the `mode` field unchanged. Higher-level code in the policy module can interpret this as a hint to retry the operation a little bit later. This approach is also useful if the mode change can be started but will take some time to complete, for example shutting down a socket connection, and additional processing will be needed later on.
   b. If the request is felt to be inappropriate, for example switching off a device that is still in use, the mode change function can call `power_set_controller_mode` to reset the desired mode for this controller back to the current mode. Higher-level code can then interpret this as a hint that there is more activity in the system than had been apparent.
Implementing a Power Controller

c. For a global mode change, if the new mode is felt to be inappropriate then the power controller can call \texttt{power\_set\_mode} to indicate this. An example of this would be the policy module deciding to switch off the whole unit while there is still I/O activity.

Mode change functions should not directly manipulate any other fields in the PowerController data structure. If it is necessary to keep track of additional data then static variables can be used.

It should be noted that the above are only guidelines. Their application in any given situation may be unclear. In addition the detailed requirements of specific systems will vary, so even if the power controller for a given device driver follows the above guidelines exactly it may turn out that slightly different behaviour would be more appropriate for the actual system that is being developed. Fortunately the open source nature of eCos allows system developers to fine-tune power controllers to meet their exact requirements.
XXXIII. eCos USB Slave Support
Implementing a Power Controller
Introduction

Name

Introduction — eCos support for USB slave devices

Introduction

The eCos USB slave support allows developers to produce USB peripherals. It consists of a number of different eCos packages:

1. Device drivers for specific implementations of USB slave hardware, for example the on-chip USB Device Controller provided by the Intel SA110 processor. A typical USB peripheral will only provide one USB slave port and therefore only one such device driver package will be needed. Usually the device driver package will be loaded automatically when you create an eCos configuration for target hardware that has a USB slave device. If you select a target which does have a USB slave device but no USB device driver is loaded, this implies that no such device driver is currently available.

2. The common USB slave package. This serves two purposes. It defines the API that specific device drivers should implement. It also provides various utilities that will be needed by most USB device drivers and applications, such as handlers for standard control messages. Usually this package will be loaded automatically at the same time as the USB device driver.

3. The common USB package. This merely provides some information common to both the host and slave sides of USB, such as details of the control protocol. It is also used to place the other USB-related packages appropriately in the overall configuration hierarchy. Usually this package will be loaded at the same time as the USB device driver.

4. Class-specific USB support packages. These make it easier to develop specific classes of USB peripheral, such as a USB-ethernet device. If no suitable package is available for a given class of peripheral then the USB device driver can instead be accessed directly from application code. Such packages will never be loaded automatically since the configuration system has no way of knowing what class of USB peripheral is being developed. Instead developers have to add the appropriate package or packages explicitly.

These packages only provide support for developing USB peripherals, not USB hosts.

USB Concepts

Information about USB can be obtained from a number of sources including the USB Implementers Forum web site (http://www.usb.org/). Only a brief summary is provided here.

A USB network is asymmetrical: it consists of a single host, one or more slave devices, and possibly some number of intermediate hubs. The host side is significantly more complicated than the slave side. Essentially, all operations are initiated by the host. For example, if the host needs to receive some data from a particular USB peripheral then it will send an IN token to that peripheral; the latter should respond with either a NAK or with appropriate data. Similarly, when the host wants to transmit data to a peripheral it will send an OUT token followed by the data; the peripheral will return a NAK if it is currently unable to receive more data or if there was corruption, otherwise it will return an ACK. All transfers are check-summed and there is a clearly-defined error recovery process. USB peripherals can only interact with the host, not with each other.

USB supports four different types of communication: control messages, interrupt transfers, isochronous transfers, and bulk transfers. Control messages are further subdivided into four categories: standard, class, vendor and a reserved category. All USB peripherals must respond to certain standard control messages, and usually
this will be handled by the common USB slave package (for complicated peripherals, application support will be needed). Class and vendor control messages may be handled by an class-specific USB support package, for example the USB-ethernet package will handle control messages such as getting the MAC address or enabling/disabling promiscuous mode. Alternatively, some or all of these messages will have to be handled by application code.

Interrupt transfers are used for devices which need to be polled regularly. For example, a USB keyboard might be polled once every millisecond. The host will not poll the device more frequently than this, so interrupt transfers are best suited to peripherals that involve a relatively small amount of data. Isochronous transfers are intended for multimedia-related peripherals where typically a large amount of video or audio data needs to be exchanged continuously. Given appropriate host support a USB peripheral can reserve some of the available bandwidth. Isochronous transfers are not reliable; if a particular packet is corrupted then it will just be discarded and software is expected to recover from this. Bulk transfers are used for everything else: after taking care of any pending control, isochronous and interrupt transfers the host will use whatever bandwidth remains for bulk transfers. Bulk transfers are reliable.

Transfers are organized into USB packets, with the details depending on the transfer type. Control messages always involve an initial 8-byte packet from host to peripheral, optionally followed by some additional packets; in theory these additional packets can be up to 64 bytes, but hardware may limit it to 8 bytes. Interrupt transfers involve a single packet of up to 64 bytes. Isochronous transfers involve a single packet of up to 1024 bytes. Bulk transfers involve multiple packets. There will be some number, possibly zero, of 64-byte packets. The transfer is terminated by a single packet of less than 64 bytes. If the transfer involves an exact multiple of 64 bytes than the final packet will be 0 bytes, consisting of just a header and checksum which typically will be generated by the hardware. There is no pre-defined limit on the size of a bulk transfer. Instead higher-level protocols are expected to handle this, so for a USB-ethernet peripheral the protocol could impose a limit of 1514 bytes of data plus maybe some additional protocol overhead.

Transfers from the host to a peripheral are addressed not just to that peripheral but to a specific endpoint within that peripheral. Similarly, the host requests incoming data from a specific endpoint rather than from the peripheral as a whole. For example, a combined keyboard/touchpad device could provide the keyboard events on endpoint 1 and the mouse events on endpoint 2. A given USB peripheral can have up to 16 endpoints for incoming data and another 16 for outgoing data. However, given the comparatively high speed of USB I/O this endpoint addressing is typically implemented in hardware rather than software, and the hardware will only implement a small number of endpoints. Endpoint 0 is generally used only for control messages.

In practice, many of these details are irrelevant to application code or to class packages. Instead, such higher-level code usually just performs blocking read and write, or non-blocking USB-specific calls, to transfer data between host and target via a specific endpoint. Control messages are more complicated but are usually handled by existing code.

When a USB peripheral is plugged into the host there is an initial enumeration and configuration process. The peripheral provides information such as its class of device (audio, video, etc.), a vendor id, which endpoints should be used for what kind of data, and so on. The host OS uses this information to identify a suitable host device driver. This could be a generic driver for a class of peripherals, or it could be a vendor-specific driver. Assuming a suitable driver is installed the host will then activate the USB peripheral and perform additional application-specific initialisation. For example for a USB-ethernet device this would involve obtaining an ethernet MAC address. Most USB peripherals will be fairly simple, but it is possible to build multifunction peripherals with multiple configurations, interfaces, and alternate interface settings.

It is not possible for any of the eCos packages to generate all the enumeration data automatically. Some of the required information such as the vendor id cannot be supplied by generic packages; only by the application developer. Class support code such as the USB-ethernet package could in theory supply some of the information automatically, but there are also hardware dependencies such as which endpoints get used for incoming and outgoing ethernet frames. Instead it is the responsibility of the application developer to provide all the enumeration data and perform some additional initialisation. In addition, the common USB slave package can
handle all the standard control messages for a simple USB peripheral, but for something like a multifunction peripheral additional application support is needed.

**Note:** The initial implementation of the eCos USB slave packages involved hardware that only supported control and bulk transfers, not isochronous or interrupt. There may be future changes to the USB code and API to allow for isochronous and interrupt transfers, especially the former. Other changes may be required to support different USB devices. At present there is no support for USB remote wakeups, since again it is not supported by the hardware.

### eCos USB I/O Facilities

For protocols other than control messages, eCos provides two ways of performing USB I/O. The first involves device table or devtab entries such as `/dev/usb1r`, with one entry per endpoint per USB device. It is possible to open these devices and use conventional blocking I/O functions such as `read` and `write` to exchange data between host and peripheral.

There is also a lower-level USB-specific API, consisting of functions such as `usbs_start_rx_buffer`. A USB device driver will supply a data structure for each endpoint, for example a `usbs_rx_endpoint` structure for every receive endpoint. The first argument to `usbs_start_rx_buffer` should be a pointer to such a data structure. The USB-specific API is non-blocking: the initial call merely starts the transfer; some time later, once the transfer has completed or has been aborted, the device driver will invoke a completion function.

Control messages are different. With four different categories of control messages including application and vendor specific ones, the conventional `open/read/write` model of I/O cannot easily be applied. Instead, a USB device driver will supply a `usbs_control_endpoint` data structure which can be manipulated appropriately. In practice the standard control messages will usually be handled by the common USB slave package, and other control messages will be handled by class-specific code such as the USB-ethernet package. Typically, application code remains responsible for supplying the enumeration data and for actually starting up the USB device.

### Enabling the USB code

If the target hardware contains a USB slave device then the appropriate USB device driver and the common packages will typically be loaded into the configuration automatically when that target is selected (assuming a suitable device driver exists). However, the driver will not necessarily be active. For example a processor might have an on-chip USB device, but not all applications using that processor will want to use USB functionality. Hence by default the USB device is disabled, ensuring that applications do not suffer any memory or other penalties for functionality that is not required.

If the application developer explicitly adds a class support package such as the USB-ethernet one then this implies that the USB device is actually needed, and the device will be enabled automatically. However, if no suitable class package is available and the USB device will instead be accessed by application code, it is necessary to enable the USB device manually. Usually the easiest way to do this is to enable the configuration option `CYGGLO_IO_USB_SLAVE_APPLICATION`, and the USB device driver and related packages will adjust accordingly. Alternatively, the device driver may provide some configuration options to provide more fine-grained control.
USB Enumeration Data

Name

Enumeration Data — The USB enumeration data structures

Synopsis

#include <cyg/io/usb/usb.h>
#include <cyg/io/usb/usbs.h>

typedef struct usb_device_descriptor {
  ...
} usb_device_descriptor __attribute__((packed));

typedef struct usb_configuration_descriptor {
  ...
} usb_configuration_descriptor __attribute__((packed));

typedef struct usb_interface_descriptor {
  ...
} usb_interface_descriptor __attribute__((packed));

typedef struct usb_endpoint_descriptor {
  ...
} usb_endpoint_descriptor;

typedef struct usbs_enumeration_data {
  usb_device_descriptor device;
  int total_number_interfaces;
  int total_number_endpoints;
  int total_number_strings;
  const usb_configuration_descriptor* configurations;
  const usb_interface_descriptor* interfaces;
  const usb_endpoint_descriptor* endpoints;
  const unsigned char** strings;
} usbs_enumeration_data;

USB Enumeration Data

When a USB host detects that a peripheral has been plugged in or powered up, one of the first steps is to ask the peripheral to describe itself by supplying enumeration data. Some of this data depends on the class of peripheral. Other fields are vendor-specific. There is also a dependency on the hardware, specifically which endpoints are available should be used. In general it is not possible for generic code to provide this information, so it is the responsibility of application code to provide a suitable usbs_enumeration_data data structure and install it in the endpoint 0 data structure during initialization. This must happen before the USB device is enabled by a call to usbs_start, for example:

    const usbs_enumeration_data usb_enum_data = {
      ...
    };

    int
    main(int argc, char** argv)
USB Enumeration Data

```c
{
    usbs_sa11x0_ep0.enumeration_data = &usb_enum_data;
    ...
    usbs_start(&usbs_sa11x0_ep0);
    ...
}
```

For most applications the enumeration data will be static, although the `usbs_enumeration_data` structure can be filled in at run-time if necessary. Full details of the enumeration data can be found in the Universal Serial Bus specification obtainable from the USB Implementers Forum web site (http://www.usb.org/), although the meaning of most fields is fairly obvious. The various data structures and utility macros are defined in the header files `cyg/io/usb/usb.h` and `cyg/io/usb/usbs.h`. Note that the example code below makes use of the gcc labelled element extension.

**usb_device_descriptor**

The main information about a USB peripheral comes from a single `usb_device_descriptor` structure, which is embedded in the `usbs_enumeration_data` structure. A typical example might look like this:

```c
const usbs_enumeration_data usb_enum_data = {
    {
        length: USB_DEVICE_DESCRIPTOR_LENGTH,
        type: USB_DEVICE_DESCRIPTOR_TYPE,
        usb_spec_lo: USB_DEVICE_DESCRIPTOR_USB11_LO,
        usb_spec_hi: USB_DEVICE_DESCRIPTOR_USB11_HI,
        device_class: USB_DEVICE_DESCRIPTOR_CLASS_VENDOR,
        device_subclass: USB_DEVICE_DESCRIPTOR_SUBCLASS_VENDOR,
        device_protocol: USB_DEVICE_DESCRIPTOR_PROTOCOL_VENDOR,
        max_packet_size: 8,
        vendor_lo: 0x42,
        vendor_hi: 0x42,
        product_lo: 0x42,
        product_hi: 0x42,
        device_lo: 0x00,
        device_hi: 0x01,
        manufacturer_str: 1,
        product_str: 2,
        serial_number_str: 0,
        number_configurations: 1
    },
    ...
};
```

The length and type fields are specified by the USB standard. The `usb_spec_lo` and `usb_spec_hi` fields identify the particular revision of the standard that the peripheral implements, for example revision 1.1.

The device class, subclass, and protocol fields are used by generic host-side USB software to determine which host-side device driver should be loaded to interact with the peripheral. A number of standard classes are defined, for example mass-storage devices and human-interface devices. If a peripheral implements one of the standard classes then a standard existing host-side device driver may exist, eliminating the need to write a custom driver. The value 0xFF (VENDOR) is reserved for peripherals that implement a vendor-specific protocol rather than a standard one. Such peripherals will require a custom host-side device driver. The value 0x00 (INTERFACE) is reserved and indicates that the protocol used by the peripheral is defined at the interface level rather than for the peripheral as a whole.
The `max_package_size` field specifies the maximum length of a control message. There is a lower bound of eight bytes, and typical hardware will not support anything larger because control messages are usually small and not performance-critical.

The `vendor_lo` and `vendor_hi` fields specify a vendor id, which must be obtained from the USB Implementor’s Forum. The numbers used in the code fragment above are examples only and must not be used in real USB peripherals. The product identifier is determined by the vendor, and different USB peripherals should use different identifiers. The device identifier field should indicate a release number in binary-coded decimal.

The above fields are all numerical in nature. A USB peripheral can also provide a number of strings as described below, for example the name of the vendor can be provided. The various `str` fields act as indices into an array of strings, with index 0 indicating that no string is available.

A typical USB peripheral involves just a single configuration. However more complicated peripherals can support multiple configurations. Only one configuration will be active at any one time, and the host will switch between them as appropriate. If a peripheral does involve multiple configurations then typically it will be the responsibility of application code to handle the standard set-configuration control message.

**usb_configuration_descriptor**

A USB peripheral involves at least one and possible several different configurations. The `usbs_enumeration_data` structure requires a pointer to an array, possibly of length 1, of `usb_configuration_descriptor` structures. Usually a single structure suffices:

```c
const usb_configuration_descriptor usb_configuration = {
    length: USB_CONFIGURATION_DESCRIPTOR_LENGTH,
    type: USB_CONFIGURATION_DESCRIPTOR_TYPE,
    total_length_lo: USB_CONFIGURATION_DESCRIPTOR_TOTAL_LENGTH_LO(1, 2),
    total_length_hi: USB_CONFIGURATION_DESCRIPTOR_TOTAL_LENGTH_HI(1, 2),
    number_interfaces: 1,
    configuration_id: 1,
    configuration_str: 0,
    attributes: USB_CONFIGURATION_DESCRIPTOR_ATTR_REQUIRED | USB_CONFIGURATION_DESCRIPTOR_ATTR_SELF_POWERED,
    max_power: 50
};

const usbs_enumeration_data usb_enum_data = {
    ... configurations: &usb_configuration, ... 
};
```

The values for the `length` and `type` fields are determined by the standard. The `total_length` field depends on the number of interfaces and endpoints used by this configuration, and convenience macros are provided to calculate this: the first argument to the macros specify the number of interfaces, the second the number of endpoints. The `number_interfaces` field is self-explanatory. If the peripheral involves multiple configurations then each one must have a unique id, and this will be used in the set-configuration control message. The id 0 is reserved, and a set-configuration control message that uses this id indicates that the peripheral should be inactive. Configurations can have a string description if required. The `attributes` field must have the `REQUIRED` bit set; the `SELF_POWERED` bit informs the host that the peripheral has its own power supply and will not draw any power over the bus, leaving more bus power available to other peripherals; the `REMOTE_WAKEUP` bit is used if the peripheral can interrupt the host when the latter is in power-saving mode. For peripherals that are not self-powered, the `max_power` field specifies the power requirements in units of 2mA.
USB Enumeration Data

**usb_interface_descriptor**

A USB configuration involves one or more interfaces, typically corresponding to different streams of data. For example, one interface might involve video data while another interface is for audio. Multiple interfaces in a single configuration will be active at the same time.

```c
const usb_interface_descriptor usb_interface = {
    length: USB_INTERFACE_DESCRIPTOR_LENGTH,
    type: USB_INTERFACE_DESCRIPTOR_TYPE,
    interface_id: 0,
    alternate_setting: 0,
    number_endpoints: 2,
    interface_class: USB_INTERFACE_DESCRIPTOR_CLASS_VENDOR,
    interface_subclass: USB_INTERFACE_DESCRIPTOR_SUBCLASS_VENDOR,
    interface_protocol: USB_INTERFACE_DESCRIPTOR_PROTOCOL_VENDOR,
    interface_str: 0
};
```

```c
const usb Enumeration_data usb_enum_data = {
    ...
    total_number_interfaces: 1,
    interfaces: &usb_interface,
    ...
};
```

Again, the `length` and `type` fields are specified by the standard. Each interface within a configuration requires its own id. However, a given interface may have several alternate settings, in other words entries in the interfaces array with the same id but different `alternate_setting` fields. For example, there might be one setting which requires a bandwidth of 100K/s and another setting that only needs 50K/s. The host can use the standard set-interface control message to choose the most appropriate setting. The handling of this request is the responsibility of higher-level code, so the application may have to install its own handler. The number of endpoints used by an interface is specified in the `number_endpoints` field. Exact details of which endpoints are used is held in a separate array of endpoint descriptors. The class, subclass and protocol fields are used by host-side code to determine which host-side device driver should handle this specific interface. Usually this is determined on a per-peripheral basis in the `usb_device_descriptor` structure, but that can defer the details to individual interfaces. A per-interface string is allowed as well.

For USB peripherals involving multiple configurations, the array of `usb_interface_descriptor` structures should first contain all the interfaces for the first configuration, then all the interfaces for the second configuration, and so on.

**usb_endpoint_descriptor**

The host also needs information about which endpoint should be used for what. This involves an array of endpoint descriptors:

```c
const usb_endpoint_descriptor usb_endpoints[] = {
    {
        length: USB_ENDPOINT_DESCRIPTOR_LENGTH,
        type: USB_ENDPOINT_DESCRIPTOR_TYPE,
        endpoint: USB_ENDPOINT_DESCRIPTOR_ENDPOINT_OUT | 1,
        attributes: USB_ENDPOINT_DESCRIPTOR_ATTR_BULK,
        max_packet_lo: 64,
        max_packet_hi: 0,
        interval: 0
    },
...```

The `length` and `type` fields are specified by the standard. Each interface within a configuration requires its own id. However, a given interface may have several alternate settings, in other words entries in the interfaces array with the same id but different `alternate_setting` fields. For example, there might be one setting which requires a bandwidth of 100K/s and another setting that only needs 50K/s. The host can use the standard set-interface control message to choose the most appropriate setting. The handling of this request is the responsibility of higher-level code, so the application may have to install its own handler. The number of endpoints used by an interface is specified in the `number_endpoints` field. Exact details of which endpoints are used is held in a separate array of endpoint descriptors. The class, subclass and protocol fields are used by host-side code to determine which host-side device driver should handle this specific interface. Usually this is determined on a per-peripheral basis in the `usb_device_descriptor` structure, but that can defer the details to individual interfaces. A per-interface string is allowed as well.

For USB peripherals involving multiple configurations, the array of `usb_interface_descriptor` structures should first contain all the interfaces for the first configuration, then all the interfaces for the second configuration, and so on.
USB Enumeration Data

```
{
    length: USB_ENDPOINT_DESCRIPTOR_LENGTH,
    type: USB_ENDPOINT_DESCRIPTOR_TYPE,
    endpoint: USB_ENDPOINT_DESCRIPTOR_ENDPOINT_IN | 2,
    attributes: USB_ENDPOINT_DESCRIPTOR_ATTR_BULK,
    max_packet_lo: 64,
    max_packet_hi: 0,
    interval: 0
}

const usbs_enumeration_data usb_enum_data = {
...            total_number_endpoints: 2,
    endpoints: usb_endpoints,
...    
};
```

As usual the values for the `length` and `type` fields are specified by the standard. The `endpoint` field gives both the endpoint number and the direction, so in the above example endpoint 1 is used for OUT (host to peripheral) transfers and endpoint 2 is used for IN (peripheral to host) transfers. The `attributes` field indicates the USB protocol that should be used on this endpoint: CONTROL, ISOCHRONOUS, BULK or INTERRUPT. The `max_packet` field specifies the maximum size of a single USB packet. For bulk transfers this will typically be 64 bytes. For isochronous transfers this can be up to 1023 bytes. For interrupt transfers it can be up to 64 bytes, although usually a smaller value will be used. The `interval` field is ignored for control and bulk transfers. For isochronous transfers it should be set to 1. For interrupt transfers it can be a value between 1 and 255, and indicates the number of milliseconds between successive polling operations.

For USB peripherals involving multiple configurations or interfaces the array of endpoint descriptors should be organized sequentially: first the endpoints corresponding to the first interface of the first configuration, then the second interface in that configuration, and so on; then all the endpoints for all the interfaces in the second configuration; etc.

**Strings**

The enumeration data can contain a number of strings with additional information. Unicode encoding is used for the strings, and it is possible for a peripheral to supply a given string in multiple languages using the appropriate characters. The first two bytes of each string give a length and type field. The first string is special; after the two bytes header it consists of an array of 2-byte language id codes, indicating the supported languages. The language code 0x0409 corresponds to English (United States).

```
const unsigned char* usb_strings[] = {
    "\004\003\011\004",
    "\020\003\00e\00d\000  \000\000\00a\000t\000"
};
```

```
const usbs_enumeration_data usb_enum_data = {
...            total_number_strings: 2,
    strings: usb_strings,
...    
};
```

The default handler for standard control messages assumes that the peripheral only uses a single language. If this is not the case then higher-level code will have to handle the standard get-descriptor control messages when a string descriptor is requested.
USB Enumeration Data

usbsEnumerationData

The usbsEnumerationData data structure collects together all the various descriptors that make up the enumeration data. It is the responsibility of application code to supply a suitable data structure and install it in the control endpoints’s enumeration_data field before the USB device is started.
Starting up a USB Device

Name

usbs_start — Starting up a USB Device

Synopsis

```c
#include <cyg/io/usb/usbs.h>
void usbs_start(usbs_control_endpoint* ep0);
```

Description

Initializing a USB device requires some support from higher-level code, typically the application, in the form of enumeration data. Hence it is not possible for the low-level USB driver to activate a USB device itself. Instead the higher-level code has to take care of this by invoking `usbs_start`. This function takes a pointer to a USB control endpoint data structure. USB device drivers should provide exactly one such data structure for every USB device, so the pointer uniquely identifies the device.

```c
cost usbs Enumeration_data usb_enum_data = {
    ...
};

int main(int argc, char** argv)
{
    usbs_Sa11x0_ep0.enumeration_data = &usb_enum_data;
    ...
    usbs_start(&usbs_Sa11x0_ep0);
    ...
}
```

The exact behaviour of `usbs_start` depends on the USB hardware and the device driver. A typical implementation would change the USB data pins from tristated to active. If the peripheral is already plugged into a host then the latter should detect this change and start interacting with the peripheral, including requesting the enumeration data. Some of this may happen before `usbs_start` returns, but given that multiple interactions between USB host and peripheral are required it is likely that the function will return before the peripheral is fully configured. Control endpoints provide a mechanism for informing higher-level code of USB state changes. `usbs_start` will return even if the peripheral is not currently connected to a host: it will not block until the connection is established.

`usbs_start` should only be called once for a given USB device. There are no defined error conditions. Note that the function affects the entire USB device and not just the control endpoint: there is no need to start any data endpoints as well.
Starting up a USB Device
Devtab Entries

Name

Devtab Entries — Data endpoint data structure

Synopsis

/dev/usb0c
/dev/usb1r
/dev/usb2w

USB device drivers provide two ways of transferring data between host and peripheral. The first involves USB-specific functionality such as `usbs_start_rx_buffer`. This provides non-blocking I/O: a transfer is started, and some time later the device driver will call a supplied completion function. The second uses the conventional I/O model: there are entries in the device table corresponding to the various endpoints. Standard calls such as `open` can then be used to get a suitable handle. Actual I/O happens via blocking `read` and `write` calls. In practice the blocking operations are simply implemented using the underlying non-blocking functionality.

Each endpoint will have its own devtab entry. The exact names are controlled by the device driver package, but typically the root will be `/dev/usb`. This is followed by one or more decimal digits giving the endpoint number, followed by `c` for a control endpoint, `r` for a receive endpoint (host to peripheral), and `w` for a transmit endpoint (peripheral to host). If the target hardware involves more than one USB device then different roots should be used, for example `/dev/usb0c` and `/dev/usb1_0c`. This may require explicit manipulation of device driver configuration options by the application developer.

At present the devtab entry for a control endpoint does not support any I/O operations.

write operations

cyg_io_write and similar functions in higher-level packages can be used to perform a transfer from peripheral to host. Successive write operations will not be coalesced. For example, when doing a 1000 byte write to an endpoint that uses the bulk transfer protocol this will involve 15 full-size 64-byte packets and a terminating 40-byte packet. USB device drivers are not expected to do any locking, and if higher-level code performs multiple concurrent write operations on a single endpoint then the resulting behaviour is undefined.

A USB write operation will never transfer less data than specified. It is the responsibility of higher-level code to ensure that the amount of data being transferred is acceptable to the host-side code. Usually this will be defined by a higher-level protocol. If an attempt is made to transfer more data than the host expects then the resulting behaviour is undefined.

There are two likely error conditions. `EPIPE` indicates that the connection between host and target has been broken. `EAGAIN` indicates that the endpoint has been stalled, either at the request of the host or by other activity inside the peripheral.

read operations

cyg_io_read and similar functions in higher-level packages can be used to perform a transfer from host to peripheral. This should be a complete transfer: higher-level protocols should define an upper bound on the
Devtab Entries

Amount of data being transferred, and the read operation should involve at least this amount of data. The return value will indicate the actual transfer size, which may be less than requested.

Some device drivers may support partial reads, but USB device drivers are not expected to perform any buffering because that involves both memory and code overheads. One technique that may work for bulk transfers is to exploit the fact that such transfers happen in 64-byte packets. It is possible to read an initial 64 bytes, corresponding to the first packet in the transfer. These 64 bytes can then be examined to determine the total transfer size, and the remaining data can be transferred in another read operation. This technique is not guaranteed to work with all USB hardware. Also, if the delay between accepting the first packet and the remainder of the transfer is excessive then this could cause timeout problems for the host-side software. For these reasons the use of partial reads should be avoided.

There are two likely error conditions. EPIPE indicates that the connection between host and target has been broken. EAGAIN indicates that the endpoint has been stalled, either at the request of the host or by other activity inside the peripheral.

USB device drivers are not expected to do any locking. If higher-level code performs multiple concurrent read operations on a single endpoint then the resulting behaviour is undefined.

select operations

Typical USB device drivers will not provide any support for select. Consider bulk transfers from the host to the peripheral. At the USB device driver level there is no way of knowing in advance how large a transfer will be, so it is not feasible for the device driver to buffer the entire transfer. It may be possible to buffer part of the transfer, for example the first 64-byte packet, and copy this into application space at the start of a read, but this adds code and memory overheads. Worse, it means that there is an unknown but potentially long delay between a peripheral accepting the first packet of a transfer and the remaining packets, which could confuse or upset the host-side software.

With some USB hardware it may be possible for the device driver to detect OUT tokens from the host without actually accepting the data, and this would indicate that a read is likely to succeed. However, it would not be reliable since the host-side I/O operation could time out. A similar mechanism could be used to implement select for outgoing data, but again this would not be reliable.

Some device drivers may provide partial support for select anyway, possibly under the control of a configuration option. The device driver’s documentation should be consulted for further information. It is also worth noting that the USB-specific non-blocking API can often be used as an alternative to select.

get_config and set_config operations

There are no set_config or get_config (also known as ioctl) operations defined for USB devices. Some device drivers may provide hardware-specific facilities this way.

Note: Currently the USB-specific functions related to halted endpoints cannot be accessed readily via devtab entries. This functionality should probably be made available via set_config and get_config. It may also prove useful to provide a get_config operation that maps from the devtab entries to the underlying endpoint data structures.
Presence

The devtab entries are optional. If the USB device is accessed primarily by class-specific code such as the USB-ethernet package and that package uses the USB-specific API directly, the devtab entries are redundant. Even if application code does need to access the USB device, the non-blocking API may be more convenient than the blocking I/O provided via the devtab entries. In these cases the devtab entries serve no useful purpose, but they still impose a memory overhead. It is possible to suppress the presence of these entries by disabling the configuration option `CYGGLO_IO_USB_SLAVE_PROVIDEDEVTAB_ENTRIES`. 
Devtab Entries
Receiving Data from the Host

Name
usbs_start_rx_buffer — Receiving Data from the Host

Synopsis

```c
#include <cyg/io/usb/usbs.h>
void usbs_start_rx_buffer(usbs_rx_endpoint* ep, unsigned char* buffer, int length, void (*)(void*, int) complete_fn, void * complete_data);
void usbs_start_rx(usbs_rx_endpoint* ep);
```

Description

usbs_start_rx_buffer is a USB-specific function to accept a transfer from host to peripheral. It can be used for bulk, interrupt or isochronous transfers, but not for control messages. Instead those involve manipulating the usbs_control_endpoint data structure directly. The function takes five arguments:

1. The first argument identifies the specific endpoint that should be used. Different USB devices will support different sets of endpoints and the device driver will provide appropriate data structures. The device driver’s documentation should be consulted for details of which endpoints are available.

2. The buffer and length arguments control the actual transfer. USB device drivers are not expected to perform any buffering or to support partial transfers, so the length specified should correspond to the maximum transfer that is currently possible and the buffer should be at least this large. For isochronous transfers the USB specification imposes an upper bound of 1023 bytes, and a smaller limit may be set in the enumeration data. Interrupt transfers are similarly straightforward with an upper bound of 64 bytes, or less as per the enumeration data. Bulk transfers are more complicated because they can involve multiple 64-byte packets plus a terminating packet of less than 64 bytes, so there is no predefined limit on the transfer size. Instead it is left to higher-level protocols to specify an appropriate upper bound.

One technique that may work for bulk transfers is to exploit the fact that such transfers happen in 64-byte packets: it may be possible to receive an initial 64 bytes, corresponding to the first packet in the transfer; these 64 bytes can then be examined to determine the total transfer size, and the remaining data can be transferred in another receive operation. This technique is not guaranteed to work with all USB hardware. Also, if the delay between accepting the first packet and the remainder of the transfer is excessive then this could cause timeout problems for the host-side software. For these reasons this technique should be avoided.

3. usbs_start_rx_buffer is non-blocking. It merely starts the receive operation, and does not wait for completion. At some later point the USB device driver will invoke the completion function parameter with two arguments: the completion data defined by the last parameter and a result field. A result \( \geq 0 \) indicates a successful transfer of that many bytes, which may be less than the upper bound imposed by the length argument. A result \( < 0 \) indicates an error. The most likely errors are \(-EPIPE\) to indicate that the connection between the host and the target has been broken, and \(-EAGAIN\) for when the endpoint has been halted. Specific USB device drivers may specify additional error conditions.

The normal sequence of events is that the USB device driver will update the appropriate hardware registers. At some point after that the host will attempt to send data by transmitting an OUT token followed by a data
Receiving Data from the Host

packet, and since a receive operation is now in progress the data will be accepted and ACK’d. If there were
no receive operation then the peripheral would instead generate a NAK. The USB hardware will generate an
interrupt once the whole packet has been received, and the USB device driver will service this interrupt and
arrange for a DSR to be called. Isochronous and interrupt transfers involve just a single packet. However, bulk
transfers may involve multiple packets so the device driver has to check whether the packet was a full 64 bytes
or whether it was a terminating packet of less than this. When the device driver DSR detects a complete transfer
it will inform higher-level code by invoking the supplied completion function.

This means that the completion function will normally be invoked by a DSR and not in thread context -
although some USB device drivers may have a different implementation. Therefore the completion function
is restricted in what it can do. In particular it must not make any calls that will or may block such as locking
a mutex or allocating memory. The kernel documentation should be consulted for more details of DSR’s and
interrupt handling generally.

It is possible that the completion function will be invoked before usbs_start_rx_buffer returns. Such an
event would be unusual because the transfer cannot happen until the next time the host tries to send data to this
peripheral, but it may happen if for example another interrupt happens and a higher priority thread is scheduled
to run. Also, if the endpoint is currently halted then the completion function will be invoked immediately with
-EAGAIN: typically this will happen in the current thread rather than in a separate DSR. The completion function
is allowed to start another transfer immediately by calling usbs_start_rx_buffer again.

USB device drivers are not expected to perform any locking. It is the responsibility of higher-level code to
ensure that there is only one receive operation for a given endpoint in progress at any one time. If there
are concurrent calls to usbs_start_rx_buffer then the resulting behaviour is undefined. For typical USB
applications this does not present any problems, because only one piece of code will access a given endpoint
at any particular time.

The following code fragment illustrates a very simple use of usbs_start_rx_buffer to implement a blocking
receive, using a semaphore to synchronise between the foreground thread and the DSR. For a simple example
like this no completion data is needed.

```c
static int error_code = 0;
static cyg_sem_t completion_wait;

static void
completion_fn(void* data, int result)
{
    error_code = result;
    cyg_semaphore_post(&completion_wait);
}

int
blocking_receive(usbs_rx_endpoint* ep, unsigned char* buf, int len)
{
    error_code = 0;
    usbs_start_rx_buffer(ep, buf, len, &completion_fn, NULL);
    cyg_semaphore_wait(&completion_wait);
    return error_code;
}
```

There is also a utility function usbs_start_rx. This can be used by code that wants to manipulate data
endpoints directly, specifically the complete_fn, complete_data, buffer and buffer_size fields.
usbs_start_tx just invokes a function supplied by the device driver.
Sending Data to the Host

Name

usbs_start_tx_buffer — Sending Data to the Host

Synopsis

```c
#include <cyg/io/usb/usbs.h>
void usbs_start_tx_buffer(usbs_tx_endpoint* ep, const unsigned char* buffer, int length, void (*)(void*,int) complete_fn, void * complete_data);
void usbs_start_tx(usbs_tx_endpoint* ep);
```

Description

usbs_start_tx_buffer is a USB-specific function to transfer data from peripheral to host. It can be used for bulk, interrupt or isochronous transfers, but not for control messages; instead those involve manipulating the usbs_control_endpoint data structure directly. The function takes five arguments:

1. The first argument identifies the specific endpoint that should be used. Different USB devices will support different sets of endpoints and the device driver will provide appropriate data structures. The device driver’s documentation should be consulted for details of which endpoints are available.

2. The `buffer` and `length` arguments control the actual transfer. USB device drivers are not allowed to modify the buffer during the transfer, so the data can reside in read-only memory. The transfer will be for all the data specified, and it is the responsibility of higher-level code to make sure that the host is expecting this amount of data. For isochronous transfers the USB specification imposes an upper bound of 1023 bytes, but a smaller limit may be set in the enumeration data. Interrupt transfers have an upper bound of 64 bytes or less, as per the enumeration data. Bulk transfers are more complicated because they can involve multiple 64-byte packets plus a terminating packet of less than 64 bytes, so the basic USB specification does not impose an upper limit on the total transfer size. Instead it is left to higher-level protocols to specify an appropriate upper bound. If the peripheral attempts to send more data than the host is willing to accept then the resulting behaviour is undefined and may well depend on the specific host operating system being used.

For bulk transfers, the USB device driver or the underlying hardware will automatically split the transfer up into the appropriate number of full-size 64-byte packets plus a single terminating packet, which may be 0 bytes.

3. `usbs_start_tx_buffer` is non-blocking. It merely starts the transmit operation, and does not wait for completion. At some later point the USB device driver will invoke the completion function parameter with two arguments: the completion data defined by the last parameter, and a result field. This result will be either an error code < 0, or the amount of data transferred which should correspond to the `length` argument. The most likely errors are -EPIPE to indicate that the connection between the host and the target has been broken, and -EAGAIN for when the endpoint has been halted. Specific USB device drivers may define additional error conditions.

The normal sequence of events is that the USB device driver will update the appropriate hardware registers. At some point after that the host will attempt to fetch data by transmitting an IN token. Since a transmit operation is now in progress the peripheral can send a packet of data, and the host will generate an ACK. At this point
the USB hardware will generate an interrupt, and the device driver will service this interrupt and arrange for a DSR to be called. Isochronous and interrupt transfers involve just a single packet. However, bulk transfers may involve multiple packets so the device driver has to check whether there is more data to send and set things up for the next packet. When the device driver DSR detects a complete transfer it will inform higher-level code by invoking the supplied completion function.

This means that the completion function will normally be invoked by a DSR and not in thread context - although some USB device drivers may have a different implementation. Therefore the completion function is restricted in what it can do, in particular it must not make any calls that will or may block such as locking a mutex or allocating memory. The kernel documentation should be consulted for more details of DSR’s and interrupt handling generally.

It is possible that the completion function will be invoked before \texttt{usbs\_start\_tx\_buffer} returns. Such an event would be unusual because the transfer cannot happen until the next time the host tries to fetch data from this peripheral, but it may happen if, for example, another interrupt happens and a higher priority thread is scheduled to run. Also, if the endpoint is currently halted then the completion function will be invoked immediately with -EAGAIN: typically this will happen in the current thread rather than in a separate DSR. The completion function is allowed to start another transfer immediately by calling \texttt{usbs\_start\_tx\_buffer} again.

USB device drivers are not expected to perform any locking. It is the responsibility of higher-level code to ensure that there is only one transmit operation for a given endpoint in progress at any one time. If there are concurrent calls to \texttt{usbs\_start\_tx\_buffer} then the resulting behaviour is undefined. For typical USB applications this does not present any problems because only piece of code will access a given endpoint at any particular time.

The following code fragment illustrates a very simple use of \texttt{usbs\_start\_tx\_buffer} to implement a blocking transmit, using a semaphore to synchronise between the foreground thread and the DSR. For a simple example like this no completion data is needed.

```c
static int error_code = 0;
static cyg_sem_t completion_wait;

static void
completion_fn(void* data, int result)
{
    error_code = result;
    cyg_semaphore_post(&completion_wait);
}

int
blocking_transmit(usbs_tx_endpoint* ep, const unsigned char* buf, int len)
{
    error_code = 0;
    usbs_start_tx_buffer(ep, buf, len, &completion_fn, NULL);
    cyg_semaphore_wait(&completion_wait);
    return error_code;
}
```

There is also a utility function \texttt{usbs\_start}. This can be used by code that wants to manipulate data endpoints directly, specifically the \texttt{complete\_fn}, \texttt{complete\_data}, \texttt{buffer} and \texttt{buffer\_size} fields. \texttt{usbs\_start\_tx} just calls a function supplied by the device driver.
Halted Endpoints

Name

Halted Endpoints — Support for Halting and Halted Endpoints

Synopsis

```c
#include <cyg/io/usb/usbs.h>
cyg_bool usbs_rx_endpoint_halted(usbs_rx_endpoint* ep);
void usbs_set_rx_endpoint_halted(usbs_rx_endpoint* ep, cyg_bool new_state);
void usbs_start_rx_endpoint_wait(usbs_rx_endpoint* ep, void (*)(void*, int) complete_fn, void *complete_data);
cyg_bool usbs_tx_endpoint_halted(usbs_tx_endpoint* ep);
void usbs_set_tx_endpoint_halted(usbs_tx_endpoint* ep, cyg_bool new_state);
void usbs_start_tx_endpoint_wait(usbs_tx_endpoint* ep, void (*)(void*, int) complete_fn, void *complete_data);
```

Description

Normal USB traffic involves straightforward handshakes, with either an ACK to indicate that a packet was transferred without errors, or a NAK if an error occurred, or if a peripheral is currently unable to process another packet from the host, or has no packet to send to the host. There is a third form of handshake, a STALL, which indicates that the endpoint is currently halted.

When an endpoint is halted it means that the host-side code needs to take some sort of recovery action before communication over that endpoint can resume. The exact circumstances under which this can happen are not defined by the USB specification, but one example would be a protocol violation if say the peripheral attempted to transmit more data to the host than was permitted by the protocol in use. The host can use the standard control messages get-status, set-feature and clear-feature to examine and manipulate the halted status of a given endpoint. There are USB-specific functions which can be used inside the peripheral to achieve the same effect. Once an endpoint has been halted the host can then interact with the peripheral using class or vendor control messages to perform appropriate recovery, and then the halted condition can be cleared.

Halting an endpoint does not constitute a device state change, and there is no mechanism by which higher-level code can be informed immediately. However, any ongoing receive or transmit operations will be aborted with an -EAGAIN error, and any new receives or transmits will fail immediately with the same error.

There are six functions to support halted endpoints, one set for receive endpoints and another for transmit endpoints, with both sets behaving in essentially the same way. The first, `usbs_rx_endpoint_halted`, can be used to determine whether or not an endpoint is currently halted: it takes a single argument that identifies the endpoint of interest. The second function, `usbs_set_rx_endpoint_halted`, can be used to change the halted condition of an endpoint: it takes two arguments; one to identify the endpoint and another to specify the new state. The last function `usbs_start_rx_endpoint_wait` operates in much the same way as `usbs_start_rx_buffer`: when the endpoint is no longer halted the device driver will invoke the supplied completion function with a status of 0. The completion function has the same signature as that for a transfer operation. Often it will be possible to use a single completion function and have the foreground code invoke either `usbs_start_rx_buffer` or `usbs_start_rx_endpoint_wait` depending on the current state of the endpoint.
Halted Endpoints
Control Endpoints

Name
Control Endpoints — Control endpoint data structure

Synopsis

```c
#include <cyg/io/usb/usbs.h>

typedef struct usbs_control_endpoint {
    /* ... */
} usbs_control_endpoint;
```

usbs_control_endpoint Data Structure

The device driver for a USB slave device should supply one `usbs_control_endpoint` data structure per USB device. This corresponds to endpoint 0 which will be used for all control message interaction between the host and that device. The data structure is also used for internal management purposes, for example to keep track of the current state. In a typical USB peripheral there will only be one such data structure in the entire system, but if there are multiple USB slave ports, allowing the peripheral to be connected to multiple hosts, then there will be a separate data structure for each one. The name or names of the data structures are determined by the device drivers. For example, the SA11x0 USB device driver package provides `usbs_sa11x0_ep0`.

The operations on a control endpoint do not fit cleanly into a conventional open/read/write I/O model. For example, when the host sends a control message to the USB peripheral this may be one of four types: standard, class, vendor and reserved. Some or all of the standard control messages will be handled automatically by the common USB slave package or by the device driver itself. Other standard control messages and the other types of control messages may be handled by a class-specific package or by application code. Although it would be possible to have devtab entries such as `/dev/usbs_ep0/standard` and `/dev/usbs_ep0/class`, and then support read and write operations on these devtab entries, this would add significant overhead and code complexity. Instead, all of the fields in the control endpoint data structure are public and can be manipulated directly by higher level code if and when required.

Control endpoints involve a number of callback functions, with higher-level code installing suitable function pointers in the control endpoint data structure. For example, if the peripheral involves vendor-specific control messages then a suitable handler for these messages should be installed. Although the exact details depend on the device driver, typically these callback functions will be invoked at DSR level rather than thread level. Therefore, only certain eCos functions can be invoked; specifically, those functions that are guaranteed not to block. If a potentially blocking function such as a semaphore wait or a mutex lock operation is invoked from inside the callback then the resulting behaviour is undefined, and the system as a whole may fail. In addition, if one of the callback functions involves significant processing effort then this may adversely affect the system’s real time characteristics. The eCos kernel documentation should be consulted for more details of DSR handling.

Initialization

The `usbs_control_endpoint` data structure contains the following fields related to initialization.

```c
typedef struct usbs_control_endpoint {
    ...
} usbs_control_endpoint;
```
Control Endpoints

const usbs_enumeration_data* enumeration_data;
void ( *start_fn)(usbs_control_endpoint*);
...
);

It is the responsibility of higher-level code, usually the application, to define the USB enumeration data. This needs to be installed in the control endpoint data structure early on during system startup, before the USB device is actually started and any interaction with the host is possible. Details of the enumeration data are supplied in the section USB Enumeration Data. Typically, the enumeration data is constant for a given peripheral, although it can be constructed dynamically if necessary. However, the enumeration data cannot change while the peripheral is connected to a host: the peripheral cannot easily claim to be a keyboard one second and a printer the next.

The start_fn member is normally accessed via the utility usbs_start rather than directly. It is provided by the device driver and should be invoked once the system is fully initialized and interaction with the host is possible. A typical implementation would change the USB data pins from tristated to active. If the peripheral is already plugged into a host then the latter should detect this change and start interacting with the peripheral, including requesting the enumeration data.

State

There are three usbs_control_endpoint fields related to the current state of a USB slave device, plus some state constants and an enumeration of the possible state changes:

typedef struct usbs_control_endpoint {
  ...
  int state;
  void ( *state_change_fn)(struct usbs_control_endpoint*, void*,
                          usbs_state_change, int);
  void* state_change_data;
  ...
};

#define USBS_STATE_DETACHED 0x01
#define USBS_STATE_ATTACHED 0x02
#define USBS_STATE_POWERED 0x03
#define USBS_STATE_DEFAULT 0x04
#define USBS_STATE_ADDRESSED 0x05
#define USBS_STATE_CONFIGURED 0x06
#define USBS_STATE_MASK 0x7F
#define USBS_STATE_SUSPENDED (1 << 7)

typedef enum {
  USBS_STATE_CHANGE_DETACHED = 1,
  USBS_STATE_CHANGE_ATTACHED = 2,
  USBS_STATE_CHANGE_POWERED = 3,
  USBS_STATE_CHANGE_RESET = 4,
  USBS_STATE_CHANGE_ADDRESSED = 5,
  USBS_STATE_CHANGE_CONFIGURED = 6,
  USBS_STATE_CHANGE_DECONFIGURED = 7,
  USBS_STATE_CHANGE_SUSPENDED = 8,
  USBS_STATE_CHANGE_RESUMED = 9
} usbs_state_change;

The USB standard defines a number of states for a given USB peripheral. The initial state is detached, where the peripheral is either not connected to a host at all or, from the host’s perspective, the peripheral has not
Control Endpoints

started up yet because the relevant pins are tristated. The peripheral then moves via intermediate attached and powered states to its default or reset state, at which point the host and peripheral can actually start exchanging data. The first message is from host to peripheral and provides a unique 7-bit address within the local USB network, resulting in a state change to addressed. The host then requests enumeration data and performs other initialization. If everything succeeds the host sends a standard set-configuration control message, after which the peripheral is configured and expected to be up and running. Note that some USB device drivers may be unable to distinguish between the detached, attached and powered states but generally this is not important to higher-level code.

A USB host should generate at least one token every millisecond. If a peripheral fails to detect any USB traffic for a period of time then typically this indicates that the host has entered a power-saving mode, and the peripheral should do the same if possible. This corresponds to the suspended bit. The actual state is a combination of suspended and the previous state, for example configured and suspended rather than just suspended. When the peripheral subsequently detects USB traffic it would switch back to the configured state.

The USB device driver and the common USB slave package will maintain the current state in the control endpoint's state field. There should be no need for any other code to change this field, but it can be examined whenever appropriate. In addition whenever a state change occurs the generic code can invoke a state change callback function. By default, no such callback function will be installed. Some class-specific packages such as the USB-ethernet package will install a suitable function to keep track of whether or not the host-peripheral connection is up, that is whether or not ethernet packets can be exchanged. Application code can also update this field. If multiple parties want to be informed of state changes, for example both a class-specific package and application code, then typically the application code will install its state change handler after the class-specific package and is responsible for chaining into the package's handler.

The state change callback function is invoked with four arguments. The first identifies the control endpoint. The second is an arbitrary pointer: higher-level code can fill in the state_change_data field to set this. The third argument specifies the state change that has occurred, and the last argument supplies the previous state (the new state is readily available from the control endpoint structure).

eCos does not provide any utility functions for updating or examining the state_change_fn or state_change_data fields. Instead, it is expected that the fields in the usbs_control_endpoint data structure will be manipulated directly. Any utility functions would do just this, but at the cost of increased code and cpu overheads.

Standard Control Messages

typedef struct usbs_control_endpoint {
    ...
    unsigned char control_buffer[8];
    usbs_control_return (*standard_control_fn)(struct usbs_control_endpoint*, void*);
    void* standard_control_data;
    ...
} usbs_control_endpoint;

typedef enum {
    USBS_CONTROL_RETURN_HANDLED = 0,
    USBS_CONTROL_RETURN_UNKNOWN = 1,
    USBS_CONTROL_RETURNSTALL = 2
} usbs_control_return;

extern usbs_control_return usbs_handle_standard_control(struct usbs_control_endpoint*);

When a USB peripheral is connected to the host it must always respond to control messages sent to endpoint 0. Control messages always consist of an initial eight-byte header, containing fields such as a request type. This may be followed by a further data transfer, either from host to peripheral or from peripheral to host. The way this is handled is described in the Buffer Management section below.
**Control Endpoints**

The USB device driver will always accept the initial eight-byte header, storing it in the `control_buffer` field. Then it determines the request type: standard, class, vendor, or reserved. The way in which the last three of these are processed is described in the section Other Control Messages. Some standard control messages will be handled by the device driver itself; typically the `set-address` request and the `get-status`, `set-feature` and `clear-feature` requests when applied to endpoints.

If a standard control message cannot be handled by the device driver itself, the driver checks the `standard_control_fn` field in the control endpoint data structure. If higher-level code has installed a suitable callback function then this will be invoked with two argument, the control endpoint data structure itself and the `standard_control_data` field. The latter allows the higher level code to associate arbitrary data with the control endpoint. The callback function can return one of three values: `HANDLED` to indicate that the request has been processed; `UNKNOWN` if the message should be handled by the default code; or `STALL` to indicate an error condition. If higher level code has not installed a callback function or if the callback function has returned `UNKNOWN` then the device driver will invoke a default handler, `usbs_handle_standard_control` provided by the common USB slave package.

The default handler can cope with all of the standard control messages for a simple USB peripheral. However, if the peripheral involves multiple configurations, multiple interfaces in a configuration, or alternate settings for an interface, then this cannot be handled by generic code. For example, a multimedia peripheral may support various alternate settings for a given data source with different bandwidth requirements, and the host can select a setting that takes into account the current load. Clearly higher-level code needs to be aware when the host changes the current setting, so that it can adjust the rate at which data is fed to or retrieved from the host. Therefore the higher-level code needs to install its own standard control callback and process appropriate messages, rather than leaving these to the default handler.

The default handler will take care of the `get-descriptor` request used to obtain the enumeration data. It has support for string descriptors but ignores language encoding issues. If language encoding is important for the peripheral then this will have to be handled by an application-specific standard control handler.

The header file `<cyg/io/usb/usb.h>` defines various constants related to control messages, for example the function codes corresponding to the standard request types. This header file is provided by the common USB package, not by the USB slave package, since the information is also relevant to USB hosts.

**Other Control Messages**

```c
typedef struct usbs_control_endpoint {
    ...
    usbs_control_return (*class_control_fn)(struct usbs_control_endpoint*, void*);
    void* class_control_data;
    usbs_control_return (*vendor_control_fn)(struct usbs_control_endpoint*, void*);
    void* vendor_control_data;
    usbs_control_return (*reserved_control_fn)(struct usbs_control_endpoint*, void*);
    void* reserved_control_data;
    ...
} usbs_control_endpoint;
```

Non-standard control messages always have to be processed by higher-level code. This could be class-specific packages. For example, the USB-ethernet package will handle requests for getting the MAC address and for enabling or disabling promiscuous mode. In all cases the device driver will store the initial request in the `control_buffer` field, check for an appropriate handler, and invoke it with details of the control endpoint and any handler-specific data that has been installed alongside the handler itself. The handler should return either `USBS_CONTROL_RETURN_HANDLED` to report success or `USBS_CONTROL_RETURNSTALL` to report failure. The device driver will report this to the host.

If there are multiple parties interested in a particular type of control messages, it is the responsibility of application code to install an appropriate handler and process the requests appropriately.
Buffer Management

typedef struct usbs_control_endpoint {
    ...  
    unsigned char* buffer;
    int buffer_size;
    void (*fill_buffer_fn)(struct usbs_control_endpoint*);
    void* fill_data;
    int fill_index;
    usbs_control_return (*complete_fn)(struct usbs_control_endpoint*, int);
    ...  
} usbs_control_endpoint;

Many USB control messages involve transferring more data than just the initial eight-byte header. The header indicates the direction of the transfer, OUT for host to peripheral or IN for peripheral to host. It also specifies a length field, which is exact for an OUT transfer or an upper bound for an IN transfer. Control message handlers can manipulate six fields within the control endpoint data structure to ensure that the transfer happens correctly.

For an OUT transfer, the handler should examine the length field in the header and provide a single buffer for all the data. A class-specific protocol would typically impose an upper bound on the amount of data, allowing the buffer to be allocated statically. The handler should update the buffer and complete_fn fields. When all the data has been transferred the completion callback will be invoked, and its return value determines the response sent back to the host. The USB standard allows for a new control message to be sent before the current transfer has completed, effectively cancelling the current operation. When this happens the completion function will also be invoked. The second argument to the completion function specifies what has happened, with a value of 0 indicating success and an error code such as -EPIPE or -EIO indicating that the current transfer has been cancelled.

IN transfers are a little bit more complicated. The required information, for example the enumeration data, may not be in a single contiguous buffer. Instead a mechanism is provided by which the buffer can be refilled, thus allowing the transfer to move from one record to the next. Essentially, the transfer operates as follows:

1. When the host requests another chunk of data (typically eight bytes), the USB device driver will examine the buffer_size field. If non-zero then buffer contains at least one more byte of data, and then buffer_size is decremented.
2. When buffer_size has dropped to 0, the fill_buffer_fn field will be examined. If non-null it will be invoked to refill the buffer.
3. The fill_data and fill_index fields are not used by the device driver. Instead these fields are available to the refill function to keep track of the current state of the transfer.
4. When buffer_size is 0 and fill_buffer_fn is NULL, no more data is available and the transfer has completed.
5. Optionally a completion function can be installed. This will be invoked with 0 if the transfer completes successfully, or with an error code if the transfer is cancelled because of another control message.

If the requested data is contiguous then the only fields that need to be manipulated are buffer and buffer_size, and optionally complete_fn. If the requested data is not contiguous then the initial control message handler should update fill_buffer_fn and some or all of the other fields, as required. An example of this is the handling of the standard get-descriptor control message by usbs_handle_standard_control.

Polling Support

typedef struct usbs_control_endpoint {
    void (*poll_fn)(struct usbs_control_endpoint*);
    ...  
} usbs_control_endpoint;
Control Endpoints

int interrupt_vector;
...
} usbs_control_endpoint;

In nearly all circumstances USB I/O should be interrupt-driven. However, there are special environments such as RedBoot where polled operation may be appropriate. If the device driver can operate in polled mode then it will provide a suitable function via the poll_fn field, and higher-level code can invoke this regularly. This polling function will take care of all endpoints associated with the device, not just the control endpoint. If the USB hardware involves a single interrupt vector then this will be identified in the data structure as well.
Data Endpoints

Name

Data Endpoints — Data endpoint data structures

Synopsis

```c
#include <cyg/io/usb/usbs.h>

typedef struct usbs_rx_endpoint {
    void (*start_rx_fn)(struct usbs_rx_endpoint*);
    void (*set_halted_fn)(struct usbs_rx_endpoint*, cyg_bool);
    void (*complete_fn)(void*, int);
    void* complete_data;
    unsigned char* buffer;
    int buffer_size;
    cyg_bool halted;
} usbs_rx_endpoint;

typedef struct usbs_tx_endpoint {
    void (*start_tx_fn)(struct usbs_tx_endpoint*);
    void (*set_halted_fn)(struct usbs_tx_endpoint*, cyg_bool);
    void (*complete_fn)(void*, int);
    void* complete_data;
    const unsigned char* buffer;
    int buffer_size;
    cyg_bool halted;
} usbs_tx_endpoint;
```

Receive and Transmit Data Structures

In addition to a single usbs_control_endpoint data structure per USB slave device, the USB device driver should also provide receive and transmit data structures corresponding to the other endpoints. The names of these are determined by the device driver. For example, the SA1110 USB device driver package provides `usbs_sa11x0_ep1` for receives and `usbs_sa11x0_ep2` for transmits.

Unlike control endpoints, the common USB slave package does provide a number of utility routines to manipulate data endpoints. For example `usbs_start_rx_buffer` can be used to receive data from the host into a buffer. In addition the USB device driver can provide devtab entries such as `/dev/usbs1r` and `/dev/usbs2w`, so higher-level code can open these devices and then perform blocking `read` and `write` operations.

However, the operation of data endpoints and the various endpoint-related functions is relatively straightforward. First consider a usbs_rx_endpoint structure. The device driver will provide the members `start_rx_fn` and `set_halted_fn`, and it will maintain the `halted` field. To receive data, higher-level code sets the `buffer`, `buffer_size`, `complete_fn` and optionally the `complete_data` fields. Next the `start_rx_fn` member should be called. When the transfer has finished the device driver will invoke the completion function, using `complete_data` as the first argument and a size field for the second argument. A negative size indicates an error of some sort: `-EGAIN` indicates that the endpoint has been halted, usually at the request of the host; `-EPIPE` indicates that the connection between the host and the peripheral has been broken. Certain device drivers may generate other error codes.
Data Endpoints

If higher-level code needs to halt or unhalt an endpoint then it can invoke the `set_halted_fn` member. When an endpoint is halted, invoking `start_rx_fn` with `buffer_size` set to 0 indicates that higher-level code wants to block until the endpoint is no longer halted; at that point the completion function will be invoked.

USB device drivers are allowed to assume that higher-level protocols ensure that host and peripheral agree on the amount of data that will be transferred, or at least on an upper bound. Therefore there is no need for the device driver to maintain its own buffers, and copy operations are avoided. If the host sends more data than expected then the resulting behaviour is undefined.

Transmit endpoints work in essentially the same way as receive endpoints. Higher-level code should set the `buffer` and `buffer_size` fields to point at the data to be transferred, then call `start_tx_fn`, and the device driver will invoked the completion function when the transfer has completed.

USB device drivers are not expected to perform any locking. If at any time there are two concurrent receive operations for a given endpoint, or two concurrent transmit operations, then the resulting behaviour is undefined. It is the responsibility of higher-level code to perform any synchronisation that may be necessary. In practice, conflicts are unlikely because typically a given endpoint will only be accessed sequentially by just one part of the overall system.
Writing a USB Device Driver

Name

Writing a USB Device Driver — USB Device Driver Porting Guide

Introduction

Often the best way to write a USB device driver will be to start with an existing one and modify it as necessary. The information given here is intended primarily as an outline rather than as a complete guide.

Note: At the time of writing only one USB device driver has been implemented. Hence it is possible, perhaps probable, that some portability issues have not yet been addressed. One issue involves the different types of transfer, for example the initial target hardware had no support for isochronous or interrupt transfers, so additional functionality may be needed to switch between transfer types. Another issue would be hardware where a given endpoint number, say endpoint 1, could be used for either receiving or transmitting data, but not both because a single fifo is used. Issues like these will have to be resolved as and when additional USB device drivers are written.

The Control Endpoint

A USB device driver should provide a single usbs_control_endpoint data structure for every USB device. Typical peripherals will have only one USB port so there will be just one such data structure in the entire system, but theoretically it is possible to have multiple USB devices. These may all involve the same chip, in which case a single device driver should support multiple device instances, or they may involve different chips. The name or names of these data structures are determined by the device driver, but appropriate care should be taken to avoid name clashes.

A USB device cannot be used unless the control endpoint data structure exists. However, the presence of USB hardware in the target processor or board does not guarantee that the application will necessarily want to use that hardware. To avoid unwanted code or data overheads, the device driver can provide a configuration option to determine whether or not the endpoint 0 data structure is actually provided. A default value of CYGINT_IO_USB_SLAVE_CLIENTS ensures that the USB driver will be enabled automatically if higher-level code does require USB support, while leaving ultimate control to the user.

The USB device driver is responsible for filling in the start_fn, poll_fn and interrupt_vector fields. Usually this can be achieved by static initialization. The driver is also largely responsible for maintaining the state field. The control_buffer array should be used to hold the first packet of a control message. The buffer and other fields related to data transfers will be managed jointly by higher-level code and the device driver. The remaining fields are generally filled in by higher-level code, although the driver should initialize them to NULL values.

Hardware permitting, the USB device should be inactive until the start_fn is invoked, for example by tristating the appropriate pins. This prevents the host from interacting with the peripheral before all other parts of the system have initialized. It is expected that the start_fn will only be invoked once, shortly after power-up.

Where possible the device driver should detect state changes, such as when the connection between host and peripheral is established, and report these to higher-level code via the state_change_fn callback, if any. The state change to and from configured state cannot easily be handled by the device driver itself, instead higher-level code such as the common USB slave package will take care of this.
Once the connection between host and peripheral has been established, the peripheral must be ready to accept control messages at all times, and must respond to these within certain time constraints. For example, the standard set-address control message must be handled within 50ms. The USB specification provides more information on these constraints. The device driver is responsible for receiving the initial packet of a control message. This packet will always be eight bytes and should be stored in the control_buffer field. Certain standard control messages should be detected and handled by the device driver itself. The most important is set-address, but usually the get-status, set-feature and clear-feature requests when applied to halted endpoints should also be handled by the driver. Other standard control messages should first be passed on to the standard_control_fn callback (if any), and finally to the default handler usbs_handle_standard_control provided by the common USB slave package. Class, vendor and reserved control messages should always be dispatched to the appropriate callback and there is no default handler for these.

Some control messages will involve further data transfer, not just the initial packet. The device driver must handle this in accordance with the USB specification and the buffer management strategy. The driver is also responsible for keeping track of whether or not the control operation has succeeded and generating an ACK or STALL handshake.

The polling support is optional and may not be feasible on all hardware. It is only used in certain specialised environments such as RedBoot. A typical implementation of the polling function would just check whether or not an interrupt would have occurred and, if so, call the same code that the interrupt handler would.

Data Endpoints

In addition to the control endpoint data structure, a USB device driver should also provide appropriate data endpoint data structures. Obviously this is only relevant if the USB support generally is desired, that is if the control endpoint is provided. In addition, higher-level code may not require all the endpoints, so it may be useful to provide configuration options that control the presence of each endpoint. For example, the intended application might only involve a single transmit endpoint and of course control messages, so supporting receive endpoints might waste memory.

Conceptually, data endpoints are much simpler than the control endpoint. The device driver has to supply two functions, one for data transfers and another to control the halted condition. These implement the functionality for usbs_start_rx_buffer, usbs_start_tx_buffer, usbs_set_rx_endpoint_halted and usbs_set_tx_endpoint_halted. The device driver is also responsible for maintaining the halted status.

For data transfers, higher-level code will have filled in the buffer, buffer_size, complete_fn and complete_data fields. The transfer function should arrange for the transfer to start, allowing the host to send or receive packets. Typically this will result in an interrupt at the end of the transfer or after each packet. Once the entire transfer has been completed, the driver’s interrupt handling code should invoke the completion function. This can happen either in DSR context or thread context, depending on the driver’s implementation. There are a number of special cases to consider. If the endpoint is halted when the transfer is started then the completion function can be invoked immediately with -EAGAIN. If the transfer cannot be completed because the connection is broken then the completion function should be invoked with -EPIPE. If the endpoint is stalled during the transfer, either because of a standard control message or because higher-level code calls the appropriate set_halted_fn, then again the completion function should be invoked with -EAGAIN. Finally, the <usbs_start_rx_endpoint_wait and usbs_start_tx_endpoint_wait functions involve calling the device driver’s data transfer function with a buffer size of 0 bytes.

Note: Giving a buffer size of 0 bytes a special meaning is problematical because it prevents transfers of that size. Such transfers are allowed by the USB protocol, consisting of just headers and acknowledgements and an empty data phase, although rarely useful. A future modification of the device driver specification will address this issue, although care has to be taken that the functionality remains accessible through devtab entries as well as via low-level accesses.
Devtab Entries

For some applications or higher-level packages it may be more convenient to use traditional open/read/write I/O calls rather than the non-blocking USB I/O calls. To support this the device driver can provide a devtab entry for each endpoint, for example:

```c
#ifdef CYGVAR_DEVS_USB_SA11X0_EPI1_DEVTAB_ENTRY

static CHAR_DEVIO_TABLE(usbs_sa11x0_epi1_devtab_functions, 
    &cyg_devio_cwrite, 
    &usbs_devtab_cread, 
    &cyg_devio_bwrite, 
    &cyg_devio_bread, 
    &cyg_devio_select, 
    &cyg_devio_get_config, 
    &cyg_devio_set_config);

static CHAR_DEVTAB_ENTRY(usbs_sa11x0_epi1_devtab_entry, 
    CYGDAT_DEVS_USB_SA11X0_DEVTAB_BASENAME "1r", 
    0, 
    &usbs_sa11x0_epi1_devtab_functions, 
    &usbs_sa11x0_devtab_dummy_init, 
    0, 
    (void*) &usbs_sa11x0_epi1);

#endif
```

Again care must be taken to avoid name clashes. This can be achieved by having a configuration option to control the base name, with a default value of e.g. /dev/usbs, and appending an endpoint-specific string. This gives the application developer sufficient control to eliminate any name clashes. The common USB slave package provides functions usbs_devtab_cwrite and usbs_devtab_cread, which can be used in the function tables for transmit and receive endpoints respectively. The private field priv of the devtab entry should be a pointer to the underlying endpoint data structure.

Because devtab entries are never accessed directly, only indirectly, they would usually be eliminated by the linker. To avoid this the devtab entries should normally be defined in a separate source file which ends up the special library libextras.a rather than in the default library libtarget.a.

Not all applications or higher-level packages will want to use the devtab entries and the blocking I/O facilities. It may be appropriate for the device driver to provide additional configuration options that control whether or not any or all of the devtab entries should be provided, to avoid unnecessary memory overheads.

Interrupt Handling

A typical USB device driver will need to service interrupts for all of the endpoints and possibly for additional USB events such as entering or leaving suspended mode. Usually these interrupts need not be serviced directly by the ISR. Instead, they can be left to a DSR. If the peripheral is not able to accept or send another packet just yet, the hardware will generate a NAK and the host will just retry a little bit later. If high throughput is required then it may be desirable to handle the bulk transfer protocol largely at ISR level, that is take care of each packet in the ISR and only activate the DSR once the whole transfer has completed.

Control messages may involve invoking arbitrary callback functions in higher-level code. This should normally happen at DSR level. Doing it at ISR level could seriously affect the system’s interrupt latency and impose
unacceptable constraints on what operations can be performed by those callbacks. If the device driver requires a thread anyway then it may be appropriate to use this thread for invoking the callbacks, but usually it is not worthwhile to add a new thread to the system just for this; higher-level code is expected to write callbacks that function sensibly at DSR level. Much the same applies to the completion functions associated with data transfers. These should also be invoked at DSR or thread level.

**Support for USB Testing**

Optionally a USB device driver can provide support for the USB test software. This requires defining a number of additional data structures, allowing the generic test code to work out just what the hardware is capable of and hence what testing can be performed.

The key data structure is `usbs_testing_endpoint`, defined in `cyg/io/usb/usbs.h`. In addition some commonly required constants are provided by the common USB package in `cyg/io/usb/usb.h`. One `usbs_testing_endpoint` structure should be defined for each supported endpoint. The following fields need to be filled in:

- **endpoint_type**
  
  This specifies the type of endpoint and should be one of `USB_ENDPOINT_DESCRIPTOR_ATTR_CONTROL`, `BULK`, `ISOCRONOUS` or `INTERRUPT`.

- **endpoint_number**
  
  This identifies the number that should be used by the host to address this endpoint. For a control endpoint it should be 0. For other types of endpoints it should be between 1 and 15.

- **endpoint_direction**
  
  For control endpoints this field is irrelevant. For other types of endpoint it should be either `USB_ENDPOINT_DESCRIPTOR_ENDPOINT_IN` or `USB_ENDPOINT_DESCRIPTOR_ENDPOINT_OUT`. If a given endpoint number can be used for traffic in both directions then there should be two entries in the array, one for each direction.

- **endpoint**
  
  This should be a pointer to the appropriate `usbs_control_endpoint`, `usbs_rx_endpoint` or `usbs_tx_endpoint` structure, allowing the generic testing code to perform low-level I/O.

- **devtab_entry**
  
  If the endpoint also has an entry in the system's device table then this field should give the corresponding string, for example `"/dev/usbs1r"`. This allows the generic testing code to access the device via higher-level calls like `open` and `read`.

- **min_size**
  
  This indicates the smallest transfer size that the hardware can support on this endpoint. Typically this will be one.

**Note:** Strictly speaking a minimum size of one is not quite right since it is valid for a USB transfer to involve zero bytes, in other words a transfer that involves just headers and acknowledgements and an empty data phase, and that should be tested as well. However current device drivers interpret a transfer size of 0 as special, so that would have to be resolved first.
max_size

Similarly, this specifies the largest transfer size. For control endpoints the USB protocol uses only two bytes to hold the transfer length, so there is an upper bound of 65535 bytes. In practice it is very unlikely that any control transfers would ever need to be this large, and in fact such transfers would take a long time and probably violate timing constraints. For other types of endpoint any of the protocol, the hardware, or the device driver may impose size limits. For example a given device driver might be unable to cope with transfers larger than 65535 bytes. If it should be possible to transfer arbitrary amounts of data then a value of \(-1\) indicates no upper limit, and transfer sizes will be limited by available memory and by the capabilities of the host machine.

max_in_padding

This field is needed on some hardware where it is impossible to send packets of a certain size. For example the hardware may be incapable of sending an empty bulk packet to terminate a transfer that is an exact multiple of the 64-byte bulk packet size. Instead the driver has to do some padding and send an extra byte, and the host has to be prepared to receive this extra byte. Such a driver should specify a value of 1 for the padding field. For most drivers this field should be set to 0.

A better solution would be for the device driver to supply a fragment of Tcl code that would adjust the receive buffer size only when necessary, rather than for every transfer. Forcing receive padding on all transfers when only certain transfers will actually be padded reduces the accuracy of certain tests.

alignment

On some hardware data transfers may need to be aligned to certain boundaries, for example a word boundary or a cacheline boundary. Although in theory device drivers could hide such alignment restrictions from higher-level code by having their own buffers and performing appropriate copying, that would be expensive in terms of both memory and cpu cycles. Instead the generic testing code will align any buffers passed to the device driver to the specified boundary. For example, if the driver requires that buffers be aligned to a word boundary then it should specify an alignment value of 4.

The device driver should provide an array of these structures `usbs_testing_endpoints[]`. The USB testing code examines this array and uses the information to perform appropriate tests. Because different USB devices support different numbers of endpoints the number of entries in the array is not known in advance, so instead the testing code looks for a special terminator `USBS_TESTING_ENDPOINTS_TERMINATOR`. An example array, showing just the control endpoint and the terminator, might look like this:

```c
usbs_testing_endpoint usbs_testing_endpoints[] = {
    {
        endpoint_type : USB_ENDPOINT_DESCRIPTOR_ATTR_CONTROL,
        endpoint_number : 0,
        endpoint_direction : USB_ENDPOINT_DESCRIPTOR_ENDPOINT_IN,
        endpoint : (void*) &ep0.common,
        devtab_entry : (const char*) 0,
        min_size : 1,
        max_size : 0xFFFF,
        max_in_padding : 0,
        alignment : 0
    },
    ...,
    USBS_TESTING_ENDPOINTS_TERMINATOR
};
```
**Note:** The use of a single array `usbs_testing_endpoints` limits USB testing to platforms with a single USB device: if there were multiple devices, each defining their own instance of this array, then there would a collision at link time. In practice this should not be a major problem since typical USB peripherals only interact with a single host machine via a single slave port. In addition, even if a peripheral did have multiple slave ports the current USB testing code would not support this since it would not know which port to use.
Testing

Name

Testing — Testing of USB Device Drivers

Introduction

The support for USB testing provided by the eCos USB common slave package is somewhat different in nature from the kind of testing used in many other packages. One obvious problem is that USB tests cannot be run on just a bare target platform: instead the target platform must be connected to a suitable USB host machine, and that host machine must be running appropriate software for the test code to interact with. This is very different from say a kernel test which typically will have no external dependencies. Another important difference between USB testing and say a C library `strcmp` test is sensitivity to timing and to hardware boundary conditions: although a simple test case that just performs a small number of USB transfers is better than no testing at all, it should also be possible to run tests for hours or days on end, under a variety of loads. In order to provide the required functionality the basic architecture of the USB testing support is as follows:

1. There is a single target-side program usbtarget. By default when this is run on a target platform it will appear to do nothing. In fact it is waiting to be contacted by another program usbhost which will tell it what test or tests to run. usbtarget provides mechanisms for running a wide range of tests.

2. usbtarget is a generic program, but USB testing depends to some extent on the functionality provided by the hardware. For example there is no point in testing bulk transmits to endpoint 12 if the target hardware does not support an endpoint 12. Therefore each USB device driver should supply information about what the hardware is actually capable of, in the form of an array of usbs_testing_endpoint data structures.

3. There is a single host-side program usbhost, which acts as a counterpart to usbtarget. Again usbhost has no built-in knowledge of the test or tests that are supposed to run, it only provides mechanisms for running a wide range of tests. On start-up usbhost will search the USB bus for hardware running the target-side program, specifically a USB device that identifies itself as the product "Red Hat eCos USB test".

4. usbhost contains a Tcl interpreter, and will execute any Tcl scripts specified on the command line together with appropriate arguments. The Tcl interpreter has been extended with various commands such as `usbtest::bulktest`, so the script can perform the desired test or tests.

5. Adding a new test simply involves writing a short Tcl script that invokes the appropriate USB-specific commands. Running multiple tests involves passing appropriate arguments to usbhost, or alternatively writing a single script that just invokes other scripts.

The current implementation of usbhost depends heavily on functionality provided by the Linux kernel and in particular the usbdevfs support. It uses `/proc/bus/usb/devices` to find out what devices are attached to the bus, and will then access the device by opening `/proc/bus/usb/xxx/yyy` and performing `ioctl` operations. This allows USB testing to take place without having to write a new host-side device driver, but getting the code working on host machines not running Linux would obviously be problematical.

Building and Running the Target-side Code

The target-side component of the USB testing software consists of a single program usbtarget which contains support for a range of different tests, under the control of host-side software. This program is not built by default alongside other eCos test cases since it will only operate in certain environments, specifically when the target board’s connector is plugged into a Linux host, and when the appropriate host-side software has been
installed on that host. Instead the user must enable a configuration option CYGBLD_IO_USB_SLAVE_USBTEST to add the program to the list of tests for the current configuration.

Starting the usbtarget program does not require anything unusual, so it can be run in a normal gdb session just like any eCos application. After initialization the program will wait for activity from the host. Depending on the hardware, the Linux host will detect that a new USB peripheral is present on the bus either when the usbtarget initialization is complete or when the cable between target and host is connected. The host will perform the normal USB enumeration sequence and discover that the peripheral does not match any known vendor or product id and that there is no device driver for "Red Hat eCos USB test", so it will ignore the peripheral. When the usbhost program is run on the host it will connect to the target-side software, and testing can now commence.

Building and Running the Host-side Code

**Note:** In theory the host-side software should be built when the package is installed in the component repository, and removed when a package is uninstalled. The current eCos administration tool does not provide this functionality.

The host-side software should be built via the usual sequence of "configure/make/make install". It can only be built on a Linux host and the `configure` script contains an explicit test for this. Because the eCos component repository should generally be treated as a read-only resource the configure script will also prevent you from trying to build inside the source tree. Instead a separate build tree is required. Hence a typical sequence for building the host-side software would be as follows:

```
$ mkdir usbhost_build
$ cd usbhost_build
$ <repo>packages/io/usb/slave/current/host/configure <args>
$ make
$ su
$ make install
```

1. The location of the eCos component repository should be substituted for `<repo>`.
2. If the package has been obtained via CVS or anonymous CVS then the package version will be `current`, as per the example. If instead the package has been obtained as part of a full eCos release or as a separate `.epk` file then the appropriate package version should be used instead of `current`.
3. The `configure` script takes the usual arguments such as `--prefix` to specify where the executables and support files should be installed. The only other parameter that some users may wish to specify is the location of a suitable Tcl installation. By default usbhost will use the existing Tcl installation in `/usr`, as provided by your Linux distribution. An alternative Tcl installation can be specified using the parameter `--with-tcl`, or alternatively using some combination of `--with-tcl-include`, `--with-tcl-lib` and `--with-tcl-version`.
4. One of the host-side executables that gets built, usbchmod, needs to be installed with suid root privileges. Although the Linux kernel makes it possible for applications to perform low-level USB operations such as transmitting bulk packets, by default access to this functionality is restricted to programs with superuser privileges. It is undesirable to run a complex program such as usbhost with such privileges, especially since the program contains a general-purpose Tcl interpreter. Therefore when usbhost starts up and discovers that it does not have sufficient access to the appropriate entries in `/proc/bus/usb`, it spawns an instance of
Testing

usbcmod to modify the permissions on these entries. usbcmod will only do this for a USB device "Red Hat eCos USB test", so installing this program suid root should not introduce any security problems.

During make install the following actions will take place:

1. usbc will be installed in /usr/local/bin, or some other bin directory if the default location is changed at configure-time using a --prefix= or similar option. It will be installed as the executable usbc_<version>, for example usbc_current, thus allowing several releases of the USB slave package to co-exist. For convenience a symbolic link from usbc to this executable will be created, so users can just run usbc to access the most recently-installed version.

2. usbcmod will be installed in /usr/local/libexec/ecos/io_usb_slave_<version>. This program should only be run by usbc, not invoked directly, so it is not placed in the bin directory. Again the presence of the package version in the directory name allows multiple releases of the package to co-exist.

3. A Tcl script usbc is installed in the same directory as usbcmod. This Tcl script is loaded automatically by the usbc executable.

4. A number of additional Tcl scripts, for example list.tcl will get installed alongside usbc. These correspond to various test cases provided as standard. If a given test case is specified on the command line and cannot be found relative to the current directory then usbc will search the install directory for these test cases.

Note: Strictly speaking installing the usbc and other Tcl scripts below the libexec directory deviates from standard practice: they are architecture-independent data files so should be installed below the share subdirectory. In practice the files are sufficiently small that there is no point in sharing them, and keeping them below libexec simplifies the host-side software somewhat.

The usbc should be run only when there is a suitable target attached to the USB bus and running the usbtar program. It will search /proc/bus/usb/devices for an entry corresponding to this program, invoke usbcmod if necessary to change the access rights, and then interact with usbtar over the USB bus. usbc should be invoked as follows:

$ usbc [-v|--version] [-h|--help] [-V|--verbose] <test> [<test parameters>]

1. The -v or --version option will display version information for usbc including the version of the USB slave package that was used to build the executable.

2. The -h or --help option will display usage information.

3. The -V or --verbose option can be used to obtain more information at run-time, for example some output for every USB transfer. This option can be repeated multiple times to increase the amount of output.

4. The first argument that does not begin with a hyphen specifies a test that should be run, in the form of a Tcl script. For example an argument of list.tcl will cause usbc to look for a script with that name, adding a .tcl suffix if necessary, and run that script. usbc will look in the current directory first, then in the install tree for standard test scripts provided by the USB slave package.

5. Some test scripts may want their own parameters, for example a duration in seconds. These can be passed on the command line after the name of the test, for example usbc test mytest 60.


Writing a Test

Each test is defined by a Tcl script, running inside an interpreter provided by USBhost. In addition to the normal Tcl functionality this interpreter provides a number of variables and functions related to USB testing. For example, there is a variable `bulk_in_endpoints` that lists all the endpoints on the target that can perform bulk IN operations, and a related array `bulk_in` which contains information such as the minimum and maximum packets sizes. There is a function `bulktest` which can be used to perform bulk tests on a particular endpoint. A simple test script aimed at specific hardware could ignore the information variables since it would know exactly what USB hardware is available on the target, whereas a general-purpose script would use the information to adapt to the hardware capabilities.

To avoid namespace pollution all USB-related Tcl variables and functions live in the `usbtest::` namespace. Therefore accessing requires either explicitly including the namespace any references, for example `$usbtest::bulk_in_endpoints`, or by using Tcl’s namespace import facility.

A very simple test script might look like this:

```tcl
usbtest::bulktest 1 out 4000
usbtest::bulktest 2 in 4000
if { [usbtest::start 60] } {
    puts "Test successful"
} else
    puts "Test failed"
    foreach result $usbtest::results {
        puts $result
    }
}
```

This would perform a test run involving 4000 bulk transfers from the host to the target’s endpoint 1, and concurrently 4000 bulk transfers from endpoint 2. Default settings for packet sizes, contents, and delays would be used. The actual test would not start running until `usbtest` is invoked, and it is expected that the test would complete within 60 seconds. If any failures occur then they are reported.

Available Hardware

Each target-side USB device driver provides information about the actual capabilities of the hardware, for example which endpoints are available. Strictly speaking it provides information about what is actually supported by the device driver, which may be a subset of what the hardware is capable of. For example, the hardware may support isochronous transfers on a particular endpoint but if there is no software support for this in the driver then this endpoint will not be listed. When USBhost first contacts the usbtarget program running on the target platform, it obtains this information and makes it available to test scripts via Tcl variables:

`bulk_in_endpoints`

This is a simple list of the endpoints which can support bulk IN transfers. For example if the target-side hardware supports these transfers on endpoints 3 and 5 then the value would be "3 5". Typical test scripts would iterate over the list using something like:

```tcl
if { 0 != [llength $usbtest::bulk_in_endpoints] } {
    puts "Bulk IN endpoints: $usbtest::bulk_in_endpoints"
    foreach endpoint $usbtest::bulk_in_endpoints {
        ...
    }
}
```
testing

bulk_in()

This array holds additional information about each bulk IN endpoint. The array is indexed by two fields, the endpoint number and one of min_size, max_size, max_in_padding and devtab:

min_size

This field specifies a lower bound on the size of bulk transfers, and will typically will have a value of 1.

Note: The typical minimum transfer size of a single byte is not strictly speaking correct, since under some circumstances it can make sense to have a transfer size of zero bytes. However current target-side device drivers interpret a request to transfer zero bytes as a way for higher-level code to determine whether or not an endpoint is stalled, so it is not actually possible to perform zero-byte transfers. This issue will be addressed at some future point.

max_size

This field specifies an upper bound on the size of bulk transfers. Some target-side drivers may be limited to transfers of say 0xFFFF bytes because of hardware limitations. In practice the transfer size is likely to be limited primarily to limit memory consumption of the test code on the target hardware, and to ensure that tests complete reasonably quickly. At the time of writing transfers are limited to 4K.

max_in_padding

On some hardware it may be necessary for the target-side device driver to send more data than is actually intended. For example the SA11x0 USB hardware cannot perform bulk transfers that are an exact multiple of 64 bytes, instead it must pad such transfers with an extra byte and the host must be ready to accept and discard this byte. The max_in_padding field indicates the amount of padding that is required. The low-level code inside usbhost will use this field automatically, and there is no need for test scripts to adjust packet sizes for padding. The field is provided for informational purposes only.

devtab

This is a string indicating whether or not the target-side USB device driver supports access to this endpoint via entries in the device table, in other words through conventional calls like open and write. Some device drivers may only support low-level USB access because typically that is what gets used by USB class-specific packages such as USB-ethernet. An empty string indicates that no devtab entry is available, otherwise it will be something like */dev/usbs2w".

Typical test scripts would access this data using something like:

```bash
foreach endpoint $usbtest:bulk_in_endpoints {
  puts "Endpoint $endpoint: "
  puts " minimum transfer size $usbtest::bulk_in($endpoint,min_size)"
  puts " maximum transfer size $usbtest::bulk_in($endpoint,max_size)"
  if { 0 == $usbtest::bulk_in($endpoint,max_in_padding) } {
    puts " no IN padding required"
  } else {
    puts " $usbtest::bulk_in($endpoint,max_in_padding) bytes of IN padding required"
  }
  if { "" == $usbtest::bulk_in($endpoint,devtab) } {
    puts " no devtab entry provided"
  } else {
```
Testing Bulk Transfers

The main function for initiating a bulk test is `usbtest::bulktest`. This takes three compulsory arguments, and can be given a number of additional arguments to control the exact behaviour. The compulsory arguments are:

**endpoint**

This specifies the endpoint to use. It should correspond to one of the entries in `usbtest::bulk_in_endpoints` or `usbtest::bulk_out_endpoints`, depending on the transfer direction.

**direction**

This should be either `in` or `out`.

---

puts "    corresponding devtab entry is \$usbtest::bulk_in\$(endpoint,devtab)"
}
}

bulk_out_endpoint

This is a simple list of the endpoints which can support bulk OUT transfers. It is analogous to `bulk_in_endpoints`.

bulk_out()

This array holds additional information about each bulk OUT endpoint. It can be accessed in the same way as `bulk_in()`, except that there is no `max_in_padding` field because that field only makes sense for IN transfers.

cancel()

This array holds information about the control endpoint. It contains two fields, `min_size` and `max_size`. Note that there is no variable `cancel_endpoints` because a USB target always supports a single control endpoint 0. Similarly the `control` array does not use an endpoint number as the first index because that would be redundant.

isochronous_in_endpoints and isochronous_in()

These variables provide the same information as `bulk_in_endpoints` and `bulk_in`, but for endpoints that support isochronous IN transfers.

isochronous_out_endpoints and isochronous_out()

These variables provide the same information as `bulk_out_endpoints` and `bulk_out`, but for endpoints that support isochronous OUT transfers.

interrupt_in_endpoints and interrupt_in()

These variables provide the same information as `bulk_in_endpoints` and `bulk_in`, but for endpoints that support interrupt IN transfers.

interrupt_out_endpoints and interrupt_out()

These variables provide the same information as `bulk_out_endpoints` and `bulk_out`, but for endpoints that support interrupt OUT transfers.
Testing

number of transfers

This specifies the number of transfers that should take place. The testing software does not currently support the concept of performing transfers for a given period of time because synchronising this on both the host and a wide range of targets is difficult. However it is relatively easy to work out the approximate time a number of bulk transfers should take place, based on a typical bandwidth of 1MB/second and assuming a 1ms overhead per transfer. Alternatively a test script could perform a small initial run to determine what performance can actually be expected from a given target, and then use this information to run a much longer test.

Additional arguments can be used to control the exact transfer. For example a `txdelay+` argument can be used to slowly increase the delay between transfers. All such arguments involve a value which can be passed either as part of the argument itself, for example `txdelay+5`, or as a subsequent argument, `txdelay+5`. The possible arguments fall into a number of categories: data, I/O mechanism, transmit size, receive size, transmit delay, and receive delay.

Data

An obvious parameter to control is the actual data that gets sent. This can be controlled by the argument `data` which can take one of five values: `none`, `bytefill`, `intfill`, `byteseq` and `wordseq`. The default value is `none`.

none

The transmit code will not attempt to fill the buffer in any way, and the receive code will not check it. The actual data that gets transferred will be whatever happened to be in the buffer before the transfer started.

bytefill

The entire buffer will be filled with a single byte, as per `memset`.

intfill

The buffer will be treated as an array of 32-bit integers, and will be filled with the same integer repeated the appropriate number of times. If the buffer size is not a multiple of four bytes then the last few bytes will be set to 0.

byteseq

The buffer will be filled with a sequence of bytes, generated by a linear congruential generator. If the first byte in the buffer is filled with the value $x$, the next byte will be $(m x) + i$. For example a sequence of slowly incrementing bytes can be achieved by setting both the multiplier and the increment to 1. Alternatively a pseudo-random number sequence can be achieved using values 1103515245 and 12345, as per the standard C library `rand` function. For convenience these two constants are available as Tcl variables `usbtest::MULTIPLIER` and `usbtest::INCREMENT`.

wordseq

This acts like `byteseq`, except that the buffer is treated as an array of 32-bit integers rather than as an array of bytes. If the buffer is not a multiple of four bytes then the last few bytes will be filled with zeroes.

The above requires three additional parameters `data1`, `data*` and `data+`. `data1` specifies the value to be used for byte or word fills, or the first number when calculating a sequence. The default value is 0. `data*` and `data+` specify the multiplier and increment for a sequence, and have default values of 1 and 0 respectively. For example, to perform a bulk transfer of a pseudo-random sequence of integers starting with 42 the following code could be used:
The above parameters define what data gets transferred for the first transfer, but a test can involve multiple transfers. The data format will be the same for all transfers, but it is possible to adjust the current value, the multiplier, and the increment between each transfer. This is achieved with parameters \( \text{data}^*, \text{data}+, \text{data}**, \text{data}^+, \text{data}++ \), with default values of 1 for each multiplier and 0 for each increment. For example, if the multiplier for the first transfer is set to 2 using \( \text{data}^* \), and arguments \( \text{data}** 2 \) and \( \text{data}^+ -1 \) are also supplied, then the multiplier for subsequent transfers will be 3, 5, 9, ....

Note: Currently it is not possible for a test script to send specific data, for example a specific sequence of bytes captured by a protocol analyser that caused a problem. If the transfer was from host to target then the target would have to know the exact sequence of bytes to expect, which means transferring data over the USB bus when that data is known to have caused problems in the past. Similarly for target to host transfers the target would have to know what bytes to send. A possible future extension of the USB testing support would allow for bounce operations, where a given message is first sent to the target and then sent back to the host, with only the host checking that the data was returned correctly.

I/O Mechanism

On the target side USB transfers can happen using either low-level USB calls such as \texttt{usbs_start_rx_buffer}, or by higher-level calls which go through the device table. By default the target-side code will use the low-level calls. If it is desired to test the higher-level calls instead, for example because those are what the application uses, then that can be achieved with an argument \texttt{mechanism=devtab}.

Transmit Size

The next set of arguments can be used to control the size of the transmitted buffer: \( \text{txsize1, txsize}>=, \text{txsize}<, \text{txsize}/, \text{and txsize}+. \)

\( \text{txsize1} \) determines the size of the first transfer, and has a default value of 32 bytes. The size of the next transfer is calculated by first multiplying by the \( \text{txsize}^* \) value, then dividing by the \( \text{txsize}/ \) value, and finally adding the \( \text{txsize}+ \) value. The defaults for these are 1, 1, and 0 respectively, which means that the transfer size will remain unchanged. If for example the transfer size should increase by approximately 50 per cent each time then suitable values might be \( \text{txsize}^* 3, \text{txsize}/ 2, \text{and txsize}+ 1 \).

The \( \text{txsize}>= \) and \( \text{txsize}< \) arguments can be used to impose lower and upper bounds on the transfer. By default the \texttt{min_size} and \texttt{max_size} values appropriate for the endpoint will be used. If at any time the current size falls outside the bounds then it will be normalized.

Receive Size

The receive size, in other words the number of bytes that either host or target will expect to receive as opposed to the number of bytes that actually get sent, can be adjusted using a similar set of arguments: \( \text{rxsize1, rxsize}>=, \text{rxsize}<, \text{rxsize}/, \text{and rxsize}+. \) The current receive size will be adjusted between transfers just like the transmit size. However when communicating over USB it is not a good idea to attempt to receive less data than will actually be sent: typically neither the hardware nor the software will be able to do anything useful with the excess, so there will be problems. Therefore if at any time the calculated receive size is less than the transmit size, the actual receive will be for the exact number of bytes that will get transmitted. However this will not affect the calculations for the next receive size.
Testing

The default values for $\text{rxsize1}$, $\text{rxsize*}$, $\text{rxsize/}$ and $\text{rxsize+}$ are 0, 1, 1 and 0 respectively. This means that the calculated receive size will always be less than the transmit size, so the receive operation will be for the exact number of bytes transmitted. For some USB protocols this would not accurately reflect the traffic that will happen. For example with USB-ethernet transfer sizes will vary between 16 and 1516 bytes, so the receiver will always expect up to 1516 bytes. This can be achieved using $\text{rxsize1 1516}$, leaving the other parameters at their default values.

For target hardware which involves non-zero $\text{max\_in\_padding}$, on the host side the padding will be added automatically to the receive size if necessary.

Transmit and Receive Delays

Typically during the testing there will be some minor delays between transfers on both host and target. Some of these delays will be caused by timeslicing, for example another process running on the host, or a concurrent test thread running inside the target. Other delays will be caused by the USB bus itself, for example activity from another device on the bus. However it is desirable that test cases be allowed to inject additional and somewhat more controlled delays into the system, for example to make sure that the target behaves correctly even if the target is not yet ready to receive data from the host.

The transmit delay is controlled by six parameters: $\text{txdelay1}$, $\text{txdelay*}$, $\text{txdelay/}$, $\text{txdelay+}$ and $\text{txdelay}=$. The default values for these are 0, 1, 1, 0 and 1000000000 respectively, so that by default transmits will happen as quickly as possible. Delays are measured in nanoseconds, so a value of 1000000 would correspond to a delay of 0.001 seconds or one millisecond. By default delays have an upper bound of one second. Between transfers the transmit delay is updated in much the same was as the transfer sizes.

The receive delay is controlled by a similar set of six parameters: $\text{rxdelay1}$, $\text{rxdelay*}$, $\text{rxdelay/}$, $\text{rxdelay+}$, $\text{rxdelay}=$ and $\text{rxdelay}=$. The default values for these are the same as for transmit delays.

The transmit delay is used on the side which sends data over the USB bus, so for a bulk IN transfer it is the target that sends data and hence sleeps for the specified transmit delay, while the host receives data sleeps for the receive delay. For an OUT transfer the positions are reversed.

It should be noted that although the delays are measured in nanoseconds, the actual delays will be much less precise and are likely to be of the order of milliseconds. The exact details will depend on the kernel clock speed.

Other Types of Transfer

Support for testing other types of USB traffic such as isochronous transfers is not yet implemented.

Starting a Test and Collecting Results

A USB test script should prepare one or more transfers using appropriate functions such as $\text{usbtest::bulktest}$. Once all the individual tests have been prepared they can be started by a call to $\text{usbtest::start}$. This takes a single argument, a maximum duration measured in seconds. If all transfers have not been completed in the specified time then any remaining transfers will be aborted.

$\text{usbtest::start}$ will return 1 if all the tests have succeeded, or 0 if any of them have failed. More detailed reports will be stored in the Tcl variable $\text{usbtests::results}$, which will be a list of string messages.
Testing

Existing Test Scripts

A number of test scripts are provided as standard. These are located in the host subdirectory of the common USB slave package, and will be installed as part of the process of building the host-side software. When a script is specified on the command line usbhost will first search for it in the current directory, then in the install tree. Standard test scripts include the following:

list.tcl

This script simply displays information about the capabilities of the target platform, as provided by the target-side USB device driver. It can help with tracking down problems, but its primary purpose is to let users check that everything is working correctly: if running usbhost list.tcl outputs sensible information then the user knows that the target side is running correctly and that communication between host and target is possible.

verbose.tcl

The target-side code can provide information about what is happening while tests are prepared and run. This facility should not normally be used since the extra I/O involved will significantly affect the behaviour of the system, but in some circumstances it may prove useful. Since an eCos application cannot easily be given command-line arguments the target-side verbosity level cannot be controlled using -V or --verbose options. Instead it can be controlled from inside gdb by changing the integer variable verbose. Alternatively it can be manipulated by running the test script verbose.tcl. This script takes a single argument, the desired verbosity level, which should be a small integer. For example, to disable target-side run-time logging the command usbhost verbose 0 can be used.

bulk-boundaries.tcl

This script performs simple bulk IN and OUT transfers of different sizes around interesting boundaries. This test is useful to ensure the driver correctly handles the case where a transfer is just smaller than, the same size as, and just bigger than the hardware buffer in the endpoint hardware. This script takes no parameters. It determines what endpoints the device has by asking it.

Possible Problems

If all transfers succeed within the specified time then both host and target remain in synch and further tests can be run without problem. However, if at any time a failure occurs then things get more complicated. For example, if the current test involves a series of bulk OUT transfers and the target detects that for one of these transfers it received less data than was expected then the test has failed, and the target will stop accepting data on this endpoint. However the host-side software may not have detected anything wrong and is now blocked trying to send the next lot of data.

The test code goes to considerable effort to recover from problems such as these. On the host-side separate threads are used for concurrent transfers, and on the target-side appropriate asynchronous I/O mechanisms are used. In addition there is a control thread on the host that checks the state of all the main host-side threads, and the state of the target using private control messages. If it discovers that one side has stopped sending or receiving data because of an error and the other side is blocked as a result, it will set certain flags and then cause one additional transfer to take place. That additional transfer will have the effect of unblocking the other side, which then discovers that an error has occurred by checking the appropriate flags. In this way both host and target should end up back in synch, and it is possible to move on to the next set of tests.

However, the above assumes that the testing has not triggered any serious hardware conditions. If instead the target-side hardware has been left in some strange state so that, for example, it will no longer raise an interrupt
Testing

for traffic on a particular endpoint then recovery is not currently possible, and the testing software will just hang.

A possible future enhancement to the testing software would allow the host-side to raise a USB reset signal whenever a failure occurs, in the hope that this would clear any remaining problems within the target-side USB hardware.
Testing
XXXIV. eCos Support for Developing USB-ethernet Peripherals
Testing
Introduction

Name

Introduction — eCos support for developing USB ethernet peripherals

Introduction

The eCos USB-ethernet package provides additional support for USB peripherals that involve some sort of ethernet-style network. This can be a traditional ethernet, or it can involve some other networking technology that uses ethernet frames as a unit of transfer. It provides functions to transfer ethernet frames over the USB bus, handles certain control messages from the host, and optionally it can provide a network device driver for use by the eCos TCP/IP stack. The package comes with an example host-side device driver.

The USB-ethernet package is not tied to any specific hardware. It requires the presence of USB hardware and a suitable device driver, but not all USB peripherals involve ethernet communications. Hence the configuration system cannot load the package automatically for specific targets, in the way that a USB device driver or an ethernet driver can be loaded automatically. Instead, the package has to be added explicitly. When using the command line tools this will involve an operation like the following:

$ ecosconfig add usbs_eth

Typically, this will automatically cause the USB device driver to become active. Loading the USB-ethernet package automatically provides functionality for initialization, data transfer, and the handling of control messages and state changes. If the current configuration includes the eCos TCP/IP stack then the network device driver support will be enabled as well by default, allowing the stack to exchange ethernet frames over the USB bus.

There is a USB standard for a class of communication devices including ethernet. The package does not implement this standard, due to limitations in the hardware for which the package was first developed. Instead, the package uses its own protocol between USB host device driver and the peripheral.

Usage Scenarios

The USB-ethernet package can be used several different scenarios. In a simple scenario, the peripheral serves only to connect the USB host to a suitable network:

After initialization, and once the USB connection between host and peripheral has been established, higher-level code needs to detect packets that are intended for the host, and to forward these. This can be achieved by the low-level usbs_eth_start_tx function. Similarly, higher-level code needs to detect packets coming from the host, using usbs_eth_start_rx, and to forward these using the real network. As far as the host is concerned it is connected directly to the network. In this scenario there is no confusion about addresses: there is a single MAC address for the host/peripheral combination, corresponding to the connection to the real network, and it is this address which should be supplied during initialization.
In a more complicated scenario, there is a TCP/IP stack running inside the peripheral.

This involves the USB-ethernet package providing a service both to the host and to the eCos TCP/IP stack. It achieves the latter by acting as an eCos network device. Typically, the TCP/IP stack will be configured to act as a network bridge. The USB peripheral needs to examine the packets arriving over the real network. Some of these packets will be intended for the host, while others will be intended for the peripheral itself. To distinguish between these two scenarios, two distinct MAC addresses are needed: one for the host, and one for the peripheral. Similarly, packets sent by the host may have to be forwarded via the real network, or they may be intended for the TCP/IP stack inside the peripheral. Packets generated inside the peripheral’s TCP/IP stack may need to be sent via the real network or over the USB bus. The network bridge software will have to take care of all these possibilities. Unusually for a network bridge, one of the network segments being bridged will only ever have one machine attached.

There are other possible usage scenarios. For example, the peripheral might not be attached to a real network at all. Instead it could be the USB host that acts as a network bridge, allowing a TCP/IP stack inside the peripheral to communicate with the outside world. The various details will depend on the exact type of peripheral being developed.
Initializing the USB-ethernet Package

Name

usbs_eth_init — Initializing the USB-ethernet Package

Synopsis

```c
#include <cyg/io/usb/usbs_eth.h>
void usbs_eth_init(usbs_eth* usbeth, usbs_control_endpoint* ep0, usbs_rx_endpoint* ep1, usbs_tx_endpoint* ep2, unsigned char* mac_address);
```

Description

The USB-ethernet package is not tied to any specific hardware. It requires certain functionality: there must be USB-slave hardware supported by a device driver; there must also be two endpoints for bulk transfers between host and peripheral, one for each direction; there must also be a control endpoint, although of course that is implicit with any USB hardware.

However, USB-slave hardware may well provide more endpoints than the minimum required for ethernet support. Some of those endpoints might be used by other packages, while other endpoints might be used directly by the application, or might not be needed for the peripheral being built. There is also the possibility of a USB peripheral that supports multiple configurations, with the ethernet support active in only some of those configurations. The USB-ethernet package has no knowledge about any of this, so it relies on higher-level code to tell it which endpoints should be used and other information. This is the purpose of the `usbs_eth_init` function.

The first argument identifies the specific `usbs_eth` data structure that is affected. It is expected that the vast majority of affected applications will only provide a single USB-ethernet device to a single host, and the package automatically provides a suitable data structure `usbs_eth0` to support this. If multiple `usbs_eth` structures are needed for some reason then these need to be instantiated by other code, and each one needs to be initialised by a call to `usbs_eth_init()`.

The next three arguments identify the endpoints that should be used for USB communications: a control endpoint, a receive endpoint for ethernet packets coming from the host to the peripheral, and a transmit endpoint for ethernet packets going in the other direction. Obviously all three endpoints should be provided by the same USB hardware. The USB-ethernet package assumes that it has sole access to the receive and transmit endpoints, subject to the use of `usbs_eth_disable` and `usbs_eth_enable` control functions. The package also assumes that no other code is interested in USB state changes or class control messages: it installs handlers `usbs_eth_state_change_handler` and `usbs_eth_class_control_handler` in the control endpoint. If any other code does need to handle USB state changes or class control messages then replacement handlers should be installed after the call to `usbs_eth_init`, and those replacements should invoke the USB-ethernet ones when appropriate.

The final argument to `usbs_eth_init` specifies the MAC address (or Ethernet Station Address) that should be provided to the host-side device driver. Since the USB-ethernet package does not interact directly with a real ethernet device it cannot obtain the MAC address from any hardware. Instead, it must be supplied by higher-level code. The details depend on the scenario in which the USB-ethernet package is being used.
Initializing the USB-ethernet Package

The call to `usbs_eth_init` should normally happen after the enumeration data has been provided but before the underlying USB device driver has been started. If the USB device were to be started first then a connection between host and peripheral could be established immediately, and the host-side device driver would attempt to contact the USB-ethernet package for information such as the MAC address.

```c
int main(int argc, char** argv)
{
    unsigned char host_MAC[6] = { 0x40, 0x5d, 0x90, 0xa9, 0xbc, 0x02 };  
    usbs_sa11x0_ep0.enumeration_data  = &usb_enum_data;  
    ...  
    usbs_eth_init(&usbs_eth0, &usbs_sa11x0_ep0, &usbs_sa11x0_ep1, &usbs_sa11x0_ep2, host_MAC);  
    ...  
    usbs_start(&usbs_sa11x0_ep0);  
    ...  
}
```
USB-ethernet Data Transfers

Name

USB-ethernet Data Transfers — Exchanging ethernet packets with the USB host

Synopsis

```c
#include <cyg/io/usb/usbs_eth.h>

void usbs_eth_start_rx(usbs_eth* usbseth, unsigned char* buffer, void (*)(usbs_eth*,
    void*, int) complete_fn, void* complete_data);

void usbs_eth_start_tx(usbs_eth* usbseth, unsigned char* buffer, void (*)(usbs_eth*,
    void*, int) complete_fn, void* complete_data);
```

Description

The USB-ethernet package provides two main modes of operation. In the first mode it provides a network device driver for use by a TCP/IP stack running inside the USB peripheral. All incoming ethernet packets should be passed up the TCP/IP stack, and only the stack will generate outgoing packets. Apart from initialization and possibly certain control operations, higher-level code will not interact with the USB-ethernet package directly.

In the second mode there is no TCP/IP stack running inside the USB peripheral. For example, a simple USB-ethernet converter has an ethernet chip and a USB port: ethernet packets received by the ethernet chip need to be forwarded to the USB host, and ethernet packets sent by the USB host need to be sent out of the ethernet chip. `usbs_eth_start_rx` and `usbs_eth_start_tx` allow for this lower-level access to the USB-ethernet package.

The two modes of operation are mutually exclusive. If the network device driver mode is enabled then application code should communicate at the TCP/IP level, and not by using the lower-level functions. Instead, it is the network device driver that will make use of these functions, and it assumes that it has exclusive access. The package does not perform any locking.

The transmit and receive functions work in much the same way. The first argument identifies the usbs_eth structure that should be used. For the majority of applications this will be `usbs_eth0`. The second argument specifies the location of the ethernet packet; outgoing for `usbs_eth_start_tx` and incoming for `usbs_eth_start_rx`. This buffer should correspond to the protocol:

1. Outgoing packets can consist of up to 1516 bytes, consisting of a two-byte header specific to USB-ethernet followed by a standard ethernet frame (a header with 6-byte destination address, 6-byte source address and a further two bytes, followed by a payload of up to 1500 bytes). The two-byte USB-ethernet header consists simply of the size of the ethernet frame, i.e. the size of the rest of the packet not including the USB-ethernet header, with the least significant byte first.

2. For incoming packets the supplied buffer should usually be at least 1516 bytes. There may be special circumstances in which a smaller buffer might be safe; for example, if the host-side device driver is modified to support only smaller packets. Once the packet has been received the buffer will contain a two-byte header specific to USB-ethernet, followed by a normal ethernet frame. The header gives the size of the ethernet frame, excluding the header, with the least significant byte first.
Both `usbs_eth_start_tx` and `usbs_eth_start_rx` are asynchronous: the transfer is started and, some time later, a completion function will be invoked. The third and fourth arguments to both `usbs_eth_start_tx` and `usbs_eth_start_rx` supply the completion function and an argument to that function respectively. The completion function will be invoked with three arguments: a pointer to the usbs_eth data structure, usually `usbs_eth0`; the supplied completion data; and a return code field. A negative value indicates that an error occurred, for example `-EPIPE` if the connection between USB host and peripheral has been broken, or `-EAGAIN` if an endpoint has been halted. A positive value indicates the total size of the transfer, which should correspond to the size in the USB-ethernet header plus an additional two bytes for the header itself.

If the data transfer is successful then the completion function will typically be invoked in DSR context rather than in thread context, although this depends on the implementation of the underlying USB device driver. Therefore the completion function is restricted in what it can do; in particular, it must not make any calls that will or may block such as locking a mutex or allocating memory. The kernel documentation should be consulted for more details of DSR’s and interrupt handling generally. Note that if the transfer finishes quickly then the completion function may be invoked before `usbs_eth_start_rx` or `usbs_eth_start_tx` returns. This is especially likely to happen if the current thread is descheduled after starting the data transfer but before returning from these functions.

For transmit operations, it is possible for `usbs_eth_start_tx` to invoke the completion function immediately. If there is no current connection between host and target then the transmit will fail immediately with `-EPIPE`. In addition the USB-ethernet package will check the destination MAC address and make sure that the ethernet frame really is intended for the host: either it must be for the address specified in the initialization call `usbs_eth_init`, or it must be a broadcast packet, or the host must have enabled promiscuous mode.
USB-ethernet State Handling

Name

USB-ethernet State Handling — Maintaining the USB-ethernet connection with the host

Synopsis

```c
#include <cyg/io/usb/usbs_eth.h>

usbs_control_return usbs_eth_class_control_handler(usbs_control_endpoint* ep0, void* callback_data);
void usbs_eth_state_change_handler(usbs_control_endpoint* ep0, void* callback_data,
                                   usbs_state_change change, int old_state);
void usbs_eth_disable(usbs_eth* usbseth);
void usbs_eth_enable(usbs_eth* usbseth);
```

Description

When the USB-ethernet package is initialized by a call to `usbs_eth_init` it installs `usbs_eth_state_change_handler` to handle USB state changes. This allows the package to detect when the connection between the host and the peripheral is established or broken, resulting in internal calls to `usbs_eth_enable` and `usbs_eth_disable` respectively. This is appropriate if no other code needs to access the USB device. However, if there is other code, either other USB-related packages or the application itself, that needs to perform I/O over the USB bus, then typically the USB-ethernet package should not have exclusive access to state change events. Instead, the assumption is that higher-level code, typically provided by the application, will install an alternative state change handler in the control endpoint data structure after the call to `usbs_eth_init`. This alternative handler will either chain into `usbs_eth_state_change_handler` when appropriate, or else it will invoke `usbs_eth_enable` and `usbs_eth_disable` directly. For further details of state change handlers and control endpoints generally, see the documentation for the common USB-slave package.

Similarly, `usbs_eth_init` will install `usbs_eth_class_control_handler` in the control endpoint data structure as the appropriate handler for class-specific USB control messages. This code will handle the ethernet-specific control messages, for example requests by the host to enable or disable promiscuous mode or to obtain the MAC address. If the USB device is not shared with any other code then this is both necessary and sufficient. However, if other code is involved and if that code also needs to process certain control messages, higher-level code should install its own handler and chain to the USB-ethernet one when appropriate. It should be noted that the request code is encoded in just a single byte, so there is a real possibility that exactly the same number will be used by different protocols for different requests. Any such problems will have to be identified and resolved by application developers, and may involve modifying the source code for the USB-ethernet package.

As an alternative to chaining the state change handler, higher-level code can instead call `usbs_eth_disable` and `usbs_eth_enable` directly. These functions may also be called if the USB-ethernet package should become inactive for reasons not related directly to events on the USB bus. The main effect of `usbs_eth_enable` is to restart receive operations and to allow transmits. The main effect of `usbs_eth_disable` is to block further transmits: any current receive operations need to be aborted at the USB level, for example by halting the appropriate endpoint.
Network Device for the eCos TCP/IP Stack

Name

Network Device — USB-ethernet support for the eCos TCP/IP Stack

Description

If the USB peripheral involves running the eCos TCP/IP stack and that stack needs to use USB-ethernet as a transport layer (or as one of the transports), then the USB-ethernet package can provide a suitable network device driver. It is still necessary for higher-level code to perform appropriate initialization by calling `usbs_eth_init`, but after that it will be the TCP/IP stack rather than application code that transmits or receives ethernet frames.

Not all peripherals involving the USB-ethernet package will require a TCP/IP stack. Hence the provision of the network device is controlled by a configuration option `CYGPKG_USBS_ETHDRV`. By default this will be enabled if the TCP/IP package `CYGPKG_NET` is loaded, and disabled otherwise.

There are a number of other configuration options related to the network device. `CYGFUN_USBS_ETHDRV_STATISTICS` determines whether or not the package will maintain statistics, mainly intended for SNMP: by default this will be enabled if the SNMP support package `CYGPKG_SNMPAGENT` is loaded, and disabled otherwise. The name of the ethernet device is controlled by `CYGDATA_USBS_ETHDRV_NAME`, and has a default value of either `eth0` or `eth1` depending on whether or not there is another network device driver present in the configuration.

Usually eCos network device drivers default to using DHCP for obtaining necessary information such as IP addresses. This is not appropriate for USB-ethernet devices. On the host-side the USB-ethernet network device will not exist until the USB peripheral has been plugged in and communication has been established. Therefore any DHCP daemon on the host would not be listening on that network device at the point that eCos requests its IP and other information. A related issue is that the use of DHCP would imply the presence of a DHCP daemon on every affected host machine, as opposed to a single daemon (plus backups) for the network as a whole. For these reasons the USB-ethernet package precludes the use of DHCP as a way of setting the IP address, instead requiring alternatives such as manual configuration.
Network Device for the eCos TCP/IP Stack
Example Host-side Device Driver

Name
Example Host-side Device Driver — Provide host-side support for the eCos USB-ethernet package

Description
The USB-ethernet package is supplied with a single host-side device driver. This driver has been developed against the Linux kernel 2.2.16-22, as shipped with Red Hat 7. The driver is provided as is and should not be considered production quality: for example it only checks for a bogus vendor id 0x4242 rather than an official vendor id supplied by the USB Implementers Forum (http://www.usb.org/). Also, if the peripheral involves multiple configurations or multiple interfaces, it will fail to detect this. However, the driver can be used for simple testing and as the basis of a full device driver. Details of the protocol used between host and peripheral can be found in the Communication Protocol section.

The host-side device driver can be found in the host subdirectory of the USB-ethernet package, specifically the file ecos_usbeth.c, and comes with a Makefile. Both files may need to be modified for specific applications. For example, the vendor id table ecos_usbeth_implementations may need to be updated for the specific USB peripheral being built. The Makefile assumes that the Linux kernel sources reside in /usr/src/linux, and that the kernel has already been configured and built. Assuming this is the case, the device driver can be built simply by invoking make with no additional arguments. This will result in a dynamically loadable kernel module, ecos_usbeth.o, in the current directory.

Note: As normal for Linux kernel builds, the generated files such as ecos_usbeth.o live in the same directory as the source tree. This is very different from eCos where the source tree (or component repository) is kept separate from any builds. There may be problems if the component repository is kept read-only or if it is put under source code control. Any such problems can be avoided by making a copy of the host subdirectory and building that copy.

Loading the kernel module into the current system requires root privileges. If the generic USB support is also a loadable module and has not been loaded already, this must happen first:

```
# insmod usb-uhci
Using /lib/modules/2.2.16-22/usb/usb-uhci.o
```

Depending on the host hardware, the uhci or usb-ohci modules may be more appropriate. Loading the generic USB module will typically result in a number of messages to the logfile /var/log/messages, giving details of the specific host-side hardware that has been detected plus any hubs. The next step is to load the USB-ethernet module:

```
# insmod ecos_usbeth.o
```

This should result in a number of additional diagnostics in the logfile:

```
Apr 1 18:01:08 grumpy kernel: eCos USB-ethernet device driver
Apr 1 18:01:08 grumpy kernel: usb.c: registered new driver ecos_usbeth
```

If a suitable USB peripheral is now connected the host will detect this, assign an address in the local USB network, obtain enumeration data, and find a suitable device driver. Assuming the peripheral and device driver
Example Host-side Device Driver

agree on the supported vendor ids, the `eco_usbeth.o` module will be selected and this will be reported in the system log:

```
Apr 1 18:04:12 grumpy kernel: usb.c: USB new device connect, assigned device number 3
Apr 1 18:04:12 grumpy kernel: eCos-based USB ethernet peripheral active at eth1
```

What can happen next depends very much on the software that is running on top of the USB-ethernet package inside the peripheral. For example, if there is a TCP/IP stack then it should be possible to bring up a network connection between host and peripheral using `ifconfig`. 
Communication Protocol

Name
Communication Protocol — Protocol used between the host-side device driver and the eCos USB-ethernet package

Description
There is a USB standard for the protocol to be used between the host and a class of communication devices, including ethernet. However, the eCos USB-ethernet package does not implement this protocol: the target hardware for which the package was first developed had certain limitations, and could not implement the standard. Instead, the package implements a simple new protocol.

A USB-ethernet peripheral involves bulk transfers on two endpoints: one endpoint will be used for packets from host to peripheral and the other will be used for the opposite direction. Transfers in both directions are variable length, with a lower limit of 16 bytes and an upper limit of 1516 bytes. The first two bytes of each transfer constitute a header specific to USB-ethernet. The next 14 bytes form the normal header for an ethernet frame: destination MAC address, source MAC address, and a protocol field. The remaining data, up to 1500 bytes, are the payload. The first two bytes give the size of the ethernet frame, least significant byte first, with a value between 14 and 1514.

For example an ARP request from host to peripheral involves an ethernet frame of 42 bytes (0x002A), with the usual 14-byte header and a 28-byte payload. The destination is the broadcast address 0xFFFFFFFFFFFF. The source depends on the MAC address specified for the host in the call to usbs_eth_init, e.g. 0x405D90A9BC02. The remaining data is as specified by the appropriate IETF RFC’s (http://www.ietf.org). The actual bulk USB transfer involves the following sequence of 44 bytes:

```
2a 00 ff ff ff ff ff ff 40 5d 90 a9 bc 02 08 06
00 01 08 00 06 04 00 01 40 5d 90 a9 bc 02 0a 00
00 01 00 00 00 00 00 00 0a 00 00 02
```

In addition there are two control messages. These will be sent by the host to endpoint 0, the control endpoint, and by default they will be handled by usbs_eth_class_control_handler. If class-specific control messages are intercepted by other code then it is the responsibility of that code to invoke the USB-ethernet handler when appropriate.

The first control message can be used by the host to obtain a MAC address:

```
#define ECOS_USBETH_CONTROL_GET_MAC_ADDRESS 0x01
```

The control message’s type field should specify IN as the direction. The request field should be 0x01. The length fields should specify a size of 6 bytes. The remaining fields of the control message will be ignored by the USB-ethernet package. The response consists of the 6-byte MAC address supplied by the initialization call usbs_eth_init.

The second control message can be used by the host to enable or disable promiscuous mode:

```
#define ECOS_USBETH_CONTROL_SET_PROMISCUOUS_MODE 0x02
```

This control message involves no further data so the length field should be set to 0. The value field should be non-zero to enable promiscuous mode, zero to disable it. The request field should be 0x02. The remaining fields in the control message will be ignored. It is the responsibility of the host-side device driver to keep track of whether or not promiscuous mode is currently enabled. It will be disabled when the peripheral changes to Configured state, typically at the point where the host-side device driver has been activated.
XXXV. eCos Support for USB Serial like Peripherals
Introduction

Name

Introduction — eCos support for USB Serial like Peripherals

Introduction

The eCos USB-Serial package provides additional support for USB peripherals that look like a serial port to the host. These can follow the ACM communication device specification or simpler devices which just have two bulk endpoints. Microsoft Windows requires ACM mode. Linux should operate with both modes, however ACM may cause problems since the eCos driver does not implement all the class descriptors, so generic mode is recommended.

The USB-Serial package is not tied to any specific hardware. It requires the presence of USB hardware on the target and a suitable device driver to make endpoints available for this code to use. The configuration system cannot load the eCos package automatically for specific targets, in the way that a USB device driver or an ethernet driver can be loaded automatically. Instead, the package has to be added explicitly. When using the command line tools this will involve an operation like the following:

$ ecosconfig add usbs_serial

Typically, this will automatically cause the USB device driver to become active.
Introduction
Configuration

Name
Configuration — Configuration USB Serial like Peripherals

Configuration
The package requires a few basic configurations plus optionally some additional configuration options.
The driver needs two or three endpoints, depending if ACM communications or a more generic model is used.
This is configured with CYGDAT_IO_USB_SLAVE_CLASS_TYPE which can take the value ACM or generic.
The CYGDAT_IO_USB_SLAVE_SERIAL_EP0 must be configured with the control end point of the USB device.
CYGDAT_IO_USB_SLAVE_SERIAL_TX_EP must be configured with the endpoint to be used for transmission and
CYGDAT_IO_USB_SLAVE_SERIAL_RX_EP must be configured with the end point used for reception. Associated
with these are CYGNUM_IO_USB_SLAVE_SERIAL_RX_EP_NUM and CYGNUM_IO_USB_SLAVE_SERIAL_TX_EP_NUM
which are the endpoint numbers and are used during enumeration of the device. The TX and RX endpoints
must operate in BULK mode.
If operation mode ACM is selected a third endpoint is needed. This must operate in interrupt mode and should be configured in CYGNUM_IO_USB_SLAVE_SERIAL_INTR_EP and
CYGNUM_IO_USB_SLAVE_SERIAL_INTR_EP_NUM.
The USB serial device will make its vendor:product ID known to the host. This should be configured with
CYGNUM_IO_USB_SLAVE_SERIAL_VENDOR_ID and CYGNUM_IO_USB_SLAVE_SERIAL_PRODUCT_ID. NOTE: The
default configurations are not valid for products, but should work for testing.
The USB enumeration also contains text strings to describe the device. This text string can be set with
CYGDAT_IO_USB_SLAVE_SERIAL_PRODUCT_STR.
The last configuration option of interest is CYGPKG_IO_USB_SLAVE_SERIAL_EXAMPLES. When true example
programs will be built when the eCos tests are built. These are not pass/fail test like other eCos tests, but
examples of how the eCos USB serial class can be used.
Configuration
Host Configuration

Name

Host Configuration — Host Configuration for USB Serial like Peripherals

Host Configuration

Configuration for two hosts are listed here, Microsoft Windows and Linux. It should also be possible to use the eCos USB serial like peripheral driver with other hosts.

Linux

The eCos USB serial like peripheral driver can be used in Linux in one of two ways.

- Using the generic usbserial kernel module passing the vendor and product ID as module parameters. e.g.
  
  modprobe usbserial vendor=0xabcd product=0x1234

  would load the kernel module so that it would use a USB device abcd:1234 as a serial device.

- Using the mini driver provided with eCos in the host/linux directory. This driver must be edited and the correct vendor and product ID set to match the vendor and product ID used by the device. Once compiled this driver can be loaded with:
  
  modprobe usbserial
  modprobe ecos_usbserial

  This driver is known to compile with kernel versions 2.6.18 and probably works fine with other kernels. However it fails to compile with kernels after 2.6.25.

Both of these methods will result in the Linux Kernel making a new serial device available. This is typically named /dev/ttyUSB0.

Microsoft Windows

To install the device in a Microsoft Windows system make use of the INF file in host/windows/eCosUsbSerial.inf. Copy this INF file and usbser.sys from your version of Windows into an empty directory. Then plug in the USB device. When prompted to load a driver navigate to the INF file and select it.
Host Configuration
API Function

**Name**

usbs_serial_start, usbs_serial_init, usbs_serial_start,
usbs_serial_wait_until_configured, usbs_serial_is_configured,
usbs_serial_start_tx, usbs_serial_wait_for_tx, usbs_serial_tx,
usbs_serial_start_rx, usbs_serial_wait_for_rx, usbs_serial_rx,
usbs_serial_state_change_handler — eCos USB Serial like Peripherals API

**Synopsis**

```c
#include <cyg/io/usb/usbs_serial.h>

void usbs_serial_start (void);
void usbs_serial_init (usbs_serial * ser, usbs_tx_endpoint * tx_ep, usbs_rx_endpoint * rx_ep);
void usbs_serial_wait_until_configured (void);
cyg_bool usbs_serial_is_configured (void);
void usbs_serial_start_tx (usbs_serial * ser, const void * buf, int * n);
int usbs_serial_wait_for_tx (usbs_serial * ser);
void usbs_serial_start_rx (usbs_serial * ser, const void * buf, int * n);
int usbs_serial_wait_for_rx (usbs_serial * ser);
int usbs_serial_rx (usbs_serial * ser, const void * buf, int * n);
void usbs_serial_state_change_handler (usbs_control_endpoint * ep, void * data,
usbs_state_change change, int prev_state);
```

**Description**

For examples of how to use this API see the files .../tests/usbserial_echo.c and ...
.../tests/usb2serial.c

The first function that needs calling is usbs_serial_start(). This will initialise the eCos USB slave layer,
creating all the enumeration data and then let the host know that the device exists.

Once the USB subsystem has been started it is necessary to wait for the host to configure the device using the
function usbs_serial_wait_until_configured(). The host will assign the device an ID and then load the
appropriate device driver in the host in order to make use the device.

Once the device is configured it is then possible to make use of it, i.e. send and receive data. This transfer of
data can be accomplished either asynchronously or synchronously. It is also possible to mix asynchronously
and synchronously between receiving and sending data.

To perform asynchronous operations the functions usbs_serial_start_rx() and
usbs_serial_start_tx() is used to start the operation. These functions start the necessary actions
and then return immediately. At a later time the functions usbs_serial_wait_for_tx() or
usbs_serial_wait_for_rx() should be called. These will, if necessary, block and then return the status and
any data for the previously started asynchronous call.

To perform synchronous operations the functions usbs_serial_rx() and usbs_serial_tx() are used. These
functions will block until the requested action is complete.
API Function
XXXVI. eCos Synthetic Target
Overview

Name
The eCos synthetic target — Overview

Description
Usually eCos runs on either a custom piece of hardware, specially designed to meet the needs of a specific application, or on a development board of some sort that is available before the final hardware. Such boards have a number of things in common:

1. Obviously there has to be at least one processor to do the work. Often this will be a 32-bit processor, but it can be smaller or larger. Processor speed will vary widely, depending on the expected needs of the application. However the exact processor being used tends not to matter very much for most of the development process: the use of languages such as C or C++ means that the compiler will handle those details.

2. There needs to be memory for code and for data. A typical system will have two different types of memory. There will be some non-volatile memory such as flash, EPROM or masked ROM. There will also be some volatile memory such as DRAM or SRAM. Often the code for the final application will reside in the non-volatile memory and all of the RAM will be available for data. However updating non-volatile memory requires a non-trivial amount of effort, so for much of the development process it is more convenient to burn suitable firmware, for example RedBoot, into the non-volatile memory and then use that to load the application being debugged into RAM, alongside the application data and a small area reserved for use by the firmware.

3. The platform must provide certain minimal I/O facilities. Most eCos configurations require a clock signal of some sort. There must also be some way of outputting diagnostics to the user, often but not always via a serial port. Unless special debug hardware is being used, source level debugging will require bidirectional communication between a host machine and the target hardware, usually via a serial port or an ethernet device.

4. All the above is not actually very useful yet because there is no way for the embedded device to interact with the rest of the world, except by generating diagnostics. Therefore an embedded device will have additional I/O hardware. This may be fairly standard hardware such as an ethernet or USB interface, or special hardware designed specifically for the intended application, or quite often some combination. Standard hardware such as ethernet or USB may be supported by eCos device drivers and protocol stacks, whereas the special hardware will be driven directly by application code.

Much of the above can be emulated on a typical PC running Linux. Instead of running the embedded application being developed on a target board of some sort, it can be run as a Linux process. The processor will be the PC’s own processor, for example an x86, and the memory will be the process’ address space. Some I/O facilities can be emulated directly through system calls. For example clock hardware can be emulated by setting up a SIGALRM signal, which will cause the process to be interrupted at regular intervals. This emulation of real hardware will not be particularly accurate, the number of cpu cycles available to the eCos application between clock ticks will vary widely depending on what else is running on the PC, but for much development work it will be good enough.

Other I/O facilities are provided through an I/O auxiliary process, ecosynth, that gets spawned by the eCos application during startup. When an eCos device driver wants to perform some I/O operation, for example send out an ethernet packet, it sends a request to the I/O auxiliary. That is an ordinary Linux application so it has ready access to all normal Linux I/O facilities. To emulate a device interrupt the I/O auxiliary can
Overview

raise a SIGIO signal within the eCos application. The HAL’s interrupt subsystem installs a signal handler for this, which will then invoke the standard eCos ISR/DSR mechanisms. The I/O auxiliary is based around Tcl scripting, making it easy to extend and customize. It should be possible to configure the synthetic target so that its I/O functionality is similar to what will be available on the final target hardware for the application being developed.

A key requirement for synthetic target code is that the embedded application must not be linked with any of the standard Linux libraries such as the GNU C library: that would lead to a confusing situation where both eCos and the Linux libraries attempted to provide functions such as printf. Instead the synthetic target support must be implemented directly on top of the Linux kernels’ system call interface. For example, the kernel provides a system call for write operations. The actual function write is implemented in the system’s C library, but all it does is move its arguments on to the stack or into certain registers and then execute a special trap instruction such as int 0x80. When this instruction is executed control transfers into the kernel, which will validate the arguments and perform the appropriate operation. Now, a synthetic target application cannot be linked with the system’s C library. Instead it contains a function cyg_hal_sys_write which, like the C library’s write function, pushes its arguments on to the stack and executes the trap instruction. The Linux kernel cannot tell the difference, so it will perform the I/O operation requested by the synthetic target. With appropriate knowledge of what system calls are available, this makes it possible to emulate the required I/O facilities. For example, spawning the ecosynth I/O auxiliary involves system calls cyg_hal_sys_fork and cyg_hal_sys_execve, and sending a request to the auxiliary uses cyg_hal_sys_write.

In many ways developing for the synthetic target is no different from developing for real embedded targets. eCos must be configured appropriately: selecting a suitable target such as i386linux will cause the configuration system to load the appropriate packages for this hardware; this includes an architectural HAL package and a platform-specific package; the architectural package contains generic code applicable to all Linux platforms, whereas the platform package is for specific Linux implementations such as the x86 version and contains any processor-specific code. Selecting this target will also bring in some device driver packages. Other aspects of the configuration such as which API’s are supported are determined by the template, by adding and removing packages, and by fine-grained configuration.

In other ways developing for the synthetic target can be much easier than developing for a real embedded target. For example there is no need to worry about building and installing suitable firmware on the target hardware, and then downloading and debugging the actual application over a serial line or a similar connection. Instead an eCos application built for the synthetic target is mostly indistinguishable from an ordinary Linux program. It can be run simply by typing the name of the executable file at a shell prompt. Alternatively you can debug the application using whichever version of gdb is provided by your Linux distribution. There is no need to build or install special toolchains. Essentially using the synthetic target means that the various problems associated with real embedded hardware can be bypassed for much of the development process.

The eCos synthetic target provides emulation, not simulation. It is possible to run eCos in suitable architectural simulators but that involves a rather different approach to software development. For example, when running eCos on the psim PowerPC simulator you need appropriate cross-compilation tools that allow you to build PowerPC executables. These are then loaded into the simulator which interprets every instruction and attempts to simulate what would happen if the application were running on real hardware. This involves a lot of processing overhead, but depending on the functionality provided by the simulator it can give very accurate results. When developing for the synthetic target the executable is compiled for the PC’s own processor and will be executed at full speed, with no need for a simulator or special tools. This will be much faster and somewhat simpler than using an architectural simulator, but no attempt is made to accurately match the behaviour of a
real embedded target.
Installation

Name

Installation — Preparing to use the synthetic target

Host-side Software

To get the full functionality of the synthetic target, users must build and install the I/O auxiliary ecosynth and various support files. It is possible to develop applications for the synthetic target without the auxiliary, but only limited I/O facilities will be available. The relevant code resides in the host subdirectory of the synthetic target architectural HAL package, and building it involves the standard configure, make, and make install steps.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the I/O auxiliary, in a single build tree. This involves using the configure script at the toplevel of the eCos repository, which will automatically search the packages hierarchy for host-side software. For more information on this, see the README.host file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target architectural HAL package then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This involves creating a suitable build directory and running the configure script. Note that building directly in the source tree is not allowed.

$ cd <somewhere suitable>
$ mkdir synth_build
$ cd synth_build
$ <repo>/packages/hal/synth/arch/<version>/host/configure <options>
$ make
$ make install

The code makes extensive use of Tcl/TK and requires version 8.3 or later. This is checked by the configure script. By default it will use the system’s Tcl installation in /usr. If a different, more recent Tcl installation should be used then its location can be specified using the options --with-tcl=<path>, --with-tcl-header=<path> and --with-tcl-lib=<path>. For more information on these options see the README.host file at the toplevel of the eCos repository.

Some users may also want to specify the install location using a --prefix=<path> option. The default install location is /usr/local. It is essential that the bin subdirectory of the install location is on the user’s search PATH, otherwise the eCos application will be unable to locate and execute the I/O auxiliary ecosynth.

Because ecosynth is run automatically by an eCos application rather than explicitly by the user, it is not installed in the bin subdirectory itself. Instead it is installed below libexec, together with various support files such as images. At configure time it is usually possible to specify an alternative location for libexec using --exec-prefix=<path> or --libexecdir=<path>. These options should not be used for this package because the eCos application is built completely separately and does not know how the host-side was configured.

Toolchain

When developing eCos applications for a normal embedded target it is necessary to use a suitable cross-compiler and related tools such as the linker. Developing for the synthetic target is easier because you can just use the standard GNU tools (gcc, g++, ld, ...) which were provided with your Linux distribution, or which
you used to build your own Linux setup. Any reasonably recent version of the tools, for example gcc 2.96 (Red Hat) as shipped with Red Hat Linux 7, should be sufficient.

There is one important limitation when using these tools: current gdb will not support debugging of eCos threads on the synthetic target. As far as gdb is concerned a synthetic target application is indistinguishable from a normal Linux application, so it assumes that any threads will be created by calls to the Linux `pthread_create` function provided by the C library. Obviously this is not the case since the application is never linked with that library. Therefore gdb never notices the eCos thread mechanisms and assumes the application is single-threaded. Fixing this is possible but would involve non-trivial changes to gdb.

Theoretically it is possible to develop synthetic target applications on, for example, a PC running Windows and then run the resulting executables on another machine that runs Linux. This is rarely useful: if a Linux machine is available then usually that machine will also be used for building ecos and the application. However, if for some reason it is necessary or desirable to build on another machine then this requires a suitable cross-compiler and related tools. If the application will be running on a typical PC with an x86 processor then a suitable configure triplet would be `i686-pc-linux-gnu`. The installation instructions for the various GNU tools should be consulted for further information.

**Hardware Preparation**

Preparing a real embedded target for eCos development can be tricky. Often the first step is to install suitable firmware, usually RedBoot. This means creating and building a special configuration for eCos with the RedBoot template, then somehow updating the target’s flash chips with the resulting RedBoot image. Typically it will also be necessary to get a working serial connection, and possibly set up ethernet as well. Although usually none of the individual steps are particularly complicated, there are plenty of ways in which things can go wrong and it can be hard to figure out what is actually happening. Of course some board manufacturers make life easier for their developers by shipping hardware with RedBoot preinstalled, but even then it is still necessary to set up communication between host and target.

None of this is applicable to the synthetic target. Instead you can just build a normal eCos configuration, link your application with the resulting libraries, and you end up with an executable that you can run directly on your Linux machine or via gdb. A useful side effect of this is that application development can start before any real embedded hardware is actually available.

Typically the memory map for a synthetic target application will be set up such that there is a read-only ROM region containing all the code and constant data, and a read-write RAM region for the data. The default locations and sizes of these regions depend on the specific platform being used for development. Note that the application always executes out of ROM: on a real embedded target much of the development would involve running RedBoot firmware there, with application code and data loaded into RAM; usually this would change for the final system; the firmware would be replaced by the eCos application itself, configured for ROM bootstrap, and it would perform the appropriate hardware initialization. Therefore the synthetic target actually emulates the behaviour of a final system, not of a development environment. In practice this is rarely significant, although having the code in read-only memory can help catch some problems in application code.
Running a Synthetic Target Application

Name

Execution — Arguments and configuration files

Description

The procedure for configuring and building eCos and an application for the synthetic target is the same as for any other eCos target. Once an executable has been built it can be run like any Linux program, for example from a shell prompt,

$ ecos_hello <options>

or using gdb:

$ gdb --nw --quiet --args ecos_hello <options>
(gdb) run
Starting program: ecos_hello <options>

By default use of the I/O auxiliary is disabled. If its I/O facilities are required then the option --io must be used.

Note: In future the default behaviour may change, with the I/O auxiliary being started by default. The option --nio can be used to prevent the auxiliary from being run.

Command-line Arguments

The syntax for running a synthetic target application is:

$ <ecos_app> [options] [-- [app_options]]

Command line options up to the -- are passed on to the I/O auxiliary. Subsequent arguments are not passed on to the auxiliary, and hence can be used by the eCos application itself. The full set of arguments can be accessed through the variables cyg_hal_sys_argc and cyg_hal_sys_argv.

The following options are accepted as standard:

--io

This option causes the eCos application to spawn the I/O auxiliary during HAL initialization. Without this option only limited I/O will be available.

--nio

This option prevents the eCos application from spawning the I/O auxiliary. In the current version of the software this is the default.

--nw, --no-windows

The I/O auxiliary can either provide a graphical user interface, or it can run in a text-only mode. The default is to provide the graphical interface, but this can be disabled with --nw. Emulation of some devices, for example buttons connected to digital inputs, requires the graphical interface.
Running a Synthetic Target Application

-\texttt{w}, --windows

The \texttt{-w} causes the I/O auxiliary to provide a graphical user interface. This is the default.

-\texttt{v}, --version

The \texttt{-v} option can be used to determine the version of the I/O auxiliary being used and where it has been installed. Both the auxiliary and the eCos application will exit immediately.

-\texttt{h}, --help

\texttt{-h} causes the I/O auxiliary to list all accepted command-line arguments. This happens after all devices have been initialized, since the host-side support for some of the devices may extend the list of recognised options. After this both the auxiliary and the eCos application will exit immediately. This option implies \texttt{-nw}.

-\texttt{k}, --keep-going

If an error occurs in the I/O auxiliary while reading in any of the configuration files or initializing devices, by default both the auxiliary and the eCos application will exit. The \texttt{-k} option can be used to make the auxiliary continue in spite of errors, although obviously it may not be fully functional.

-\texttt{nr}, --no-rc

Normally the auxiliary processes two user configuration files during startup: \texttt{initrc.tcl} and \texttt{mainrc.tcl}. This can be suppressed using the \texttt{-nr} option.

-\texttt{x}, --exit

When providing a graphical user interface the I/O auxiliary will normally continue running even after the eCos application has exited. This allows the user to take actions such as saving the current contents of the main text window. If run with \texttt{-x} then the auxiliary will exit as soon the application exits.

-\texttt{nx}, --no-exit

When the graphical user interface is disabled with \texttt{-nw} the I/O auxiliary will normally exit immediately when the eCos application exits. Without the graphical frontend there is usually no way for the user to interact directly with the auxiliary, so there is no point in continuing to run once the eCos application will no longer request any I/O operations. Specifying the \texttt{-nx} option causes the auxiliary to continue running even after the application has exited.

-\texttt{V}, --verbose

This option causes the I/O auxiliary to output some additional information, especially during initialization.

\texttt{-l \langle file \rangle}, --logfile \texttt{\langle file \rangle}

Much of the output of the eCos application and the I/O auxiliary is simple text, for example resulting from eCos \texttt{printf} or \texttt{diag_printf} calls. When running in graphical mode this output goes to a central text window, and can be saved to a file or edited via menus. The \texttt{-l} can be used to automatically generate an additional logfile containing all the text. If graphical mode is disabled then by default all the text just goes to the current standard output. Specifying \texttt{-l} causes most of the text to go into a logfile instead, although some messages such as errors generated by the auxiliary itself will still go to stdout as well.

\texttt{-t \langle file \rangle}, --target \texttt{\langle file \rangle}

During initialization the I/O auxiliary reads in a target definition file. This file holds information such as which Linux devices should be used to emulate the various eCos devices. The \texttt{-t} option can be used to specify which target definition should be used for the current run, defaulting to \texttt{default.tdf}. It is not necessary to include the \texttt{.tdf} suffix, this will be appended automatically if necessary.
Running a Synthetic Target Application

This option can be used to control the size and position of the main window, as per X conventions.

The I/O auxiliary loads support for the various devices dynamically and some devices may accept additional command line arguments. Details of these can be obtained using the -h option or by consulting the device-specific documentation. If an unrecognised command line argument is used then a warning will be issued.

The Target Definition File

The eCos application will want to access devices such as eth0 or /dev/ser0. These need to be mapped on to Linux devices. For example some users may all traffic on the eCos /dev/ser0 serial device to go via the Linux serial device /dev/ttyS1, while ethernet I/O for the eCos eth0 device should be mapped to the Linux ethertap device tap3. Some devices may need additional configuration information, for example to limit the number of packets that should be buffered within the I/O auxiliary. The target definition file provides all this information.

By default the I/O auxiliary will look for a file default.tdf. An alternative target definition can be specified on the command line using -t, for example:

$ bridge_app --io -t twineth

A .tdf suffix will be appended automatically if necessary. If a relative pathname is used then the I/O auxiliary will search for the target definition file in the current directory, then in ~/.ecos/synth/, and finally in its install location.

A typical target definition file might look like this:

synth_device console {
    # appearance -foreground white -background black
    filter trace {^TRACE:.} -foreground HotPink1 -hide 1
}

synth_device ethernet {
    eth0 real eth1
    eth1 ethertap tap4 00:01:02:03:FE:06

    ## Maximum number of packets that should be buffered per interface.
    ## Default 16
    #max_buffer 32

    ## Filters for the various recognised protocols.
    ## By default all filters are visible and use standard colours.
    filter ether -hide 0
    #filter arp -hide 1
    #filter ipv4 -hide 1
    #filter ipv6 -hide 1
}

A target definition file is actually a Tcl script that gets run in the main interpreter of the I/O auxiliary during initialization. This provides a lot of flexibility if necessary. For example the script could open a socket to a resource management server of some sort to determine which hardware facilities are already in use and adapt accordingly. Another possibility is to adapt based on command line arguments. Users who are not familiar with Tcl programming should still be able to edit a simple target definition file without too much difficulty, using a mixture of cut’n’paste, commenting or uncommenting various lines, and making small edits such as changing tap4 to eth2.

Each type of device will have its own entry in the target definition file, taking the form:
Running a Synthetic Target Application

synth_device <device type> {
  <options>
}

The documentation for each synthetic target device should provide details of the options available for that device, and often a suitable fragment that can be pasted into a target definition file and edited. There is no specific set of options that a given device will always provide. However in practice many devices will use common code exported by the main I/O auxiliary, or their implementation will involve some re-use of code for an existing device. Hence certain types of option are common to many devices.

A good example of this is filters, which control the appearance of text output. The above target definition file defines a filter `trace` for output from the eCos application. The regular expression will match output from the infrastructure package’s tracing facilities when `CYGDBG_USE_TRACING` and `CYGDBG_INFRA_DEBUG_TRACE_ASSERT_SIMPLE` are enabled. With the current settings this output will not be visible by default, but can be made visible using the menu item `System Filters`. If made visible the trace output will appear in an unusual colour, so users can easily distinguish the trace output from other text. All filters accept the following options:

- `hide [0|1]`
  This controls whether or not text matching this filter should be invisible by default or not. At run-time the visibility of each filter can be controlled using the `System Filters` menu item.

- `foreground <colour>`
  This specifies the foreground colour for all text matching this filter. The colour can be specified using an RGB value such as `#F08010`, or a symbolic name such as "light steel blue". The X11 utility `showrgb` can be used to find out about the available colours.

- `background <colour>`
  This specifies the background colour for all text matching the filter. As with `foreground` the colour can be specified using a symbolic name or an RGB value.

Some devices may create their own subwindows, for example to monitor ethernet traffic or to provide additional I/O facilities such as emulated LED’s or buttons. Usually the target definition file can be used to control the layout of these windows.

The I/O auxiliary will not normally warn about `synth_device` entries in the target definition file for devices that are not actually needed by the current eCos application. This makes it easier to use a single file for several different applications. However it can lead to confusion if an entry is spelled incorrectly and hence does not actually get used. The `-V` command line option can be used to get warnings about unused device entries in the target definition file.

If the body of a `synth_device` command contains an unrecognised option and the relevant device is in use, the I/O auxiliary will always issue a warning about such options.

User Configuration Files

During initialization the I/O auxiliary will execute two user configuration files, `initrc.tcl` and `mainrc.tcl`. It will look for these files in the directory `~/.ecos/synth/`. If that directory does not yet exist it will be created and populated with initial dummy files.

Both of these configuration files are Tcl scripts and will be run in the main interpreter used by the I/O auxiliary itself. This means that they have full access to the internals of the auxiliary including the various Tk widgets, and they can perform file or socket I/O if desired. The section Writing New Devices - host contains information...
about the facilities available on the host-side for writing new device drivers, and these can also be used in the initialization scripts.

The `initrc.tcl` script is run before the auxiliary has processed any requests from the eCos application, and hence before any devices have been instantiated. At this point the generic command-line arguments has been processed, the target definition file has been read in, and the hooks functionality has been initialized. If running in graphical mode the main window will have been created, but has been withdrawn from the screen to allow new widgets to be added without annoying screen flicker. A typical `initrc.tcl` script could add some menu or toolbar options, or install a hook function that will be run when the eCos application exits.

The `mainrc.tcl` script is run after eCos has performed all its device initialization and after C++ static constructors have run, and just before the call to `cyg_start` which will end up transferring control to the application itself. A typical `mainrc.tcl` script could look at what interrupt vectors have been allocated to which devices and create a little monitor window that shows interrupt activity.

---

**Session Information**

When running in graphical mode, the I/O auxiliary will read in a file `~/.ecos/synth/guisession` containing session information. This file should not normally be edited manually, instead it gets updated automatically when the auxiliary exits. The purpose of this file is to hold configuration options that are manipulated via the graphical interface, for example which browser should be used to display online help.

---

**Warning**

GUI session functionality is not yet available in the current release. When that functionality is fully implemented it is possible that some target definition file options may be removed, to be replaced by graphical editing via a suitable preferences dialog, with the current settings saved in the session file.
The I/O Auxiliary’s User Interface

Name
User Interface — Controlling the I/O Auxiliary

Description
The synthetic target auxiliary is designed to support both extensions and user customization. Support for the desired devices is dynamically loaded, and each device can extend the user interface. For example it is possible for a device to add menu options, place new buttons on the toolbar, create its own sub-window within the overall layout, or even create entire new toplevel windows. These subwindows or toplevels could show graphs of activity such as interrupts or packets being transferred. They could also allow users to interact with the eCos application, for example by showing a number of buttons which will be mapped on to digital inputs in the eCos application. Different applications will have their own I/O requirements, changing the host-side support files that get loaded and that may modify the user interface. The I/O auxiliary also reads in user configuration scripts which can enhance the interface in the same way. Therefore the exact user interface will depend on the user and on the eCos application being run. However the overall layout is likely to remain the same.

The title bar identifies the window as belonging to an eCos synthetic target application and lists both the application name and its process id. The latter is especially useful if the application was started directly from a shell prompt and the user now wants to attach a gdb session. The window has a conventional menu bar with the usual entries, plus a toolbar with buttons for common operations such as cut and paste. Balloon help is supported.

There is a central text window, possibly surrounded by various sub-windows for various devices. For example there could be a row of emulated LED’s above the text window, and monitors of ethernet traffic and interrupt activity on the right. At the bottom of the window is a status line, including a small animation that shows whether or not the eCos application is still running.
The I/O Auxiliary’s User Interface

Menus and the Toolbar

Usually there will be four menus on the menu bar: File, Edit, View and Help.

On the File menu there are three entries related to saving the current contents of the central text window. Save is used to save the currently visible contents of the text window. Any text that is hidden because of filters will not be written to the savefile. If there has been a previous Save or Save As operation then the existing savefile will be re-used, otherwise the user will be asked to select a suitable file. Save As also saves just the currently visible contents but will always prompt the user for a filename. Save All can be used to save the full contents of the text window, including any text that is currently hidden. It will always prompt for a new filename, to avoid confusion with partial savefiles.

Usually the eCos application will be run from inside gdb or from a shell prompt. Killing off the application while it is being debugged in a gdb session is not a good idea, it would be better to use gdb’s own kill command. Alternatively the eCos application itself can use the CYG_TEST_EXIT or cyg_hal_sys_exit functionality. However it is possible to terminate the application from the I/O auxiliary using Kill eCos. A clean shutdown will be attempted, but that can fail if the application is currently halted inside gdb or if it has crashed completely. As a last resort SIGKILL will be used.

When operating in graphical mode the I/O auxiliary will normally continue to run even after the eCos application has exited. This allows the user to examine the last few lines of output, and perhaps perform actions such as saving the output to a file. The Exit menu item can be used to shut down the auxiliary. Note that this behaviour can be changed with command line arguments --exit and --no-exit.

If Exit is used while the eCos application is still running then the I/O auxiliary will first attempt to terminate the application cleanly, and then exit.

The Edit menu contains the usual entries for text manipulation: Cut, Copy, Paste, Clear and Select All. These all operate on the central text window. By default this window cannot be edited so the cut, paste and clear operations are disabled. If the user wants to edit the contents of the text window then the Read Only checkbutton should be toggled.

The Preferences menu item brings up a miscellaneous preferences dialog. One of the preferences relates to online help: the I/O auxiliary does not currently have a built-in html viewer; instead it will execute an external browser of some sort. With the example settings shown, the I/O auxiliary will first attempt to interact with an existing mozilla session. If that fails it will try to run a new mozilla instance, or as a last result use the Gnome help viewer.
The View menu contains the System Filters entry, used to edit the settings for the current filters.

The Help menu can be used to activate online help for eCos generally, for the synthetic target as a whole, and for specific devices supported by the generic target. The Preferences dialog can be used to select the browser that will be used.

Note: At the time of writing there is no well-defined toplevel index file for all eCos documentation. Hence the relevant menu item is disabled. Documentation for the synthetic target and the supported devices is stored as part of the package itself so can usually be found fairly easily. It may be necessary to set the ECOS_REPOSITORY environment variable.

The Main Text Window

The central text window holds the console output from the eCos application: the screen shot above shows DHCP initialization data from the TCP/IP stack, and some output from the main thread at the bottom. Some devices can insert text of their own, for example the ethernet device support can be configured to show details of incoming and outgoing packets. Mixing the output from the eCos application and the various devices can make it easier to understand the order in which events occur.

The appearance of text from different sources can be controlled by means of filters, and it is also possible to hide some of the text. For example, if tracing is enabled in the eCos configuration then the trace output can be given its own colour scheme, making it stand out from the rest of the output. In addition the trace output is generally voluminous so it can be hidden by default, made visible only to find out more about what was happening when a particular problem occurred. Similarly the ethernet device support can output details of the various packets being transferred, and using a different background colour for this output again makes it easier to distinguish from console output.

The default appearance for most filters is controlled via the target definition file. An example entry might be:

```
filter trace {^TRACE:.} -foreground HotPink1 -hide 1
```

The various colours and the hide flag for each filter can be changed at run-time, using the System Filters item on the View menu. This will bring up a dialog like the following:
The I/O Auxiliary’s User Interface

It should be noted that the text window is line-oriented, not character-oriented. If an eCos application sends a partial line of text then that will remain buffered until a newline character is received, rather than being displayed immediately. This avoids confusion when there is concurrent output from several sources.

By default the text window is read-only. This means it will not allow cut, paste and clear operations, and keyboard input will be ignored. The Edit menu has a checkbutton Read Only which can be toggled to allow write operations. For example, a user could type in a reminder of what was happening at this time, or paste in part of a gdb session. Such keyboard input does not get forwarded to the eCos application: if the latter requires keyboard input then that should happen via a separate keyboard device.

Positioning Optional Windows

Some devices may create their own subwindows, for example to monitor ethernet traffic or to provide additional I/O facilities such as emulated LED’s or buttons. Usually the target definition file can be used to control the layout of these windows. This requires an understanding of the overall layout of the display.
Subwindows are generally packed in one of eight frames surrounding the central text window: .main.nw, .main.n, .main.ne, .main.w, .main.e, .main.sw, .main.s, and .main.se. To position a row of LED’s above the text window and towards the left, a target definition file could contain an entry such as:

```tcl
synth_device led {
    pack -in .main.n -side left
    ...
}
```

Similarly, to put a traffic monitor window on the right of the text window would involve something like:

```tcl
...
    monitor_pack -in .main.e -side bottom
    ...
```

Often it will be sufficient to specify a container frame and one of left, right, top or bottom. Full control over the positioning requires an understanding of Tcl/Tk and in particular the packing algorithm, and an appropriate reference work should be consulted.

**Global Settings**

*Note:* This section still to be written - it should document the interaction between X resources and ecosynth, and how users can control settings such as the main foreground and background colours.
The Console Device

Name
The console device — Show output from the eCos application

Description
The eCos application can generate text output in a variety of ways, including calling `printf` or `diag_printf`. When the I/O auxiliary is enabled the eCos startup code will instantiate a console device to process all such output. If operating in text mode the output will simply go to standard output, or to a logfile if the `-l` command line option is specified. If operating in graphical mode the output will go to the central text window, and optionally to a logfile as well. In addition it is possible to control the appearance of the main text via the target definition file, and to install extra filters for certain types of text.

It should be noted that the console device is line-oriented, not character-oriented. This means that outputting partial lines is not supported, and some functions such as `fflush` and `setvbuf` will not operate as expected. This limitation prevents much possible confusion when using filters to control the appearance of the text window, and has some performance benefits - especially when the eCos application generates a great deal of output such as when tracing is enabled. For most applications this is not a problem, but it is something that developers should be aware of.

The console device is output-only, it does not provide any support for keyboard input. If the application requires keyboard input then that should be handled by a separate eCos device package and matching host-side code.

Installation
The eCos side of the console device is implemented by the architectural HAL itself, in the source file `synth_diag.c`, rather than in a separate device package. Similarly the host-side implementation, `console.tcl`, is part of the architectural HAL’s host-side support. It gets installed automatically alongside the I/O auxiliary itself, so no separate installation procedure is required.

Target Definition File
The target definition file can contain a number of entries related to the console device. These are all optional, they only control the appearance of text output. If such control is desired then the relevant options should appear in the body of a `synth_device` entry:

```tcl
synth_device console {
   ...
}
```

The first option is `appearance`, used to control the appearance of any text generated by the eCos application that does not match one of the installed filters. This option takes the same argument as any other filter, for example:

```tcl
synth_device console {
   appearance -foreground white -background black
   ...
}
```

Any number of additional filters can be created with a `filter` option, for example:
The Console Device

```tcl
synth_device console {
  ...  
  filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
  ...  
}
```

The first argument gives the new filter a name which will be used in the filters dialog. Filter names should be unique. The second argument is a Tcl regular expression. The console support will match each line of eCos output against this regular expression, and if a match is found then the filter will be used for this line of text. The above example matches any line of output that begins with TRACE:, which corresponds to the eCos infrastructure’s tracing facilities. The remaining options control the desired appearance for matched text. If some eCos output matches the regular expressions for several different filters then only the first match will be used.

**Target-side Configuration Options**

There are no target-side configuration options related to the console device.

**Command Line Arguments**

The console device does not use any command-line arguments.

**Hooks**

The console device does not provide any hooks.

**Additional Tcl Procedures**

The console device does not provide any additional Tcl procedures that can be used by other scripts.
System Calls

Name
cyg_hal_sys_xyz — Access Linux system facilities

Synopsis

#include <cyg/hal/hal_io.h>

int cyg_hal_sys_xyzzy(...);  

Description

On a real embedded target eCos interacts with the hardware by peeking and poking various registers, manipulating special regions of memory, and so on. The synthetic target does not access hardware directly. Instead I/O and other operations are emulated by making appropriate Linux system calls. The HAL package exports a number of functions which allow other packages, or even application code, to make these same system calls. However this facility must be used with care: any code which calls, for example, cyg_hal_sys_write will only ever run on the synthetic target; that functionality is obviously not provided on any real hardware because there is no underlying Linux kernel to implement it.

The synthetic target only provides a subset of the available system calls, specifically those calls which have proved useful to implement I/O emulation. This subset can be extended fairly easily if necessary. All of the available calls, plus associated data structures and macros, are defined in the header file cyg/hal/hal_io.h. There is a simple convention: given a Linux system call such as open, the synthetic target will prefix cyg_hal_sys and provide a function with that name. The second argument to the open system call is a set of flags such as O_RDONLY, and the header file will define a matching constant CYG_HAL_SYS_O_RDONLY. There are also data structures such as cyg_hal_sys_sigset_t, matching the Linux data structure sigset_t. In most cases the functions provided by the synthetic target behave as per the documentation for the Linux system calls, and section 2 of the Linux man pages can be consulted for more information. There is one important difference: typically the documentation will say that a function returns -1 to indicate an error, with the actual error code held in errno; the actual underlying system call and hence the cyg_hal_sys_xyz provided by eCos instead returns a negative number to indicate an error, with the absolute value of that number corresponding to the error code; usually it is the C library which handles this and manipulates errno, but of course synthetic target applications are not linked with that Linux library.

However, there are some exceptions. The Linux kernel has evolved over the years, and some of the original system call interfaces are no longer appropriate. For example the original select system call has been superseded by _newselect, and that is what the select function in the C library actually uses. The old call is still available to preserve binary compatibility but, like the C library, eCos makes use of the new one because it provides the appropriate functionality. In an attempt to reduce confusion the eCos function is called cyg_hal_sys___newselect, in other words it matches the official system call naming scheme. The authoritative source of information on such matters is the Linux kernel sources themselves, and especially its header files.

eCos packages and applications should never #include Linux header files directly. For example, doing a #include </usr/include/fcntl.h> to access additional macros or structure definitions, or alternatively manipulating the header file search path, will lead to problems because the Linux header files are likely to
duplicate and clash with definitions in the eCos headers. Instead the appropriate functionality should be extracted from the Linux headers and moved into either cyg/hal/hal_io.h or into application code, with suitable renaming to avoid clashes with eCos names. Users should be aware that large-scale copying may involve licensing complications.

Adding more system calls is usually straightforward and involves adding one or more lines to the platform-specific file in the appropriate platform HAL, for example syscall-i386-linux-1.0.S. However it is necessary to do some research first about the exact interface implemented by the system call, because of issues such as old system calls that have been superseded. The required information can usually be found fairly easily by searching through the Linux kernel sources and possibly the GNU C library sources.
Writing New Devices - target

Name
Writing New Devices — extending the synthetic target, target-side

Synopsis

```c
#include <cyg/hal/hal_io.h>

int synth_auxiliary_instantiate(const char* package, const char* version, const char* device, const char* instance, const char* data);
void synth_auxiliary_xchgmsg(int device_id, int request, int arg1, int arg2, const unsigned char* txdata, int txlen, int* reply, unsigned char* rxdata, int* rxlen, int max_rxlen);
```

Description

In some ways writing a device driver for the synthetic target is very similar to writing one for a real target. Obviously it has to provide the standard interface for that class of device, so for example an ethernet device has to provide `can_send`, `send`, `recv` and similar functions. Many devices will involve interrupts, so the driver contains ISR and DSR functions and will call `cyg_drv_interrupt_create`, `cyg_drv_interrupt_acknowledge`, and related functions.

In other ways writing a device driver for the synthetic target is very different. Usually the driver will not have any direct access to the underlying hardware. In fact for some devices the I/O may not involve real hardware, instead everything is emulated by widgets on the graphical display. Therefore the driver cannot just peek and poke device registers, instead it must interact with host-side code by exchanging message. The synthetic target HAL provides a function `synth_auxiliary_xchgmsg` for this purpose.

Initialization of a synthetic target device driver is also very different. On real targets the device hardware already exists when the driver’s initialization routine runs. On the synthetic target it is first necessary to instantiate the device inside the I/O auxiliary, by a call to `synth_auxiliary_instantiate`. That function performs a special message exchange with the I/O auxiliary, causing it to load a Tcl script for the desired type of device and run an instantiation procedure within that script.

Use of the I/O auxiliary is optional: if the user does not specify `--io` on the command line then the auxiliary will not be started and hence most I/O operations will not be possible. Device drivers should allow for this possibility, for example by just discarding any data that gets written. The HAL exports a flag `synth_auxiliary_running` which should be checked.

Instantiating a Device

Device instantiation should happen during the C++ prioritized static constructor phase of system initialization, before control switches to `cyg_user_start` and general application code. This ensures that there is a clearly defined point at which the I/O auxiliary knows that all required devices have been loaded. It can then perform various consistency checks and clean-ups, run the user’s `mainrc.tcl` script, and make the main window visible.
Writing New Devices - target

For standard devices generic eCos I/O code will call the device initialization routines at the right time, iterating through the DEVTAB table in a static constructor. The same holds for network devices and file systems. For more custom devices code like the following can be used:

```cpp
#include <cyg/infra/cyg_type.h>
class mydev_init {
public:
    mydev_init() {
        ...
    }
};
static mydev_init mydev_init_object CYGBLD_ATTRIB_INIT_PRI(CYG_INIT_IO);
```

Some care has to be taken because the object `mydev_init_object` will typically not be referenced by other code, and hence may get eliminated at link-time. If the code is part of an eCos package then problems can be avoided by putting the relevant file in `libextras.a`:

```cpp
cdl_package CYGPKG_DEVS_MINE {
    ...
    compile -library=libextras.a init.cxx
}
```

For devices inside application code the same can be achieved by linking the relevant module as a `.o` file rather than putting it in a `.a` library.

In the device initialization routine the main operation is a call to `synth_auxiliary_instantiate`. This takes five arguments, all of which should be strings:

- **package**
  
  For device drivers which are eCos packages this should be a directory path relative to the eCos repository, for example `devs/eth/synth/ecosynth`. This will allow the I/O auxiliary to find the various host-side support files for this package within the install tree. If the device is application-specific and not part of an eCos package then a NULL pointer can be used, causing the I/O auxiliary to search for the support files in the current directory and then in `~/.ecos/synth` instead.

- **version**
  
  For eCos packages this argument should be the version of the package that is being used, for example `current`. A simple way to get this version is to use the `SYNTH_MAKESTRING` macro on the package name. If the device is application-specific then a NULL pointer should be used.

- **device**
  
  This argument specifies the type of device being instantiated, for example `ethernet`. More specifically the I/O auxiliary will append a `.tcl` suffix, giving the name of a Tcl script that will handle all I/O requests for the device. If the application requires several instances of a type of device then the script will only be loaded once, but the script will contain an instantiation procedure that will be called for each device instance.

- **instance**
  
  If it is possible to have multiple instances of a device then this argument identifies the particular instance, for example `eth0` or `eth1`. Otherwise a NULL pointer can be used.

- **data**
  
  This argument can be used to pass additional initialization data from eCos to the host-side support. This is useful for devices where eCos configury must control certain aspects of the device, rather than host-side...
configury such as the target definition file, because eCos has compile-time dependencies on some or all of the relevant options. An example might be an emulated frame buffer where eCos has been statically configured for a particular screen size, orientation and depth. There is no fixed format for this string, it will be interpreted only by the device-specific host-side Tcl script. However the string length should be limited to a couple of hundred bytes to avoid possible buffer overflow problems.

Typical usage would look like:

```c
if (!synth_auxiliary_running) {
    return;
}

id = synth_auxiliary_instantiate("devs/eth/synth/ecosynth",
    SYNTH_MAKESTRING(CYGPKG_DEVS_ETH_ECOSYNTH),
    "ethernet",
    "eth0",
    (const char*) 0);
```

The return value will be a device identifier which can be used for subsequent calls to `synth_auxiliary_xchgmsg`. If the device could not be instantiated then -1 will be returned. It is the responsibility of the host-side software to issue suitable diagnostics explaining what went wrong, so normally the target-side code should fail silently.

Once the desired device has been instantiated, often it will be necessary to do some additional initialization by a message exchange. For example an ethernet device might need information from the host-side about the MAC address, the interrupt vector, and whether or not multicasting is supported.

## Communicating with a Device

Once a device has been instantiated it is possible to perform I/O by sending messages to the appropriate Tcl script running inside the auxiliary, and optionally getting back replies. I/O operations are always initiated by the eCos target-side, it is not possible for the host-side software to initiate data transfers. However the host-side can raise interrupts, and the interrupt handler inside the target can then exchange one or more messages with the host.

There is a single function to perform I/O operations, `synth_auxiliary_xchgmsg`. This takes the following arguments:

- `device_id`
  - This should be one of the identifiers returned by a previous call to `synth_auxiliary_instantiate`, specifying the particular device which should perform some I/O.

- `request`
  - Request are just signed 32-bit integers that identify the particular I/O operation being requested. There is no fixed set of codes, instead each type of device can define its own.

- `arg1`
- `arg2`
  - For some requests it is convenient to pass one or two additional parameters alongside the request code. For example an ethernet device could define a multicast-all request, with `arg1` controlling whether this mode should be enabled or disabled. Both `arg1` and `arg2` should be signed 32-bit integers, and their values are interpreted only by the device-specific Tcl script.
Writing New Devices - target

txdata
txlen

Some I/O operations may involve sending additional data, for example an ethernet packet. Alternatively a control operation may require many more parameters than can easily be encoded in arg1 and arg2, so those parameters have to be placed in a suitable buffer and extracted at the other end. txdata is an arbitrary buffer of txlen bytes that should be sent to the host-side. There is no specific upper bound on the number of bytes that can be sent, but usually it is a good idea to allocate the transmit buffer statically and keep transfers down to at most several kilobytes.

reply

If the host-side is expected to send a reply message then reply should be a pointer to an integer variable and will be updated with a reply code, a simple 32-bit integer. The synthetic target HAL code assumes that the host-side and target-side agree on the protocol being used: if the host-side will not send a reply to this message then the reply argument should be a NULL pointer; otherwise the host-side must always send a reply code and the reply argument must be valid.

rxdata
rxlen

Some operations may involve additional data coming from the host-side, for example an incoming ethernet packet. rxdata should be a suitably-sized buffer, and rxlen a pointer to an integer variable that will end up containing the number of bytes that were actually received. These arguments will only be used if the host-side is expected to send a reply and hence the reply argument was not NULL.

max_rxlen

If a reply to this message is expected and that reply may involve additional data, max_rxlen limits the size of that reply. In other words, it corresponds to the size of the rxdata buffer.

Most I/O operations involve only some of the arguments. For example transmitting an ethernet packet would use the request, txdata and txlen fields (in addition to device_id which is always required), but would not involve arg1 or arg2 and no reply would be expected. Receiving an ethernet packet would involve request, rxdata, rxlen and max_rxlen; in addition reply is needed to get any reply from the host-side at all, and could be used to indicate whether or not any more packets are buffered up. A control operation such as enabling multicast mode would involve request and arg1, but none of the remaining arguments.

Interrupt Handling

Interrupt handling in the synthetic target is much the same as on a real target. An interrupt object is created using cyg_drv_interrupt_create, attached, and unmasked. The emulated device - in other words the Tcl script running inside the I/O auxiliary - can raise an interrupt. Subject to interrupts being disabled and the appropriate vector being masked, the system will invoke the specified ISR function. The synthetic target HAL implementation does have some limitations: there is no support for nested interrupts, interrupt priorities, or a separate interrupt stack. Supporting those might be appropriate when targetting a simulator that attempts to model real hardware accurately, but not for the simple emulation provided by the synthetic target.

Of course the actual implementation of the ISR and DSR functions will be rather different for a synthetic target device driver. For real hardware the device driver will interact with the device by reading and writing device registers, managing DMA engines, and the like. A synthetic target driver will instead call synth_auxiliary_xchgmsg to perform the I/O operations.

There is one other significant difference between interrupt handling on the synthetic target and on real hardware. Usually the eCos code will know which interrupt vectors are used for which devices. That information is fixed when the target hardware is designed. With the synthetic target interrupt vectors are assigned to devices
on the host side, either via the target definition file or dynamically when the device is instantiated. Therefore the initialization code for a target-side device driver will need to request interrupt vector information from the host-side, via a message exchange. Such interrupt vectors will be in the range 1 to 31 inclusive, with interrupt 0 being reserved for the real-time clock.
Writing New Devices - host

Name
Writing New Devices — extending the synthetic target, host-side

Description
On the host-side adding a new device means writing a Tcl/Tk script that will handle instantiation and subsequent requests from the target-side. These scripts all run in the same full interpreter, extended with various commands provided by the main I/O auxiliary code, and running in an overall GUI framework. Some knowledge of programming with Tcl/Tk is required to implement host-side device support.

Some devices can be implemented entirely using a Tcl/Tk script. For example, if the final system will have some buttons then those can be emulated in the synthetic target using a few Tk widgets. A simple emulation could just have the right number of buttons in a row. A more advanced emulation could organize the buttons with the right layout, perhaps even matching the colour scheme, the shapes, and the relative sizes. With other devices it may be necessary for the Tcl script to interact with an external program, because the required functionality cannot easily be accessed from a Tcl script. For example interacting with a raw ethernet device involves some ioctl calls, which is easier to do in a C program. Therefore the ethernet.tcl script which implements the host-side ethernet support spawns a separate program rawether, written in C, that performs the low-level I/O. Raw ethernet access usually also requires root privileges, and running a small program rawether with such privileges is somewhat less of a security risk than the whole eCos application, the I/O auxiliary, and various dynamically loaded Tcl scripts.

Because all scripts run in a single interpreter, some care has to be taken to avoid accidental sharing of global variables. The best way to avoid problems is to have each script create its own Tcl namespace, so for example the ethernet.tcl script creates a namespace ethernet:: and all variables and procedures reside in this namespace. Similarly the I/O auxiliary itself makes use of a synth:: namespace.

Building and Installation
When an eCos device driver or application code instantiates a device, the I/O auxiliary will attempt to load a matching Tcl script. The third argument to synth_auxiliary_instantiate specifies the type of device, for example ethernet, and the I/O auxiliary will append a .tcl suffix and look for a script ethernet.tcl.

If the device being instantiated is application-specific rather than part of an eCos package, the I/O auxiliary will look first in the current directory, then in ~/.ecos/synth. If it is part of an eCos package then the auxiliary will expect to find the Tcl script and any support files below libexec/eco in the install tree - note that the same install tree must be used for the I/O auxiliary itself and for any device driver support. The directory hierarchy below libexec/eco matches the structure of the eCos repository, allowing multiple versions of a package to be installed to allow for incompatible protocol changes.

The preferred way to build host-side software is to use autoconf and automake. Usually this involves little more than copying the acinclude.m4, configure.in and Makefile.am files from an existing package, for example the synthetic target ethernet driver, and then making minor edits. In acinclude.m4 it may be necessary to adjust the path to the root of the repository. configure.in may require a similar change, and the AC_INIT macro invocation will have to be changed to match one of the files in the new package. A critical macro in this file is ECOS_PACKAGE_DIRS which will set up the correct install directory. Makefile.am may require some more changes, for example to specify the data files that should be installed (including the Tcl script). These files should then be processed using aclocal, autoconf and automake in that order. Actually building the soft-
writing new devices - host

To assist developers, if the environment variable ECOSYNTH_DEVEL is set then a slightly different algorithm is used for locating device Tcl scripts. Instead of looking only in the install tree the I/O auxiliary will also look in the source tree, and if the script there is more recent than the installed version it will be used in preference. This allows developers to modify the master copy without having to run make install all the time.

If a script needs to know where it has been installed it can examine the Tcl variable synth::device_install_dir. This variable gets updated whenever a script is loaded, so if the value may be needed later it should be saved away in a device-specific variable.

Instantiation

The I/O auxiliary will source the device-specific Tcl script when the eCos application first attempts to instantiate a device of that type. The script should return a procedure that will be invoked to instantiate a device.

```
namespace eval ethernet {
...
  proc instantiate { id instance data } {
    ...
    return ethernet::handle_request
  }

return ethernet::instantiate
```

The id argument is a unique identifier for this device instance. It will also be supplied on subsequent calls to the request handler, and will match the return value of synth_auxiliary_instantiate on the target side. A common use for this value is as an array index to support multiple instances of this type of device. The instance and data arguments match the corresponding arguments to synth_auxiliary_instantiate on the target side, so a typical value for instance would be eth0, and data is used to pass arbitrary initialization parameters from target to host.

The actual work done by the instantiation procedure is obviously device-specific. It may involve allocating an interrupt vector, adding a device-specific subwindow to the display, opening a real Linux device, establishing a socket connection to some server, spawning a separate process to handle the actual I/O, or a combination of some or all of the above.

If the device is successfully instantiated then the return value should be a handler for subsequent I/O requests. Otherwise the return value should be an empty string, and on the target-side the synth_auxiliary_instantiate call will return -1. The script is responsible for providing diagnostics explaining why the device could not be instantiated.

Handling Requests

When the target-side calls synth_auxiliary_xchgmsg, the I/O auxiliary will end up calling the request handler for the appropriate device instance returned during instantiation:

```
namespace eval ethernet {
...
  proc handle_request { id request arg1 arg2 txdata txlen max_rxlen } {
    ...
    if { <some condition> } {
      synth::send_reply <error code> 0 ""
      return
```
The `id` argument is the same device id that was passed to the instantiate function, and is typically used as an array index to access per-device data. The `request`, `arg1`, `arg2`, and `max_rxlen` are the same values that were passed to `synth_auxiliary_xchgmsg` on the target-side, although since this is a Tcl script obviously the numbers have been converted to strings. The `txdata` buffer is raw data as transmitted by the target, or an empty string if the I/O operation does not involve any additional data. The Tcl procedures `binary scan`, `string index` and `string range` may be found especially useful when manipulating this buffer. `txlen` is provided for convenience, although `string length $txdata` would give the same information.

The code for actually processing the request is of course device specific. If the target does not expect a reply then the request handler should just return when finished. If a reply is expected then there should be a call to `synth::send_reply`. The first argument is the reply code, and will be turned into a 32-bit integer on the target side. The second argument specifies the length of the reply data, and the third argument is the reply data itself. For some devices the Tcl procedure `binary format` may prove useful. If the reply involves just a code and no additional data, the second and third arguments should be 0 and an empty string respectively.

Attempts to send a reply when none is expected, fail to send a reply when one is expected, or send a reply that is larger than the target-side expects, will all be detected by the I/O auxiliary and result in run-time error messages.

It is not possible for the host-side code to send unsolicited messages to the target. If host-side code needs attention from the target, for example because some I/O operation has completed, then an interrupt should be raised.

## Interrupts

The I/O auxiliary provides a number of procedures for interrupt handling.

```
synth::interrupt_allocate <name>
synth::interrupt_get_max
synth::interrupt_get_devicename <vector>
synth::interrupt_raise <vector>
```

`synth::interrupt_allocate` is normally called during device instantiation, and returns the next free interrupt vector. This can be passed on to the target-side device driver in response to a suitable request, and it can then install an interrupt handler on that vector. Interrupt vector 0 is used within the target-side code for the real-time clock, so the allocated vectors will start at 1. The argument identifies the device, for example `eth0`. This is not actually used internally, but can be accessed by user-initialization scripts that provide some sort of interrupt monitoring facility (typically via the `interrupt hook`). It is possible for a single device to allocate multiple interrupt vectors, but the synthetic target supports a maximum of 32 such vectors.

`synth::interrupt_get_max` returns the highest interrupt vector that has been allocated, or 0 if there have been no calls to `synth::interrupt_allocate`. `synth::interrupt_get_devicename` returns the string that was passed to `synth::interrupt_allocate` when the vector was allocated.

`synth::interrupt_raise` can be called any time after initialization. The argument should be the vector returned by `synth::interrupt_allocate` for this device. It will activate the normal eCos interrupt handling mechanism so, subject to interrupts being enabled and this particular interrupt not being masked out, the appropriate ISR will run.
Note: At this time it is not possible for a device to allocate a specific interrupt vector. The order in which interrupt vectors are assigned to devices effectively depends on the order in which the eCos devices get initialized, and that may change if the eCos application is rebuilt. A future extension may allow devices to allocate specific vectors, thus making things more deterministic. However that will introduce new problems, in particular the code will have to start worrying about requests for vectors that have already been allocated.

Flags and Command Line Arguments

The generic I/O auxiliary code will process the standard command line arguments, and will set various flag variables accordingly. Some of these should be checked by device-specific scripts.

**synth::flag_gui**

This is set when the I/O auxiliary is operating in graphical mode rather than text mode. Some functionality such as filters and the GUI layout are only available in graphical mode.

```tcl
if { $synth::flag_gui } {
    ...
}
```

**synth::flag_verbose**

The user has requested additional information during startup. Each device driver can decide how much additional information, if any, should be produced.

**synth::flag_keep_going**

The user has specified -k or --keep-going, so even if an error occurs the I/O auxiliary and the various device driver scripts should continue running if at all possible. Diagnostics should still be generated.

Some scripts may want to support additional command line arguments. This facility should be used with care since there is no way to prevent two different scripts from trying to use the same argument. The following Tcl procedures are available:

- `synth::argv_defined <name>`
- `synth::argv_get_value <name>`

**synth::argv_defined** returns a boolean to indicate whether or not a particular argument is present. If the argument is the name part of a name/value pair, an = character should be appended. Typical uses might be:

```tcl
if { [synth::argv_defined "-o13"] } {
    ...
}
```

```tcl
if { [synth::argv_defined "-mark="] } {
    ...
}
```

The first call checks for a flag -o13 or --o13 - the code treats options with single and double hyphens interchangeably. The second call checks for an argument of the form -mark=<value> or a pair of arguments -mark <value>. The value part of a name/value pair can be obtained using `synth::argv_get_value`;

```tcl
variable speed 1
if { [synth::argv_defined "-mark="] } {
    set mark [synth::argv_get_value "-mark="]
    if { ![string is integer $mark] || ($mark < 1) || ($mark > 9) } {
        
```
Writing New Devices - host

synth::argy_get_value should only be used after a successful call to synth::argy_defined. At present there is no support for some advanced forms of command line argument processing. For example it is not possible to repeat a certain option such as -v or --verbose, with each occurrence increasing the level of verbosity.

If a script is going to have its own set of command-line arguments then it should give appropriate details if the user specifies --help. This involves a hook function:

```tcl
namespace eval my_device {
    proc help_hook {} {
        puts " -o13 : activate the omega 13 device"
        puts " -mark <speed> : set speed. Valid values are 1 to 9."
    }
    synth::hook_add "help" my_device::help_hook
}
```

The Target Definition File

Most device scripts will want to check entries in the target definition file for run-time configuration information. The Tcl procedures for this are as follows:

```tcl
synth::tdf_has_device <name>
synth::tdf_get_devices
synth::tdf_has_option <devname> <option>
synth::tdf_get_option <devname> <option>
synth::tdf_get_all_options <devname>
```

synth::tdf_has_device can be used to check whether or not the target definition file had an entry synth_device <name>. Usually the name will match the type of device, so the console.tcl script will look for a target definition file entry console. synth::tdf_get_devices returns a list of all device entries in the target definition file.

Once it is known that the target definition file has an entry for a certain device, it is possible to check for options within the entry. synth::tdf_has_option just checks for the presence, returning a boolean:

```tcl
if { [synth::tdf_has_option "console" "appearance"] } {
    ...
}
```

synth::tdf_get_option returns a list of all the arguments for a given option. For example, if the target definition file contains an entry:

```tcl
synth_device console {
    appearance -foreground white -background black
    filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
    filter xyzzy {.*xyzzy.*} -foreground PapayaWhip
}
```

A call synth::tdf_get_option console appearance will return the list {-foreground white -background black}. This list can be manipulated using standard Tcl routines such as llength and lindex. Some options can
Writing New Devices - host

occur multiple times in one entry, for example filter in the console entry. **synth::tdf_get_options** returns a list of lists, with one entry for each option occurrence. **synth::tdf_get_all_options** returns a list of lists of all options. This time each entry will include the option name as well.

The I/O auxiliary will not issue warnings about entries in the target definition file for devices which were not loaded, unless the **-v** or **--verbose** command line argument was used. This makes it easier to use a single target definition file for different applications. However the auxiliary will issue warnings about options within an entry that were ignored, because often these indicate a typing mistake of some sort. Hence a script should always call **synth::tdf_has_option**, **synth::tdf_get_option** or **synth::tdf_get_options** for all valid options, even if some of the options preclude the use of others.

**Hooks**

Some scripts may want to take action when particular events occur, for example when the eCos application has exited and there is no need for further I/O. This is supported using hooks:

```tcl
namespace eval my_device {
    ...
    proc handle_ecos_exit { arg_list } {
        ...
    }
    synth::hook_add "ecos_exit" my_device::handle_ecos_exit
}
```

It is possible for device scripts to add their own hooks and call all functions registered for those hooks. A typical use for this is by user initialization scripts that want to monitor some types of I/O. The available Tcl procedures for manipulating hooks are:

- **synth::hook_define** creates a new hook with the specified name. This hook must not already exist.
- **synth::hook_defined** can be used to check for the existence of a hook. **synth::hook_add** allows other scripts to register a callback function for this hook, and **synth::hook_call** allows the owner script to invoke all such callback functions. A hook must already be defined before a callback can be attached. Therefore typically device scripts will only use standard hooks and their own hooks, not hooks created by some other device, because the order of device initialization is not sufficiently defined. User scripts run from mainrc.tcl can use any hooks that have been defined.
- **synth::hook_call** takes an arbitrary list of arguments, for example:

  ```tcl
  synth::hook_call "ethernet_rx" "eth0" $packet
  ```

The callback function will always be invoked with a single argument, a list of the arguments that were passed to **synth::hook_call**:

```tcl
proc rx_callback { arg_list } {
    set device [lindex $arg_list 0]
    set packet [lindex $arg_list 1]
}
```

Although it might seem more appropriate to use Tcl’s **eval** procedure and have the callback functions invoked with the right number of arguments rather than a single list, that would cause serious problems if any of the
A number of hooks are defined as standard. Some devices will add additional hooks, and the device-specific documentation should be consulted for those. User scripts can add their own hooks if desired.

**exit**
This hook is called just before the I/O auxiliary exits. Hence it provides much the same functionality as `atexit` in C programs. The argument list passed to the callback function will be empty.

**ecos_exit**
This hook is called when the eCos application has exited. It is used mainly to shut down I/O operations: if the application is no longer running then there is no point in raising interrupts or storing incoming packets. The callback argument list will be empty.

**ecos_initialized**
The synthetic target HAL will send a request to the I/O auxiliary once the static constructors have been run. All devices should now have been instantiated. A script could now check how many instances there are of a given type of device, for example ethernet devices, and create a little monitor window showing traffic on all the devices. The `ecos_initialized` callbacks will be run just before the user’s `mainrc.tcl` script. The callback argument list will be empty.

**help**
This hook is also invoked once static constructors have been run, but only if the user specified `-h` or `--help`. Any scripts that add their own command line arguments should add a callback to this hook which outputs details of the additional arguments. The callback argument list will be empty.

**interrupt**
Whenever a device calls `synth::interrupt_raise` the `interrupt` hook will be called with a single argument, the interrupt vector. The main use for this is to allow user scripts to monitor interrupt traffic.

---

### Output and Filters

Scripts can use conventional facilities for sending text output to the user, for example calling `puts` or directly manipulating the central text widget `.main.centre.text`. However in nearly all cases it is better to use output facilities provided by the I/O auxiliary itself:

```tcl
synth::report <msg>
synth::report_warning <msg>
synth::report_error <msg>
synth::internal_error <msg>
synth::output <msg> <filter>
```

**synth::report** is intended for messages related to the operation of the I/O auxiliary itself, especially additional output resulting from `-v` or `--verbose`. If running in text mode the output will go to standard output. If running in graphical mode the output will go to the central text window. In both modes, use of `-l` or `--logfile` will modify the behaviour.

**synth::report_warning**, **synth::report_error** and **synth::internal_error** have the obvious meaning, including prepending strings such as Warning: and Error:. When the eCos application informs the I/O auxiliary that all static constructors have run, if at that point there have been any calls to `synth::error` then the I/O auxiliary will exit. This can be suppressed with command line arguments `-k` or `--keep-going`. **synth::internal_error**
will output some information about the current state of the I/O auxiliary and then exit immediately. Of course it should never be necessary to call this function.

`synth::output` is the main routine for outputting text. The second argument identifies a filter. If running in text mode the filter is ignored, but if running in graphical mode the filter can be used to control the appearance of this output. A typical use would be:

```plaintext
synth::output $line "console"
```

This outputs a single line of text using the `console` filter. If running in graphical mode the default appearance of this text can be modified with the `appearance` option in the `synth_device console` entry of the target definition file. The `System filters` menu option can be used to change the appearance at run-time.

Filters should be created before they are used. The procedures available for this are:

```plaintext
synth::filter_exists <name>
synth::filter_get_list
synth::filter_add <name> [options]
synth::filter_parse_options <options> <parsed_options> <message>
synth::filter_add_parsed <name> <parsed_options>
```

`synth::filter_exists` can be used to check whether or not a particular filter already exists: creating two filters with the same name is not allowed. `synth::filter_get_list` returns a list of the current known filters. `synth::filter_add` can be used to create a new filter. The first argument names the new filter, and the remaining arguments control the initial appearance. A typical use might be:

```plaintext
synth::filter_add "my_device_tx" -foreground yellow -hide 1
```

It is assumed that the supplied arguments are valid, which typically means that they are hard-wired in the script. If instead the data comes out of a configuration file and hence may be invalid, the I/O auxiliary provides a parsing utility. Typical usage would be:

```plaintext
array set parsed_options [list]
set message ""
if { ![synth::filter_parse_options $console_appearance parsed_options message] } {
    synth::report_error "Invalid entry in target definition file $synth::target_definition
    \n    synth_device \"console\", entry \"appearance\"\n    \n    message"
} else {
    synth::filter_add_parsed "console" parsed_options
}
```

On success `parsed_options` will be updated with an internal representation of the desired appearance, which can then be used in a call to `synth::filter_add_parsed`. On failure `message` will be updated with details of the parsing error that occurred.

## The Graphical Interface

When the I/O auxiliary is running in graphical mode, many scripts will want to update the user interface in some way. This may be as simple as adding another entry to the help menu for the device, or adding a new button to the toolbar. It may also involve adding new subwindows, or even creating entire new toplevel windows. These may be simple monitor windows, displaying additional information about what is going on in the system in a graphical format. Alternatively they may emulate actual I/O operations, for example button widgets could be used to emulate real physical buttons.

The I/O auxiliary does not provide many procedures related to the graphical interface. Instead it is expected that scripts will just update the widget hierarchy directly.
So adding a new item to the Help menu involves a \texttt{.menubar.help add} operation with suitable arguments. Adding a new button to the toolbar involves creating a child window in \texttt{.toolbar} and packing it appropriately. Scripts can create their own subwindows and then pack it into one of \texttt{.main.nw}, \texttt{.main.n}, \texttt{.main.ne}, \texttt{.main.w}, \texttt{.main.e}, \texttt{.main.sw}, \texttt{.main.s} or \texttt{.main.se}. Normally the user should be allowed to control this via the target definition file. The central window \texttt{.main.centre} should normally be left alone by other scripts since it gets used for text output.

The following graphics-related utilities may be found useful:

\begin{itemize}
  \item \texttt{synth::load_image <image name> <filename>}
  \item \texttt{synth::register_balloon_help <widget> <message>}
  \item \texttt{synth::handle_help <URL>}
\end{itemize}

\texttt{synth::load_image} can be used to add a new image to the current interpreter. If the specified file has a \texttt{.xbm} extension then the image will be a monochrome bitmap, otherwise it will be a colour image of some sort. A boolean will be returned to indicate success or failure, and suitable diagnostics will be generated if necessary.

\texttt{synth::register_balloon_help} provides balloon help for a specific widget, usually a button on the toolbar.

\texttt{synth::handle_help} is a utility routine that can be installed as the command for displaying online help, for example:

\begin{verbatim}
.menubar.help add command -label "my device" -command \
  [list synth::handle_help "file://$path"]
\end{verbatim}
Writing New Devices - host
Porting

Name

Porting — Adding support for other hosts

Description

The initial development effort of the eCos synthetic target happened on x86 Linux machines. Porting to other platforms involves addressing a number of different issues. Some ports should be fairly straightforward, for example a port to Linux on a processor other than an x86. Porting to Unix or Unix-like operating systems other than Linux may be possible, but would involve more effort. Porting to a completely different operating system such as Windows would be very difficult. The text below complements the eCos Porting Guide.

Other Linux Platforms

Porting the synthetic target to a Linux platform that uses a processor other than x86 should be straightforward. The simplest approach is to copy the existing i386linux directory tree in the hal/synth hierarchy, then rename and edit the ten or so files in this package. Most of the changes should be pretty obvious, for example on a 64-bit processor some new data types will be needed in the basetype.h header file. It will also be necessary to update the toplevel ecos.db database with an entry for the new HAL package, and a new target entry will be needed.

Obviously a different processor will have different register sets and calling conventions, so the code for saving and restoring thread contexts and for implementing setjmp and longjmp will need to be updated. The exact way of performing Linux system calls will vary: on x86 linux this usually involves pushing some registers on the stack and then executing an int 0x080 trap instruction, but on a different processor the arguments might be passed in registers instead and certainly a different trap instruction will be used. The startup code is written in assembler, but needs to do little more than extract the process’ argument and environment variables and then jump to the main linux_entry function provided by the architectural synthetic target HAL package.

The header file hal_io.h provided by the architectural HAL package provides various structure definitions, function prototypes, and macros related to system calls. These are correct for x86 linux, but there may be problems on other processors. For example a structure field that is currently defined as a 32-bit number may in fact may be a 64-bit number instead.

The synthetic target’s memory map is defined in two files in the include/pkgconf subdirectory. For x86 the default memory map involves eight megabytes of read-only memory for the code at location 0x1000000 and another eight megabytes for data at 0x2000000. These address ranges may be reserved for other purposes on the new architecture, so may need changing. There may be some additional areas of memory allocated by the system for other purposes, for example the startup stack and any environment variables, but usually eCos applications can and should ignore those.

Other HAL functionality such as interrupt handling, diagnostics, and the system clock are provided by the architectural HAL package and should work on different processors with few if any changes. There may be some problems in the code that interacts with the I/O auxiliary because of lurking assumptions about endianness or the sizes of various data types.

When porting to other processors, a number of sources of information are likely to prove useful. Obviously the Linux kernel sources and header files constitute the ultimate authority on how things work at the system call level. The GNU C library sources may also prove very useful: for a normal Linux application it is the C library that provides the startup code and the system call interface.
Other Unix Platforms

Porting to a Unix or Unix-like operating system other than Linux would be somewhat more involved. The first requirement is toolchains: the GNU compilers, gcc and g++, must definitely be used; use of other GNU tools such as the linker may be needed as well, because eCos depends on functionality such as prioritizing C++ static constructors, and other linkers may not implement this or may implement it in a different and incompatible way. A closely related requirement is the use of ELF format for binary executables: if the operating system still uses an older format such as COFF then there are likely to be problems because they do not provide the flexibility required by eCos.

In the architectural HAL there should be very little code that is specific to Linux. Instead the code should work on any operating system that provides a reasonable implementation of the POSIX standard. There may be some problems with program startup, but those could be handled at the architectural level. Some changes may also be required to the exception handling code. However one file which will present a problem is hal_io.h, which contains various structure definitions and macros used with the system call interface. It is likely that many of these definitions will need changing, and it may well be appropriate to implement variant HAL packages for the different operating systems where this information can be separated out. Another possible problem is that the generic code assumes that system calls such as cyg_hal_sys_write are available. On an operating system other than Linux it is possible that some of these are not simple system calls, and instead wrapper functions will need to be implemented at the variant HAL level.

The generic I/O auxiliary code should be fairly portable to other Unix platforms. However some of the device drivers may contain code that is specific to Linux, for example the PF_PACKET socket address family and the ethertap virtual tunnelling interface. These may prove quite difficult to port.

The remaining porting task is to implement one or more platform HAL packages, one per processor type that is supported. This should involve much the same work as a port to another processor running Linux.

When using other Unix operating systems the kernel source code may not be available, which would make any porting effort more challenging. However there is still a good chance that the GNU C library will have been ported already, so its source code may contain much useful information.

Windows Platforms

Porting the current synthetic target code to some version of Windows or to another non-Unix platform is likely to prove very difficult. The first hurdle that needs to be crossed is the file format for binary executables: current Windows implementations do not use ELF, instead they use their own format PE which is a variant of the rather old and limited COFF format. It may well prove easier to first write an ELF loader for Windows executables, rather than try to get eCos to work within the constraints of PE. Of course that introduces new problems, for example existing source-level debuggers will still expect executables to be in PE format.

Under Linux a synthetic target application is not linked with the system’s C library or any other standard system library. That would cause confusion, for example both eCos and the system’s C library might try to define the printf function, and introduce complications such as working with shared libraries. For much the same reasons, a synthetic target application under Windows should not be linked with any Windows DLL’s. If an ELF loader has been specially written then this may not be much of a problem.

The next big problem is the system call interface. Under Windows system calls are generally made via DLL’s, and it is not clear that the underlying trap mechanism is well-documented or consistent between different releases of Windows.

The current code depends on the operating system providing an implementation of POSIX signal handling. This is used for I/O purposes, for example SIGALRM is used for the system clock, and for exceptions. It is not known what equivalent functionality is available under Windows.
Porting

Given the above problems a port of the synthetic target to Windows may or may not be technically feasible, but it would certainly require a very large amount of effort.
Porting
XXXVII. M68000 Architectural Support
Overview

Name

Overview — eCos Support for the M68K Family of Processors

Description

The original Motorola 68000 processor was released in 1979, and featured the following:

- Eight general purpose 32-bit data registers, %D0 to %D7. Seven 32-bit address registers %A0 to %A6, with %A7 dedicated as the stack pointer. A 16-bit status register.
- A linear address space, limited to 24-bits because the chip package only had 24 address pins. Hence the processor could address 16 megabytes of memory.
- No separate address space for I/O operations. Instead devices are accessed just like memory via the main address and data buses.
- 16-bit external data bus, even though the data registers were 32 bits wide.
- A CISC variable-length instruction set with no less than 14 different addressing modes (although of course the terms RISC and CISC were not yet in common use).
- Separate supervisor and user modes. The processor actually has two distinct stack pointer registers %A7, and the mode determines which one gets used.
- An interrupt subsystem with support for vectored and prioritized interrupts.

The 68000 processor was used in several microcomputers of its time, including the original Apple Macintosh, the Commodore Amiga, and the Atari ST. Over the years numerous variants have been developed. The core instruction set has remained essentially unchanged. Some of the variants have additional instructions. The development of MMUs led to changes in exception handling. In more recent variants, notably the Freescale ColdFire family, some infrequently used instructions and addressing modes have been removed.

- The 68008 reduced the widths of the external data and address buses to 8 bits and 20 bits respectively, giving the processor slow access to only one megabyte.
- The 68010 (1982) added virtual memory support.
- In the 68020 (1984) both the address and data buses were made 32-bits wide. A 256-byte instruction cache was added, as were some new instructions and addressing modes.
- The 68030 (1987) included an on-chip mmu and a 256-byte data cache.
- The 68040 (1991) added hardware floating point (previous processors relied on an external coprocessor or on software emulation). It also had larger caches and an improved mmu.
- The 68060 (1994) involved an internally very different superscalar implementation of the architecture, but few changes at the interface level. It also contained support for power management.
- There have been numerous 683xx variants for embedded use, with on-chip peripherals like UARTs and timers. The cpu core of these variants is also known as cpu32.
- The MCFxxxx ColdFire series (1995) resembles a stripped-down 68060, with some instructions and addressing modes removed to allow for a much smaller and more efficient implementation. Various hardware units such as the and FPU and MMU have become optional.
Overview

eCos only provides support for some of these variants, although it should be possible to add support for additional variants with few or no changes to the architectural HAL package.

The architectural HAL provides support for those features which are common to all members of the 68000 and ColdFire families, and for certain features which are present on some but not all members. A typical eCos configuration will also contain: a variant HAL package with support code for a family of processors, for example MCFxxxx; possibly a processor HAL package with support for one specific processor, for example the MCF5272; and a platform HAL which contains the code needed for a specific hardware platform such as the m5272c3.
Configuration

Name

Options — Configuring the M68K Architectural Package

Loading and Unloading the Package

The M68K architectural HAL package CYGPKG_HAL_M68K should be loaded automatically when eCos is configured for M68K-based target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. CYGPKG_HAL_M68K serves primarily as a container for lower-level HALs and has only a small number of configuration options.

Stacks

By default the architectural HAL provides a single block of memory to act as both the startup stack and the interrupt stack. The variant, processor or platform HAL may override this. For example if there are several banks of RAM with different performance characteristics it may be desirable to place the interrupt stack in fast RAM rather than in ordinary RAM.

The assembler startup code sets the stack pointer to the startup stack before switching to C code. This stack used for all HAL initialization, running any C++ static constructors defined either by eCos or by the application, and the cyg_start entry point. In configurations containing the eCos kernel cyg_start will enable interrupts, activate the scheduler and threads will then run on their own stacks. In non-kernel single-threaded applications the whole system continues to run on the startup stack.

When an interrupt occurs the default behaviour is to switch to a separate interrupt stack. This behaviour is controlled by the common HAL configuration option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK. It reduces the stack requirements of all threads in the system, at the cost of some extra instructions during interrupt handling. In kernel configurations the startup stack is no longer used once the scheduler starts running so its memory can be reused for the interrupt stack. To handle the possibility of nested interrupts the interrupt handling code will detect if it is already on the interrupt stack, so in non-kernel configurations it is also safe to use the same area of memory for both startup and interrupt stacks. This leads to the following scenarios:

1. If interrupt stacks are enabled via CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK and the interrupt stack is not provided by the variant, processor or platform HAL then a single block of memory will be used for both startup and interrupt stacks. The size of this block is determined by the common HAL configuration option CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE, with a default value CYGNUM_HAL_DEFAULT_INTERRUPT_STACK_SIZE provided by the M68K architectural HAL.

2. If the use of an interrupt stack is disabled then the M68K architectural HAL will provide just the startup stack, unless this is done by the variant, processor or platform HAL. The size of the startup stack is controlled by CYGNUM_HAL_M68K_STARTUP_STACK_SIZE.

3. Otherwise the interrupt and/or startup stacks are provided by other packages and it is up to those packages to provide configuration options for setting the sizes.
Configuration

Floating Point Support

There are many variants of the basic M68K architecture. Some of these have hardware floating point support. Originally this came in the form of a separate 68881 coprocessor, but with modern variants it will be part of the main processor chip. If the processor does not have hardware floating point then software emulation will be used instead.

If the processor on the target hardware has a floating point unit then the variant or processor HAL will implement the CDL interface CYGINT_HAL_M68K_VARIANT_FPU. This allows the architectural HAL and other packages to do the right thing on different hardware.

Saving and restoring hardware floating point context increases interrupt and dispatch latency, code size, and data size. If the application does not actually use floating point then these overheads are unnecessary, and can be suppressed by disabling the configuration option CYGIMP_HAL_M68K_FPU_SAVE. Some applications do use floating point but only in one thread. In that scenario it is also unnecessary to save the floating point context during interrupts and context switches, so the configuration option can be disabled.

The exact behaviour of the hardware floating point unit is determined by the floating point control register %fpcr. By default this is initialized to 0, giving IEE754 standard behaviour, but another initial value can be specified using the configuration option CYGNUM_HAL_M68K_FPU_CR_DEFAULT. For details of the various bits in this control register see appropriate hardware documentation. eCos assumes that the control register does not change on a per-thread basis and hence the register is not saved or restored during interrupt handling or a context switch.

Warning
At the time of writing eCos has not run on an M68K processor with hardware floating point so the support for this is untested.

Other Options

There are configuration options to change the compiler flags used for building this packages. The M68K architectural HAL package does not define any other configuration options that can be manipulated by the user. It does define a number of interfaces such as CYGINT_HAL_M68K_USE_STANDARD_PLATFORM_STUB_SUPPORT which can be used by lower levels of the M68K HAL hierarchy to enable certain functionality within the architectural package. Usually these are of no interest to application developers.
The HAL Port

Name
HAL Port — Implementation Details

Description
This documentation explains how the eCos HAL specification has been mapped onto M68K hardware, and should be read in conjunction with that specification. It also describes how variant, processor and platform HALs can modify the default behaviour.

eCos support for any given target will involve either three or four HAL packages: the architectural HAL, the platform HAL, the variant HAL, and optionally a processor HAL. This package, the architectural HAL, provides code and definitions that are applicable to all M68K processors. The platform HAL provides support for one specific board, for example an M5272C3 evaluation board, or possibly for a number of almost-identical boards. The processor HAL, if present, serves mainly to provide details of on-chip peripherals including the interrupt controller. The variant HAL provides functionality that is common to a group of processors, for example all MCFxxxx processors have very similar UARTs and hence can share HAL diagnostic code. There is no fixed specification of what should go into the variant HAL versus the processor HAL. For simplicity the description below only refers to variant HALs, but the work may actually happen in a processor HAL instead.

As a design goal lower-level HALs can always override functionality that is normally provided higher up. For example the architectural HAL will provide the required eCos `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros, but these can be provided lower down instead. Many but not all ColdFire processors have the `ff1` and `bitrev` instructions which allow for a more efficient implementation than the default architectural ones. In some areas such as handling context switching the architectural HAL will usually provide the basic functionality but it may be extended by lower HALs, for example to add support for the multiply-accumulate units present in certain ColdFire processors.

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_cache.h`, `cyg/hal/hal_io.h` and `cyg/hal/arch.inc`. These automatically include an equivalent header file from the variant HAL, for example `cyg/hal/var_arch.h`. The variant HAL header will in turn include processor and platform-specific headers. This means that application developers and other packages can simply include the architectural HAL headers without needing to know about variants or platforms. It also allows the variant and platform HALs to override architectural settings.

The port assumes that eCos and application code always runs in supervisor mode, with full access to all hardware and special registers.

Data Types
For eCos purposes all M68K processors are big-endian and 32-bit, so the default data types in `cyg/infra/cyg_type.h` are used. Some variants have external bus widths less than 32-bit, but this does not affect the architectural HAL.

When porting to another variant it is possible to override some or all of the type definitions. The variant HAL needs to implement the CDL interface `CYGINT_HAL_M68K_VARIANT_TYPES` and provide a header file `cyg/hal/var_basetype.h`.

721
The HAL Port

Startup and Exception Vectors

The conventional bootstrap mechanism involves a table of exception vectors at the base of memory. The first two words of this table give the initial program counter and stack pointer. In a typical embedded system the hardware is arranged such that non-volatile flash memory is found at location 0x0 so it is the start of flash that contains the exception vectors and the boot code. The table of exception vectors is used subsequently for interrupt handling and for hardware exceptions such as attempts to execute an illegal instruction. There are a number of common scenarios:

1. On systems with very limited memory flash may remain mapped at location 0 and the table of exception vectors remains mapped there as well. The M68K architecture defines the table to have 256 entries and hence it occupies 1K of memory, but in reality many of the entries are unused so part of the table may get used for code instead. Since the whole exception vector table is in read-only memory parts of the eCos interrupt and exception handling mechanisms have to be statically initialized and macros like HAL_VSR_SET are not available.

2. As a minor variation of the previous case, flash remains at location 0 but the table of exception vectors gets remapped elsewhere in the address space, usually RAM. This allows HAL_VSR_SET to operate normally but at the cost of increased memory usage. The exception vector table in flash only contains two entries, for the initial program counter and stack pointer. The exception vector table in RAM typically gets initialized at run-time.

3. On systems with more memory it is conventional to rearrange the address map during bootstrap. The flash gets relocated, typically to near the end of the address space, and RAM gets placed at location 0 instead. The exception vector table stays at location 0 but is now in RAM and gets initialized at run-time. The bootstrap exception vector table in flash again only needs two entries. A variation places the RAM elsewhere in the address space and moves the exception vector table there, leaving location 0 unused. This provides some protection against null pointer accesses in errant code.

As a further complication, larger systems typically support different startup types. The application can be linked against a ROM startup configuration and placed directly in flash, as before. Alternatively there may be a ROM monitor such as RedBoot in the flash, taking care of initial bootstrap. The user’s application is linked against a RAM startup configuration, loaded into RAM via the ROM monitor, and debugged using functionality provided by the ROM monitor. Yet another possibility involves a RAM startup application but it gets loaded and run via a hardware debug technology such as BDM, and the ROM monitor is either missing or not used.

The exact hardware details, the various startup types, the steps needed for low-level hardware initialization, and so on are not known to the architectural HAL. Hence although the architectural HAL does provide the basic framework for startup, much of the work is done via macros provided by lower-level HAL packages and those macros are likely to depend on various configuration options. Rather than try to enumerate all the various combinations here it is better to look at the actual code in vectors.S and in appropriate variant, processor or platform HALs. vectors.S is responsible for any low-level initialization that needs to happen. This includes setting up a standard C environment with the stack pointer set to the startup stack in working RAM, making sure all statically initialized global variables have the correct values, and that all uninitialized global variables are zeroed. Once the C environment has been set up the code jumps to hal_m68k_c_startup in file hal_m68k.c which completes the initialization and jumps to the application entry point.

Interrupt Handling

The M68K architecture reserves a 1K area of memory for 256 exception vectors. These are used for internal and external interrupts, exceptions, software traps, and special operations such as reset handling. Some of the vectors have well-defined uses. However when it comes to interrupt handling the details will depend on the processor variant and on the platform, and the appropriate package documentation should be consulted for full
The HAL Port

details. Most platforms will not use the full set of 256 vectors, instead re-using some of this memory for other purposes.

By default the exception vectors are located at location 0, but some variants allow the vectors to be located elsewhere. This is managed by an M68K-specific macro CYG_HAL_VSR_TABLE. The default value is 0, but a variant HAL can provide an alternative value.

The standard eCos macros HAL_VSR_GET and HAL_VSR_SET just manipulate one of the 256 entries in the table of exception vectors. Hence it is usually possible to replace the default handlers for exceptions and traps in addition to interrupt handlers. hal_intr.h provides #define's for the more common exception vectors, and additional ones can be provided by the platform or variant. It is the responsibility of the platform or variant HAL to initialize the table, and to provide the HAL_VSR_SET_TO_ECOS_HANDLER macro since that requires knowledge of the default table entries.

It should be noted that in some configurations the table of exception vectors may reside in read-only memory so entries cannot be changed. If so then the HAL_VSR_SET and HAL_VSR_SET_TO_ECOS_HANDLER macros will not be defined. Portable code may need to consider this possibility and test for the existence of these macros before using them.

The architectural HAL provides an entry point hal_m68k_interrupt_vsr in the file hal_arch.S. When an interrupt occurs the original 68000 pushed the program counter and the status register on to the stack, and then called the VSR via the exception table. On newer variants some additional information is pushed, including details of the interrupt source. hal_m68k_interrupt_vsr assumes the latter and can be used directly as the VSR on these newer variants. On older variants a small trampoline is needed which pushes the additional information and then jumps to the generic VSR. Interpreting the additional information is handled via an assembler macro hal_context_extract_isr_vector_shl2 which should be defined by the variant, matching the behaviour of the hardware or the trampoline.

At the architecture level there is no fixed mapping between VSR and ISR vectors. Instead that is left to the variant or platform HAL. The architectural HAL does provide default implementations of HAL_INTERRUPT_ATTACH, HAL_INTERRUPT_DETACH and HAL_INTERRUPT_IN_USE since these just involve updating a static table.

By default the interrupt state control macros HAL_DISABLE_INTERRUPTS, HAL_RESTORE_INTERRUPTS, HAL_ENABLE_INTERRUPTS and HAL_QUERY_INTERRUPTS are implemented by the architectural HAL, and simply involve updating the status register. Disabling interrupts involves setting the three IPL bits to 0x07. Enabling interrupts involves setting those bits to a smaller value, CYGNUM_HAL_INTERRUPT_DEFAULT_IPL_LEVEL, which defaults to 0.

HAL_DISABLE_INTERRUPTS has no effect on non-maskable interrupts. This causes a problem because parts of the system assume that all normal interrupt sources are affected by this macro. If the target hardware can raise non-maskable interrupts then it is the responsibility of application code to install a suitable VSR and handle non-maskable interrupts entirely within the application, bypassing the usual eCos ISR and DSR mechanisms.

The architectural HAL does not provide any support for the interrupt controller management macros like HAL_INTERRUPT_MASK. These can only be implemented on a per-variant, per-processor or per-platform basis.

### Exception Handling

Synchronous exception handling is done in much the same way as interrupt handling. The architectural HAL provides a generic entry point hal_m68k_exception_vsr. On some variants this can be used directly as the exception VSR, on others it will be called via a small trampoline.

The details of exception handling vary widely from one variant to the next. Some variants push a great deal of additional information on to the stack for certain exceptions, but not all. The pushed program counter may correspond to the specific instruction that caused the exception, or the next instruction, or there may be only a loose correlation because of buffered writes. The architectural HAL makes no attempt to cope with all these
differences, although some variants may provide more advanced support. Otherwise if an exception needs to be handled in a very specific way then it is up to the application to install a suitable VSR and handle the exception directly.

**Stacks and Stack Sizes**

cyg/hal/hal_arch.h defines values for minimal and recommended thread stack sizes, CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL. These values are specific to the current configuration, and are affected mainly by options related to interrupt handling.

By default eCos uses a separate interrupt stack, although this can be disabled through the configuration option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK. When an interrupt or exception occurs eCos will save the context on the current stack and then switch to the interrupt stack before calling the appropriate ISR interrupt handler. This means that thread stacks can be significantly smaller because there is no need to worry about interrupt handling overheads, just the thread context. However switching the stack does require some extra work and hence increases the interrupt latency. Disabling the interrupt stack removes this processing overhead but requires larger stack sizes. It depends on the application whether or not this is a sensible trade off.

By default eCos does not allow nested interrupts, but this can be controlled via the configuration option CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING. Supporting nested interrupts requires larger thread stacks, especially if the separate interrupt stack is also disabled.

Although the M68K has enough registers for typical operation, the calling conventions are memory-oriented. In particular all arguments are pushed on the stack rather than held in registers, and the return address is also pushed rather than ending up in a link register. To allow for this the recommended minimum stack sizes are a little bit larger than for some other architectures. Variant HALs cannot directly affect these stack sizes. However the sizes do depend in part on the size of a thread context, so if for example the processor supports hardware floating point and support for that is enabled then the stack sizes will increase.

Usually the M68K architectural HAL will provide a single block of memory which acts as both the startup and interrupt stack, and there are configuration options to control the size of this block. Alternatively a variant, processor or platform HAL may define either or both of _HAL_M68K_STARTUP_STACK_ and _HAL_M68K_INTERRUPT_STACK_BASE_ if for some reason the stacks should not be placed in ordinary RAM.

**Thread Contexts and Setjmp/Longjmp**

A typical thread context consists of the following:

1. The integer context. This consists of the data registers %d0 to %d7 and the address registers %a0 to %a6. The stack pointer register %a7 does not have to be saved explicitly since it is implicit in the pointer to the saved context.

   The caller-save registers are %d0, %d1, %a0, %a1, %a7 and the status register. The remaining registers are callee-save. Function arguments are always passed on the stack. The result is held in %d0.

2. Floating point context, consisting of eight 64-bit floating point registers %fp0 to %fp7 and two support registers %fpsr and %fp iar. Support for this is only relevant if the processor variant has a hardware floating point unit, and even then saving floating point context is optional and can be disabled using a configuration option CYGIMP_HAL_M68K_FPU_SAVE. The control register %fpcr is not saved as part of the context. It is assumed that a single %fpcr value, usually 0, will be used throughout the application.

   The architectural HAL provides support for the hardware floating point unit. The variant or processor HAL should implement the CDL interface CYGINT_HAL_M68K_VARIANT_FPU if this hardware unit is actually present.
3. Some M68K variants have additional hardware units, for example the multiply-accumulate units in certain ColdFire processors. The architectural HAL allows the context to be extended through various macros such as \texttt{HAL\_CONTEXT\_OTHER}.

4. The status register \texttt{\%sr} and the program counter. These are special because when an interrupt occurs the hardware automatically pushes these onto the stack, but exactly what gets pushed depends on the variant. \texttt{setjmp} and \texttt{longjmp} only deal with the integer and fpu contexts. It is assumed that any special hardware units will only be used by application code, not by the compiler. Hence it is the responsibility of application code to define and implement appropriate \texttt{setjmp} semantics for these units. The variant HAL package can override the default implementations if necessary.

When porting to a new M68K variant, if this has a hardware floating point unit then the variant HAL should implement the CDL interface \texttt{CYGINT\_HAL\_M68K\_VARIANT\_FPU}, thus enabling support provided by the architectural HAL. If the variant has additional hardware units involving state that should be preserved during a context switch or when an interrupt occurs, the variant HAL should define a number of macros.

The header file \texttt{cyg/hal/var\_arch.h} should define \texttt{HAL\_CONTEXT\_OTHER}, \texttt{HAL\_CONTEXT\_OTHER\_SIZE}, and \texttt{HAL\_CONTEXT\_OTHER\_INIT}, either directly or via \texttt{cyg/hal/proc\_arch.h}. The assembler header file \texttt{cyg/hal/var.inc} should define a number of macros such as \texttt{hal\_context\_other\_save\_caller}. For details of these macros see the architectural file \texttt{hal\_arch.S}.

Variants also need to define exactly how the status register and program counter are saved onto the stack when an interrupt or exception occurs. This is handled through C macros \texttt{HAL\_CONTEXT\_PCSR\_SIZE}, \texttt{HAL\_CONTEXT\_PCSR\_RTE\_ADJUST}, and \texttt{HAL\_CONTEXT\_PCSR\_INIT}, and a number of assembler macros such as \texttt{hal\_context\_pcsr\_save\_sr}. Again the architectural files \texttt{cyg/hal/hal\_arch.h} and \texttt{hal\_arch.S} provide more details of these.

### Bit Indexing

For performance reasons the \texttt{HAL\_LSBIT\_INDEX} and \texttt{HAL\_MSBIT\_INDEX} macros are implemented using assembler functions. A variant HAL can override the default definitions if, for example, the variant has special instructions to perform these operations.

### Idle Thread Processing

The default \texttt{HAL\_IDLE\_THREAD\_ACTION} implementation is a no-op. A variant HAL may override this, for example to put the processor into sleep mode. Alternative implementations should consider exactly how this macro gets used in eCos kernel code.

### Clock Support

The architectural HAL cannot provide the required clock support because it does not know what timer hardware may be available on the target hardware. Instead this is left to either the variant or platform HAL, depending on whether the processor has a suitable on-chip timer or whether an off-chip timer has to be used.

### HAL I/O

The M68K architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus are wired appropriately for a big-endian processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.
The variant, processor and platform equivalents of the cyg/hal/hal_io.h header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

**Cache Handling**

If the processor has a cache then the variant HAL should implement the CDL interface CYGINT_HAL_M68K_VARIANT_CACHE. This causes the architectural header cyg/hal/hal_cache.h to pick up appropriate definitions from cyg/hal/var_cache.h. The architectural header will provide null defaults for anything not defined by the variant.

**Linker Scripts**

The architectural HAL will generate the linker script for eCos applications. This involves the architectural file m68k.ld and a .ldi memory layout file provided lower down, typically by the platform HAL. It is the LDI file which specifies the types and amount of memory available and which places code and data in appropriate places, but most of the hard work is done via macros provided by the architectural m68k.ld file.

**Diagnostic Support**

The architectural HAL does not implement diagnostic support. Instead this is left to the variant or platform HAL, depending on whether suitable peripherals are available on-chip or off-chip.

**SMP Support**

The M68K port does not have SMP support.

**Debug Support**

The M68K architectural HAL package provides basic support only for gdb stubs. There is no support for more advanced debug features like hardware watchpoints.

The generic gdb support in the common HAL requires a platform header <cyg/hal/plf_stub.h. In practice there is rarely any need for the contents of this file to change between platforms so the architectural HAL can provide a suitable default. It will do so if the CDL interface CYGINT_HAL_M68K_USE_STANDARD_PLATFORM_STUB_SUPPORT is implemented.

**HAL_DELAY_US Macro**

The architectural HAL provides a default implementation of the standard HAL_DELAY_US macro using a simply busy loop. To use this support a lower-level HAL should define _HAL_M68K_DELAY_US_LOOPS_, typically a small number of about 20 but it will need to be calibrated during the porting process. If the processor has a cache then the lower-level HAL may also define _HAL_M68K_DELAY_US_LOOPS_UNCACHED_, for the case when a delay loop is triggered while the cache is disabled.
Profiling Support

The M68K architectural HAL implements the `mcount` function, allowing profiling tools like gprof to determine the application’s call graph. It does not implement the profiling timer. Instead that functionality needs to be provided by the variant or platform HAL. The implementation of `mcount` requires a dedicated frame pointer register so code should be compiled without the `-fomit-frame-pointer` flag.

Other Functionality

The M68K architectural HAL only implements the functionality provided by the eCos HAL specification and does not export any extra functionality.
The HAL Port
XXXVIII. Freescale MCFxxxx Variant Support
MCFxxxx ColdFire Processors

Name

CYGPKG_HAL_M68K_MCFxxxx — eCos Support for Freescale MCFxxxx Processors

Description

The Freescale ColdFire family is a range of processors including the MCF5206 and the MCF5282. From a programmer’s perspective these processors all share basically the same processor core, albeit with minor differences in the instruction set. They differ in areas like performance, on-chip peripherals and caches. Even when it comes to peripherals there is a lot of commonality. For example many but not all Coldfire processors use the same basic interrupt controller(s) as the MCF5282. Similarly the on-chip UARTs tend to use the same basic design although there are variations in the number of UARTs, the fifo sizes, and in certain details.

The MCFxxxx variant HAL package CYGPKG_HAL_M68K_MCFxxxx provides support for various features that are common to many but not all Coldfire processors. This includes HAL diagnostics via an on-chip UART and interrupt controller management for those processors which have MCF5282-compatible controllers. The variant HAL complements the M68K architectural HAL package. An eCos configuration should also include a processor-specific HAL package such as CYGPKG_HAL_M68K_MCF5272 to support the chip-specific peripherals and cache details, and a platform HAL package such as CYGPKG_HAL_M68K_M5272C3 to support board-level details like external memory chips. The processor or platform HAL can override the functionality provided by the variant HAL.

Configuration

The MCFxxxx variant HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

On most ColdFire platforms the variant HAL will provide the HAL diagnostics support via one of the UARTs. Some platforms may provide their own HAL diagnostics facility, for example output via an LCD. The variant HAL diagnostics support is active if the processor or platform implements the CYGINT_HAL_M68K_MCFxxxx_DIAGNOSTICS_USE_DEFAULT interface. It is also active only in configurations which do not rely on an underlying rom monitor such as RedBoot: if CYGSEM_HAL_USE_ROM_MONITOR is enabled then the default diagnostics channel will automatically be inherited from RedBoot. The variant HAL then provides a number of configuration options related to diagnostics:

CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT

This selects the destination for HAL diagnostics. The number of UARTs available depends on the processor, and on any given board some of the UARTs may not be connected. Hence the variant HAL looks for configuration options CYGHWR_HAL_M68K_MCFxxxx_UART0, CYGHWR_HAL_M68K_MCFxxxx_UART1 and CYGHWR_HAL_M68K_MCFxxxx_UART2 to see which on-chip UARTs are actually available on the processor and target hardware, and uses this information to let the user select a UART.

When a UART is in use as the HAL diagnostics channel, that UART should not be used for any other purpose. In particular application code should avoid using it for I/O via the serial driver.
When a UART is selected for HAL diagnostics this option specifies the default baud rate. The most common setting is 38400. That provides a compromise between performance and reliability, especially in electrically noisy environments such as an industrial environment or a test farm. Some platforms may define CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_DEFAULT_BAUD to handle scenarios where another default baud rate is preferable, typically for compatibility with existing software.

Usually the HAL diagnostics channel is driven in polled mode but in some scenarios interrupts are required. For example, when debugging an application over a serial line on top of the gdb stubs provided by RedBoot, the user should be able to interrupt the application with a control-C. The application will not be polling the HAL diagnostics UART at this point so instead the eCos interrupt management code interacts with the gdb stubs to do the right thing. This configuration option selects the interrupt priority. It should be noted that on some processors with MCF5282-compatible interrupt controllers all priorities for enabled interrupts should be unique, and it is the responsibility of application developers to ensure this condition is satisfied.

The HAL Port

This section describes how the MCFxxxx variant HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and processor HAL documentation.

HAL I/O

The cyg/hal/var_io.h header provides various definitions for on-chip peripherals, where the current processor has peripherals compatible with the MCF5282’s. This header is automatically included by the architectural cyg/hal/hal_io.h so other packages and application code will usually only include the latter.

It is up to the processor HAL to specify exactly what var_io.h should export. For example the MCF5213’s proc_io.h header contains the following:

```c
#define HAL_MCFxxxx_HAS_MCF5282_INTC 1
#define HAL_MCFxxxx_INTC0_BASE (HAL_MCF521x_IPSBAR + 0x00000C00)
```

This enables support within the variant HAL for a single MCF5282-compatible interrupt controller, and cases var_io.h to export symbols such as:

```c
#include HAL_MCFxxxx_HAS_MCF5282_INTC
// Two 32-bit interrupt mask registers
#define HAL_MCFxxxx_INTCx_IMRH 0x0008
#define HAL_MCFxxxx_INTCx_IMRL 0x000C
...
#define HAL_MCFxxxx_INTCx_ICRxx_IL_MASK (0x07 << 3)
#define HAL_MCFxxxx_INTCx_ICRxx_IL_SHIFT 3
```

Symbols such as HAL_MCFxxxx_INTCx_IMRH can be used to access the relevant hardware registers via HAL_READ_UINT32 and HAL_WRITE_UINT32. Symbols like HAL_MCFxxxx_INTCx_ICRxx_IL_MASK can be used to generate or decode the contents of the hardware registers.
The header file does mostly use a naming convention, but is not guaranteed to be totally consistent. There may also be discrepancies with the documentation because the manuals for the various Coldfire processors are not always consistent about their naming schemes. All I/O definitions provided by the variant HAL will start with `HAL_MCFxxxx_`, followed by the name of the peripheral. If a peripheral is likely to be a singleton, for example an on-chip flash unit, then the name is unadorned. If there may be several instances of the peripheral then the name will be followed by a lower case x. For example:

```c
#define HAL_MCFxxxx_CFM_CR 0x0000
...```

```c
#define HAL_MCFxxxx_UARTx_UMR 0x00
```  

Register names will be relative to some base address such as `HAL_MCFxxxx_CFM_BASE` or `HAL_MCFxxxx_UART0_BASE`, so code accessing a register would look like:

```c
HAL_READ_UINT32(HAL_MCFxxxx_CFM_BASE + HAL_MCFxxxx_CFM_PROT, reg);
...```

```c
HAL_WRITE_UINT8(base + HAL_MCFxxxx_UARTx_UTB, '*');
```  

Usually the register names are singletons, but in some cases such as the interrupt controller priority registers there may be multiple instances of the register and the names will be suffixed appropriately. For example `HAL_MCFxxxx_INTCx_ICRxx_IL_MASK` indicates the field IL within one of the ICR registers within one of the interrupt controllers.

As mentioned earlier the processor HAL’s `proc_io.h` will control which definitions are exported by `var_io.h`. Sometimes the processor HAL will then go on to undefine or redefine some of the symbols, to reflect incompatibilities between the processor’s devices and the equivalent devices on the MCF5282. There may also be additional symbols for the devices, and there will be additional definitions for any processor-specific hardware. In particular GPIO pin handling is handled by the processor HAL, not by the variant HAL. Application developers should examine `proc_io.h` as well as `var_io.h` and the processor-specific documentation to see exactly what I/O definitions are provided. When porting to a new Coldfire processor it is best to start with an existing processor HAL and copy code as appropriate. A search for `_HAS_` in `var_io.h` will also be informative.

### Thread Contexts and Setjmp/Longjmp

All MCFxxxx processors support interrupts and exceptions in a uniform way. When an interrupt or exception occurs the hardware pushes the current program counter, the status register, and an additional 16-bit word containing information about the interrupt source, for a total of 64 bits. Hence the PCSR part of a thread context consists of two 32-bit integers, and the variant HAL provides appropriate C and assembler macros to examine and manipulate these.

Not all MCFxxxx processors have hardware floating point, so support for this is left to the processor HAL package. Some MCFxxxx processors have additional hardware units such as a multiply-accumulator, but these are not currently supported by eCos.

### HAL Diagnostics

The various MCFxxxx processors usually have one or more UARTs based on very similar hardware. The variant HAL package can provide HAL diagnostic support using such a UART. There are some minor differences such as fifo sizes, and the UARTs will be accessed at different memory locations. These differences are handled by a small number of macros provided by the processor and platform HAL.
The MCFxxxx variant HAL only provides HAL diagnostic support via a UART if the processor or platform HAL does not provide an alternative implementation. That copes with situations where the on-chip UARTs are not actually accessible on the target board and an alternative communication channel must be used.

If the variant HAL should implement HAL diagnostics then the processor or platform HAL should implement the CDL interface CYGINT_HAL_M68K_MCFxxxx_DIAGNOSTICS_USE_DEFAULT. It should also define one or more of CYGHWR_HAL_M68K_MCFxxxx_UART0, CYGHWR_HAL_M68K_MCFxxxx_UART1 and CYGHWR_HAL_M68K_MCFxxxx_UART2, and ensure that any multi-purpose GPIO pins are set correctly. The variant HAL will take care of the rest.

Cache Handling

The various MCFxxxx processors all have very different caches, so support for these is deferred to the processor HAL.

Exceptions

All MCFxxxx processors support synchronous exceptions in a uniform way, with the hardware pushing sufficient information on to the stack to identify the nature of the exception. This means that the architectural entry point hal_m68k_exception_vsr can be used as the default VSR for all exceptions, with no need for separate trampoline functions.

The variant HAL does not provide any special support for recovering from exceptions.

Interrupts

All MCFxxxx processors supports interrupts in a uniform way. When an interrupt occurs the hardware pushes sufficient information on to the stack to identify the interrupt. Therefore the architectural entry point hal_m68k_interrupt_vsr can be used as the default VSR for all interrupts, with the variant just supplying a small number of macros that allow the generic code to extract details of the interrupt source. There is no need for separate trampoline functions for every interrupt source.

On processors which have MCF5282-compatible interrupt and edge port modules the variant HAL can provide the HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_SET_LEVEL, HAL_INTERRUPT_ACKNOWLEDGE and HAL_INTERRUPT_CONFIGURE macros. There is support for processors with a single interrupt controller or with two separate interrupt controllers. Otherwise these macros are left to the processor HAL. The allocation of interrupt vectors to the various on-chip devices is also a characteristic of the processor HAL. proc_intr.h should be consulted for appropriate definitions, for example CYGNUM_HAL_ISR_UART0.

The mask and umask operations are straightforward: if the interrupt controller has the SIMR and CIMR registers those will be used; otherwise the IRM registers will be updated by a read-modify-write cycle. The acknowledge macro is only relevant for external interrupts coming in via the edge port module and will clear the interrupt by writing to the EPIER register. There is no simple way to clear interrupts generated by the on-chip peripherals, so that is the responsibility of the various device drivers or of application code. The configure macro is only relevant for external interrupts and involves manipulating the edge port module.

The HAL_INTERRUPT_SET_LEVEL macro is used implicitly by higher level code such as cyg_interrupt_create. With MCF5282-compatible interrupt controllers the priority level corresponds to the ICRxx register. The exact format depends on the processor. Interrupt priorities corresponding to IPL level 7 are non-maskable. Such interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms. Instead application code will have to install a custom VSR and manage the entire interrupt.
Some MCF5282-compatible interrupt controllers have a major restriction: all interrupt priorities within each controller must be unique. If two interrupts go off at the same time and have exactly the same priority then the controllers’ behaviour is undefined. In a typical application some of the interrupts will be handled by eCos device drivers while others will be handled directly by application code. Since eCos cannot know which interrupts may get used, it cannot allocate unique priorities. Instead this has to be left to the application developer. eCos does provide configuration options such as CYGNUM_KERNEL_COUNTERS_CLOCK_ISR_PRIORITY and CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY to provide control over the eCos-managed interrupts, and provides default values for these which are unique.

Caution
Non-unique interrupt priorities can lead to very confusing system behaviour. For example on an MCF5282, if the PIT3 system clock (interrupt 0x3a) and ethernet RX frame (interrupt 0x1b) are accidentally given the same priority and go off at the same time, the interrupt controller may actually issue an interrupt 0x3b, the bitwise or of the two interrupt numbers. That interrupt belongs to the on-chip flash module. There may not be an installed handler for that interrupt at all, and even if there is a handler it will only manipulate the flash hardware and not clear the system clock and ethernet interrupts. Hence the system is likely to go into a spin, continually trying to service the wrong interrupt. To track down such problems during debugging it may prove useful to install a breakpoint on the hal_arch_default_isr function.

Clock Support
On processors with an MCF5282-compatible programmable interrupt timer module or PIT, the variant HAL can provide the HAL_CLOCK_INITIALIZATION, HAL_CLOCK_RESET, HAL_CLOCK_READ and HAL_CLOCK_LATENCY macros. These macros are used by the eCos kernel to implement the system clock and may be used for other purposes in non-kernel configurations. When multiple timers are available it is up to the processor or platform HAL to select which one gets used for the system clock. It is also up to the processor or platform HAL to provide various clock-related configuration options such as CYGNUM_HAL_RTC_PERIOD. Those options need to take into account the processor clock speed, which is usually a characteristic of the platform and hence not known to the variant HAL.

When porting to a new Coldfire processor, the processor or platform HAL should define the symbols CYGNUM_HAL_INTERRUPT_RTC, _HAL_MCFxxxx_CLOCK_PIT_BASE_, and _HAL_MCFxxxx_CLOCK_PIT_PRE_. Existing ports can be examined for more details.

Reset
On processors with an MCF5282-compatible reset module or RST, the variant HAL can provide the HAL_PLATFORM_RESET macro. That macro is typically used by the gdb stubs support inside RedBoot to reset the hardware between debug sessions, ensuring that each session runs in as close to pristine hardware as possible. The macro uses the SOFTRST bit of the RCR register.

Bit Indexing
By default the variant HAL will provide versions of HAL_LSBIT_INDEX and HAL_MSBIT_INDEX which are more efficient than the default ones in the architectural HAL. The implementation uses the ffl1 and bitrevl instructions. If the Coldfire processor does not support these instructions then the processor HAL should define _HAL_M68K_MCFxxxx_NO_FF1_.

735
Other Issues
The MCFxxxx variant HAL does not affect the implementation of data types, stack size definitions, idle thread processing, linker scripts, SMP support, system startup, or debug support.

Other Functionality
The MCFxxxx variant HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.
XXXIX. Freescale MCF5272 Processor Support
The MCF5272 ColdFire Processor

Name

CYGPKG_HAL_M68K_MCF5272 — eCos Support for the Freescale MCF5272 Processor

Description

The MCF5272 is one member of the Freescale MCFxxxx ColdFire range of processors. It comes with a number of on-chip peripherals including 2 UARTs, ethernet, and USB slave. The processor HAL package CYGPKG_HAL_M68K_MCF5272 provides support for features that are specific to the MCF5272. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package, for example CYGPKG_HAL_M68K_M5272C3 to support board-level details like the external memory chips.

Configuration

The MCF5272 processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The component CYGPKG_HAL_M68K_MCF5272_HARDWARE contains configuration options for the available hardware. This includes all GPIO pin settings, with defaults provided by the platform HAL. In turn the pin settings are used to determine defaults for other hardware settings, for example which of the two on-chip uarts are usable. Users can override these settings if necessary, subject to any constraints imposed by the platform HAL, but care has to be taken that the resulting configuration still matches the actual hardware.

The option CYGIMP_HAL_M68K_MCF5272_IDLE controls what happens in configurations containing the eCos kernel when the idle thread runs, i.e. when there is nothing for the processor to do until the next interrupt comes in. Usually the processor made to sleep, halting the cpu but leaving all peripherals active.

The package contains a single configuration option CYGFUN_HAL_M68K_MCF5272_PROFILE_TIMER. This controls the support for gprof-based profiling. By default it is active and enabled if the configuration contains the gprof profiling package, otherwise inactive. The relevant code uses hardware timer 2, so that timer is no longer available for application code. If the timer is required but a platform HAL provides an alternative implementation of the profiling support then this option can be disabled.

The HAL Port

This section describes how the MCF5272 processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/proc_io.h specifies which generic MCFxxxx devices are present, and provides details of MCF5272-specific devices. This header file is automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter.

It should be noted that the Freescale documentation is occasionally confusing when it comes to numbering devices. For example the four on-chip timers are numbered TMR0 to TMR3, but in the interrupt controller the
corresponding interrupts are numbered TMR1 to TMR4. The eCos port consistently starts numbering at 0, so these interrupts have been renamed TMR0 to TMR3.

Interrupt Handling

The header file `cyg/hal/proc_intr.h` provides VSR and ISR vector numbers for all interrupt sources. The VSR vector number, for example `CYGNUM_HAL_VECTOR_TMR0`, should be used for calls like `cyg_interrupt_get_vsr`. It corresponds directly to the M68K exception number. The ISR vector number, for example `CYGNUM_HAL_ISR_TMR0`, should be used for calls like `cyg_interrupt_create`. This header file is automatically included by the architectural header `cyg/hal/hal_intr.h`, and other packages and application code will normally just include the latter.

The eCos HAL macros `HAL_INTERRUPT_MASK`, `HAL_INTERRUPT_UNMASK`, `HAL_INTERRUPT_SET_LEVEL`, `HAL_INTERRUPT_ACKNOWLEDGE`, and `HAL_INTERRUPT_CONFIGURE` are implemented by the processor HAL. The mask and unmask operations are straightforward, simply manipulating the on-chip interrupt controller. The acknowledge and configure macros are only relevant for external interrupts; internal interrupts generated by on-chip devices do not need to be acknowledged. The set-level operation, used implicitly by higher level code such as `cyg_interrupt_create`, is mapped on to M68K IPL levels so interrupts can be given a priority between 1 and 7. Priority 7 corresponds to non-maskable interrupts and must be used with care: such interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms; instead application code will have to install a custom VSR and manage the entire interrupt.

Clock Support

The processor HAL provides support for the eCos system clock. This always uses hardware timer 3, which should not be used directly by application code. If gprof-based profiling is in use then that will use hardware timer 2. Timers 0 and 1 are never used by eCos so application code is free to manipulate these as required.

Some of the configuration options related to the system clock, for example `CYGNUM_HAL_RTC_PERIOD`, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Cache Handling

The MCF5272 has a small instruction cache of 1024 bytes. This is fully supported by the processor HAL. There is no data cache.

Idle Thread Support

The configuration option `CYGIMP_HAL_M68K_MCF5272_IDLE` controls what happens when the kernel idle thread runs. The default behaviour is to put the processor to sleep until the next interrupt.

Profiling Support

The MCF5272 processor HAL provides a profiling timer for use with the gprof profiling package. This uses hardware timer 2, so application code should not manipulate this timer if profiling is enabled. The M68K architectural HAL implements the `mcount` function so profiling is fully supported on all MCF5272-based platforms.
Other Issues
The MCF5272 processor HAL does not affect the implementation of data types, stack size definitions, linker scripts, SMP support, system startup, or debug support. The architectural HAL’s bit index instructions are used rather than the MCFxxxx variant HAL’s versions since the MCF5272 does not implement the ff1 and bitrev instructions.

Other Functionality
The MCF5272 processor HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.
XL. Freescale M5272C3 Board Support
Overview

Name

eCos Support for the Freescale M5272C3 Board — Overview

Description

The Freescale M5272C3 board has an MCF5272 ColdFire processor, 4MB of external SDRAM, 2MB of external flash memory, and connectors plus required support chips for all the on-chip peripherals. By default the board comes with its own dBUG ROM monitor, located in the bottom half of the flash.

For typical eCos development a RedBoot image is programmed into the top half of the flash memory, and the board is made to boot this image rather than the existing dBUG monitor. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

In a typical setup the bottom half of the flash memory is reserved for the dBUG ROM monitor and is not accessible to eCos. That leaves four flash blocks of 256K each. Of these one is used for the RedBoot image and another is used for managing the flash and holding RedBoot fconfig values. The remaining two blocks at 0xFFF40000 and 0xFFF80000 can be used by application code.

By default eCos will only support the four megabytes of external SDRAM present on the initial versions of the board, accessible at location 0x00000000. Later versions come with 16MB. If all 16MB of memory are required then the ACR0 register needs to be changed. The default value is controlled by the configuration option CYGNUM_HAL_M68K_M5272C3_ACR0, but this option is only used during ROM startup so in a typical setup it would be necessary to rebuild and update RedBoot. Alternatively the register can be updated by application code, preferably using a high priority static constructor to ensure that the extra memory is visible before any code tries to use that memory. It will also be necessary to change the memory layout so that the linker knows about the additional memory.

By default the 4K of internal SRAM is mapped to location 0x20000000 using the RAMBAR register. This is not used by eCos or by RedBoot so can be used by application code. The M68K architectural HAL has an iram1.c testcase to illustrate the linker script support for this. The internal 16K of ROM is left disabled by default because its contents are of no use to most applications. The on-chip peripherals are mapped at 0x10000000 via the MBAR register.

There is a serial driver CYGPKG_DEVS_SERIAL_MCFxxxx which supports both on-chip UARTs. One of the UARTs, usually uart0, can be used by RedBoot for communication with the host. If this UART is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the M5272C3 target.

There is an ethernet driver CYGPKG_DEVS_ETH_MCFxxxx for the on-chip ethernet device. This driver is also loaded automatically when configuring for the M5272C3 target. The M5272C3 board does not have a unique MAC address, so a suitable address has to be programmed into flash via RedBoot’s fconfig command.

eCos manages the on-chip interrupt controller. Timer 3 is used to implement the eCos system clock, but timers 0, 1 and 2 are unused and left for the application. The GPIO pins are manipulated only as needed to get the UARTs and ethernet working. eCos will reset the remaining on-chip peripherals (DMA, USB, PLCI, QSPI and PWM) during system startup or soft reset but will not otherwise manipulate them.
**Overview**

**Tools**

The M5272C3 port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.2.1, m68k-elf-gdb version 5.3, and binutils version 2.13.1.

By default eCos is built using the compiler flag `-fomit-frame-pointer`. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option `CYGBLD_GLOBAL_CFLAGS` avoids such debug problems.
Setup

Name

Setup — Preparing the M5272C3 board for eCos Development

Overview

In a typical development environment the M5272C3 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger m68k-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
<th>Use</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>RedBoot running from the board’s flash</td>
<td>redboot_ROM.ecm</td>
<td>redboot_rom.bin</td>
</tr>
<tr>
<td>dBUG</td>
<td>Used for initial setup</td>
<td>redboot_DBUG.ecm</td>
<td>redboot_dbug.srec</td>
</tr>
<tr>
<td>RAM</td>
<td>Used for upgrading ROM version</td>
<td>redboot_RAM.ecm</td>
<td>redboot_ram.bin</td>
</tr>
<tr>
<td>ROMFFE</td>
<td>RedBoot running from the board’s flash at 0xFFE00000</td>
<td>redboot_ROMFFE.ecm</td>
<td>redboot_romffe.bin</td>
</tr>
</tbody>
</table>

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 19200 baud. The ROM and RAM versions run at 38400 baud. These baud rates can be changed via the configuration option CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD and rebuilding RedBoot. By default RedBoot will use the board’s terminal port, corresponding to uart0, but this can also be changed via the configuration option CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT. On an M5272C3 platform RedBoot also supports ethernet communication and flash management.

Initial Installation

This process assumes that the board still has its original dBUG ROM monitor and does not require any special debug hardware. It leaves the existing ROM monitor in place, allowing the setup process to be repeated just in case that should ever prove necessary.

Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 cable between the M5272C3 terminal port and the host PC. A suitable cable is supplied with the board. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 19200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Make sure that the jumper next to the flash chip is set for bootstrap from the bottom of flash, location 0xFFE00000. The details of this jumper depend on the revision of the board, so the supplied board documentation should be consulted for more details. Apply power to the board and you
Setup

should see a dBUG> prompt.

Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

dBUG> dl
Escape to local host and send S-records now...

The required S-records file is redboot_debug.srec, which is normally supplied with the eCos release in the loaders directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

S-record download successful!
dBUG>

It is now possible to run the RAM-resident version of RedBoot:

dBUG> go 0x20000
+FLASH configuration checksum error or invalid key
Ethernet eth0: MAC address 00:00:00:00:00:03
Can’t get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version v2_0_1 - built 09:55:34, Jun 24 2003

Platform: M5272C3 (Freescale MCF5272)

RAM: 0x00000000-0x00400000, 0x0003f478-0x003bd000 available
FLASH: 0xffe00000 - 0x00000000, 8 blocks of 0x00040000 bytes each.

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the fis init -f command:

RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0xffff40000-0xffffc0000: ..
... Erase from 0x00000000-0x00000000:
... Erase from 0xffffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xffffc0000: .
RedBoot>

The flash chip on the M5272C3 board is slow at erasing flash blocks so this operation can take some time. At the end the block of flash at location 0xFFFFC0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot’s non-volatile configuration values:

RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x03
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xfffc0000: .

RedBoot>

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

RedBoot> load -r -m ymodem -b %{freememlo}

The file redboot_rom.bin should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

Raw file loaded 0x0003f800-0x000545a3, assumed entry at 0x0003f800

RedBoot> xyzModem - CRC mode, 2(SOH)/84(STX)/0(CAN) packets, 5 retries

Once RedBoot has been loaded into RAM it can be programmed into flash:

RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xffff0000-0xffff40000: .
... Program from 0x0003f800-0x00007f800 at 0xffff00000: .
... Erase from 0xffffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xfffc0000: .

RedBoot>

The flash-resident version of RedBoot has now programmed at location 0xFFF00000, and the flash info block at 0xFFFCC0000 has been updated. The initial setup is now complete. Power off the board and set the flash jumper to boot from location 0xFFF00000 instead of 0xFFE00000. Also set the terminal emulator to run at 38400 baud (the usual baud rate for RedBoot), and power up the board again.

*Ethernet eth0: MAC address 00:00:00:00:00:03
Can’t get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version v2_0_0 - built 09:57:50, Jun 24 2003

Platform: M5272C3 (Freescale MCF5272)

RAM: 0x000000000-0x000400000, 0x0000b4000-0x003bd000 available
FLASH: 0xffff0000 - 0xffff000000, 8 blocks of 0x00040000 bytes each.

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.
Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using dBUG to load the dBUG version of RedBoot `redboot_dbug.srec`. Alternatively the existing RedBoot install can be used to load a RAM-resident version, `redboot_ram.bin`.

The ROMFFE version of RedBoot can be installed at location 0xFFE00000, replacing dBUG. This may be useful if the system needs more flash blocks than are available with the usual ROM RedBoot. Installing this RedBoot image will typically involve a BDM-based utility.

### Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```bash
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new m5272c3 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf5272/m5272c3/v2_0_1/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_dbug.srec`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

### BDM

An alternative to debugging an application on top of RedBoot is to use a BDM hardware debug solution. On the eCos side this requires building the configuration for RAM startup and with `CYGSEM_HAL_USE_ROM_MONITOR` disabled. Note that a RAM build of RedBoot automatically has the latter configuration option disabled, so it is possible to run a RAM RedBoot via BDM and bypass the dBUG stages of the installation process.

On the host-side the details depend on exactly which BDM solution is in use. Typically it will be necessary to initialize the hardware prior to downloading the eCos application, either via a configuration file or by using gdb macros. The file `misc/bdm.gdb` in the platform HAL defines example gdb macros.
Configuration

Name
Configuration — Platform-specific Configuration Options

Overview
The M5272C3 platform HAL package is loaded automatically when eCos is configured for an M5272C3 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup
The M5272C3 platform HAL package supports four separate startup types:

RAM
This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xFFF00000 and boots from that location. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use eCos’ virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
This startup type can be used for finished applications which will be programmed into flash at location 0xFFF00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMFFE
This is a variant of the ROM startup type which can be used if the application will be programmed into flash at location 0xFFE00000, overwriting the board’s dBUG ROM monitor.

DBUG
This is a variant of the RAM startup which allows applications to be loaded via the board’s dBUG ROM monitor rather than via RedBoot. It exists mainly to support the dBUG version of RedBoot which is needed during hardware setup. Once the application has started it will take over all the hardware, and it will not depend on any services provided by dBUG. This startup type does not provide gdb debug facilities.

RedBoot and Virtual Vectors
If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this op-
Configuration

tion is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then one of the serial ports will be claimed for HAL diagnostics. By default eCos will use the terminal port, corresponding to uart0. The auxiliary port, uart1, can be selected instead via the configuration option CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT. The baud rate for the selected port is controlled by CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD.

Flash Driver

The platform HAL package contains flash driver support. By default this is inactive, and it can be made active by loading the generic flash package CYGPKG_IO_FLASH.

Special Registers

The MCF5272 processor has a number of special registers controlling the cache, on-chip RAM and ROM, and so on. The platform HAL provides a number of configuration options for setting these, for example CYGNUM_HAL_M68K_M5272C3_RAMBAR controls the initial value of the RAMBAR register. These options are only used during a ROM or ROMFFE startup. For a RAM startup it will be RedBoot that initializes these registers, so if the default values are not appropriate for the target application then it will be necessary to rebuild RedBoot with new settings for these options. Alternatively it should be possible to reprogram some or all of the registers early on during startup, for example by using a high-priority static constructor.

One of the special registers, MBAR, cannot be controlled via a configuration option. Changing the value of this register could have drastic effects on the system, for example moving the on-chip peripherals to a different location in memory, and it would be very easy to end up with inconsistencies between RedBoot and the eCos application. Instead the on-chip peripherals are always mapped to location 0x10000000.

System Clock

By default the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_PERIOD, the number of microseconds between clock ticks. Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

- **-mcpu=5272**
  
  The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. For an MCF5272 processor -mcpu=5272 should be used.

- **-malign-int**
  
  This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most
published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

-omit-frame-pointer

Traditionally the %A6 register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing -omit-frame-pointer from the default flags will make debugging easier, but the generated code may be worse.
The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the M5272C3 hardware, and should be read in conjunction with that specification. The M5272C3 platform HAL package complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF5272 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services: the UARTs and the ethernet device will not be reinitialized because they may be in use by RedBoot for communication with the host.

For a ROM or ROMFFE startup the HAL will perform additional initialization, setting up the external DRAM and programming the various internal registers. The values used for most of these registers are configurable. Full details can be found in the exported headers cyg/hal/plf.inc and cyg/hal/proc.inc.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

external SDRAM

This is mapped to location 0x00000000. The first 384 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM and ROMFFE startup all remaining SDRAM is available. For RAM and DBUG startup available SDRAM starts at location 0x00020000, with the bottom 128K reserved for use by either the RedBoot or dBUG ROM monitors.

on-chip peripherals

These are accessible at location 0x10000000 onwards, as per the defined symbol HAL_MCFxxxx_MBAR. This address cannot easily be changed during development because both the ROM monitor and the application must use the same address. The %mbar system register is initialized appropriately during a ROM or ROMFFE startup.

on-chip SRAM

The 4K of internal SRAM are normally mapped at location 0x20000000. The %rambar register is initialized during a ROM startup using the value of the configuration option CYGNUM_HAL_M68K_M5272C3_RAMBAR. Neither eCos nor RedBoot use the internal SRAM so all of it is available to application code.
The HAL Port

on-chip ROM

Usually this is left disabled since its contents are of no interest to most applications. If it is enabled then it is usually mapped at location 0x21000000. The $rombar register is initialized during a ROM startup using the value of the configuration option CYGNUM_HAL_M68K_M5272C3_ROMBAR.

off-chip Flash

This is located at the top of memory, location 0xFFE00000 onwards. For ROM and RAM startups it is assumed that a jumper is used to disable the bottom half of the flash, so location 0xFFE00000 is actually a mirror of 0xFFF00000. For ROMFFE and DBUG startups all of the flash is visible. By default the flash block at location 0xFFF00000 is used to hold RedBoot or another ROM startup application, and the block at location 0xFFF00000 is used to hold flash management data and the RedBoot fconfig variables. The blocks at 0xFFF400000 and 0xFFF800000 can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 3, which should not be used directly by application code. The gprof-based profiling code uses timer 2, so that is only available when not profiling. Timers 0 and 1 are never used by eCos so application code is free to manipulate these as required. The actual HAL macros for managing the clock are provided by the MCF5272 processor HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed.

Other Issues

The M5272C3 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MCF5272 processor HAL, the MCFxxxx variant HAL, and the M68K architectural HAL documentation should be consulted for further details.

Other Functionality

The platform HAL package also provides a flash driver for the off-chip AMD AM29PL160C flash chip. This driver is inactive by default, and only becomes active if the configuration includes the generic flash support CYGPKG_IO_FLASH.
XLI. SuperH SH4-202 MicroDev Board Support
The HAL Port
Overview

Name

eCos Support for the SuperH SH4-202 MicroDev Board — Overview

Description

The SuperH SH4-202 MicroDev board (henceforth just "MicroDev") has an SH4-202 processor, 64MB of external SDRAM, 32MB of external flash memory, an SMSC LAN91C111 ethernet controller and connectors plus required support chips for all the on-chip peripherals.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 128 blocks of 256k bytes each. In a typical setup, the first flash block is used for the ROM RedBoot image and the second is used to store a version of RedBoot that can run out of RAM. The topmost two blocks are used to manage the flash and hold RedBoot fconfig values. The remaining 124 blocks between 0xA0080000 and 0xA1F7FFFF can be used by application code.

The board is fitted with a PLCC socket suitable for an EEPROM (or PROM) such as the 1Mbit ST M29WO10B. This is enabled by toggling two DIP switches, after which the EEPROM is mapped into the same address as the flash memory. Therefore, the flash is not accessible if booting from the EEPROM.

There is a serial driver CYGPKG_DEVS_SERIAL_SH_SCIF which supports the on-chip serial device. This device can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the MicroDev target.

There is an ethernet driver CYGPKG_DEVS_ETH_SH_MICRODEV for the on-chip ethernet device. This driver is also loaded automatically when configuring for the MicroDev target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock, and timer 1 is used to implement a microsecond delay function. Timer 2 is unused and left for the application. Other on-chip devices (FEMI, EMI, INTC, TMU, CAC, UBC) are initialized only as far as is necessary for eCos to run. Other devices (eg RTC, DMAC, etc) are not touched.

Tools

The MicroDev port is intended to work with GNU tools configured for an sh-elf target. The original port was done using sh-elf-gcc version 3.2.1, sh-elf-gdb version 5.3, and binutils version 2.13.1.
**Setup**

**Name**

Setup — Preparing the MicroDev board for eCos Development

**Overview**

In a typical development environment, the MicroDev board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger `sh-elf-gdb`. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory. Alternatively RedBoot may be programmed into a PLCC EEPROM and inserted into socket U21, although in that case, the flash memory is not accessible.

The following RedBoot configurations are supported:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
<th>Use</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>RedBoot running from the board’s flash</td>
<td>redboot_ROM.ecm</td>
<td>redboot_ROM.bin</td>
</tr>
<tr>
<td>EEPROM</td>
<td>RedBoot running from the board’s socketed EEPROM</td>
<td>redboot_EEPROM.ecm</td>
<td>redboot_EEPROM.bin</td>
</tr>
<tr>
<td>RAM</td>
<td>Used for upgrading ROM version</td>
<td>redboot_RAM.ecm</td>
<td>redboot_RAM.bin</td>
</tr>
</tbody>
</table>

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_SH_SH4_SCIF_BAUD_RATE` and rebuilding RedBoot. RedBoot also supports ethernet communication and flash management.

**Initial Installation**

**Flash Installation**

This process assumes that the board is connected to a SuperH Micro Probe. The Micro Probe should be set up as described in Appendix A of the "SH4 Development Tools User Guide". You should also have access to the SuperH development tools since it is necessary to use the version of GDB that comes with those tools to access the Micro Probe, `sh-elf-gdb` will not work.

Programming the RedBoot ROM monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 null modem cable between the MicroDev serial port and the host PC. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking).

Now run the `sh4gdb` command, giving it the name of the RAM redboot ELF file, connect to the Micro Probe, load the executable and run it. The entire session should look like this:
The required `redboot_RAM.elf` file is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below.

If this sequence fails in any way then check the setup and connections of the Micro Probe. If it is successful then you should see the following printed out on the serial line:

```
+FLASH configuration checksum error or invalid key
... waiting for BOOTP information
Ethernet eth0: MAC address 00:08:ee:00:0b:37
Can't get BOOTP info for device!
```

If the ethernet cable is not plugged in there may be a fairly long wait after the "... waiting for BOOTP information" message. At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the `fis init -f` command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue {y/n}? y
*** Initialize FLASH Image System
... Unlock from 0xa1fc0000-0xa2000000: .
... Erase from 0xa1fc0000-0xa2000000: .
... Program from 0x8bfbf000-0x8bfff000 at 0xa1fc0000: .
... Lock from 0xa1fc0000-0xa2000000: .
RedBoot>
```
At the end, the block of flash at location 0xA1FC0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot’s non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Console baud rate: 38400
DNS server IP address:
Set eth0 network hardware address [MAC]: false
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from 0xa1f80000-0xa1f81000: .
... Erase from 0xa1f80000-0xa1f81000: .
... Program from 0x8bfb2000-0x8bfb3000 at 0xa1f80000: .
... Lock from 0xa1f80000-0xa1f81000: .
RedBoot>
```

For most of these configuration variables, the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled and, instead, RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work, it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m xmodem -b %{freememlo}
```

The file `redboot_ROM.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the X-modem protocol.

```
Raw file loaded 0x8812d000-0x8814e32f, assumed entry at 0x8812d000
xyzModem - CRC mode, 1064(SOH)/0(STX)/0(CAN) packets, 2 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
    An image named ‘RedBoot’ exists - continue (y/n)? y
... Erase from 0xa0000000-0xa0040000: .
... Program from 0x8812d000-0x8816d000 at 0xa0000000: .
... Unlock from 0xa1fc0000-0xa2000000: .
... Erase from 0xa1fc0000-0xa2000000: .
... Program from 0x8bfbf000-0x8bfff000 at 0xa1fc0000: .
... Lock from 0xa1fc0000-0xa2000000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0xA0000000, and the flash info block at 0xA1FC0000 has been updated. The initial setup is now complete. Power off the Micro Probe and reset the MicroDev board using S6. You should see the following:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:08:ee:00:0b:37
Can’t get BOOTP info for device!
```
RedBoot (tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:22:57, Sep 8 2003


RAM: 0x88000000-0x8c000000, 0x8800db98-0x8bfb1000 available
FLASH: 0xa0000000 - 0xa2000000, 128 blocks of 0x00040000 bytes each.

RedBoot>

When RedBoot issues its prompt, it is also ready to accept connections from sh-elf-gdb, allowing applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using the Micro Probe. Alternatively, the existing RedBoot install can be used to load the RAM-resident version. You can even install the RAM resident RedBoot in the "RedBoot[backup]" flash region. See the RedBoot documentation for instruction on how to do this.

EEPROM Installation

The board has a 32-pin PLCC socket suitable for an EEPROM, silk screened U21. To use RedBoot running from EEPROM, you must first program the file redboot_EEPROM.bin (normally supplied with the eCos release in the loaders directory) into the EEPROM using an appropriate programmer. No byte swapping is required. If RedBoot needs to be rebuilt, then instructions for this are supplied below, and the import file redboot_EEPROM.ecm should be used.

To configure the board to boot from the EEPROM instead of flash, you must power off the board and change the following DIP switch settings, which may both be found on DIP switch 2 (silk screened S2): switch 2 (silk screened FEMI SIZ1) should be set to ON, which will change the access width for FEMI area 0 from 32-bit to 8-bit; switch 6 (silk screened FPGA SW3) should be set to OFF to configure the FPGA to map memory accesses for FEMI area 0 to point at the EEPROM instead of flash. In this mode, it is no longer possible to access flash memory as the EEPROM is mapped into the same area in the address space.

Note that it is usually preferable to boot from flash instead of EEPROM as flash is accessed 32-bits at a time, whereas the EEPROM is accessed 8-bits at a time, which therefore affects performance as this requires 4 times as many read cycles.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the RAM version of RedBoot are:

$ mkdir redboot_ram
$ cd redboot_ram
$ ecosconfig new sh4_202_md redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/sh/sh4_202_md/v2_0_2/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make

At the end of the build the install/bin subdirectory should contain the file redboot.bin.
Rebuilding the ROM versions involves basically the same process. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot.bin`. Make sure you don’t mix up the different `redboot.bin` files; rename them to something more memorable such as `redboot_RAM.bin` and `redboot_ROM.bin`. 
Setup
Configuration

Name
Configuration — Platform-specific Configuration Options

Overview
The MicroDev platform HAL package is loaded automatically when eCos is configured for an sh4_202_md target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup
The MicroDev platform HAL package supports two separate startup types:

RAM
This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xA0000000 and boots from that location. sh-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
This startup type can be used for finished applications which will be programmed into flash at location 0xA0000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors
If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver
The MicroDev board contains 32Mb of Intel StrataFlash, specifically, two E28F128 parts in parallel. The CYGPKG_DEVS_FLASH_STRATA package contains all the code necessary to support these parts and the CYGPKG_DEVS_FLASH_SH_MICRODEV package contains definitions that customize the driver to the MicroDev board.
Configuration

Note that if booting from EEPROM instead of flash, the flash driver will not be able to detect or use the flash parts.

**Ethernet Driver**

The MicroDev board contains an SMSC LAN91C111 ethernet device. The CYGPKG_DEVS_ETH_SMSC_LAN91CXX package contains all the code necessary to support this part and the CYGPKG_DEVS_ETH_SH_MICRODEV package contains definitions that customize the driver to the MicroDev board.

**System Clock**

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

**Compiler Flags**

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

- `-m4`

  The sh-elf-gcc compiler supports many variants of the SH architecture, from the SH2 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m4` is the correct option for the SH4-202.
The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MicroDev hardware, and should be read in conjunction with that specification. The MicroDev platform HAL package complements the SH architectural HAL and the SH4 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external DRAM and programming the various internal registers. The values used for most of these registers are assigned fixed values from a table in the header cyg/hal/platform.inc.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

off-chip Flash

This is located at address 0x00000000 of the physical memory space and is therefore accessible in the P1 region at location 0x80000000. An uncached shadow of this memory is available in the P2 region at 0xA0000000. The contents of the flash are organized as described earlier.

off-chip EEPROM

If selected by the DIP switches, this occupies the same addresses as the off-chip flash, and the flash is no longer visible.

external SDRAM

This is located at address 0x08000000 of the physical memory space and is therefore accessible in the P1 region at location 0x88000000. An uncached shadow of this memory is available in the P2 region at 0xA8000000. The first 256 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining SDRAM is available. For RAM startup, available SDRAM starts at location 0x80100000, with the bottom 1MB reserved for use by RedBoot.

on-chip peripherals

These are accessible via the P4 region at location 0xE0000000 onwards.
The HAL Port

off-chip peripherals

The ethernet device is located at 0xA7500000. The FPGA interrupt controller is located at 0x06110000. These are the only off-chip peripherals accessed by eCos. All others are left untouched.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 0, which should not be used directly by application code. Timer 1 is used to implement a microsecond resolution busy delay service. Timer 2 is not used by eCos so application code is free to manipulate this as required. The actual HAL macros for managing the clock are provided by the SH architecture processor HAL.

There is a software model of the structure of the SH family clock supply subsystem which performs the correct calculations to yield not only the inputs for the CPU clock but also the peripheral clocks fed to the serial device, memory controllers and other devices. The values for the master crystal, the PLL multipliers and various dividers are supplied by the platform HAL. Some care must be taken in defining these since wrong values will cause the timers and the SCIF baud rate to be miscalculated. If the OSCAR chip switches are changed from the default then the value of CYGHWR_HAL_SH_OOC_XTAL must be changed to match.

Other Issues

The MicroDev platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SH4 variant HAL, and the SH architectural HAL documentation should be consulted for further details.

It should be noted that the floating point support in the SH HAL has a caveat that, if the FPSCR register is changed, it may get reverted at a later stage by certain operations performed by the GCC compiler. This behaviour is intentional as the alternative would be to update the GCC compiler’s internal state about the FPSCR at every context switch which would be expensive for a feature that is unlikely to be used frequently. If the FPSCR is to be changed by the application, the developer should call the function __set_fpscr(int), passing it the new FPSCR value.
XLII. SA11X0 USB Device Driver
The HAL Port
SA11X0 USB Device Driver

Name
SA11X0 USB Support — Device driver for the on-chip SA11X0 USB device

SA11X0 USB Hardware

The Intel StrongARM SA11x0 family of processors is supplied with an on-chip USB slave device, the UDC (USB Device Controller). This supports three endpoints. Endpoint 0 can only be used for control messages. Endpoint 1 can only be used for bulk transfers from host to peripheral. Endpoint 2 can only be used for bulk transfers from peripheral to host. Isochronous and interrupt transfers are not supported.

Caution
Different revisions of the SA11x0 silicon have had various problems with the USB support. The device driver has been tested primarily against stepping B4 of the SA1110 processor, and may not function as expected with other revisions. Application developers should obtain the manufacturer’s current errata sheets and specification updates. The B4 stepping still has a number of problems, but the device driver can work around these. However there is a penalty in terms of extra code, extra cpu cycles, and increased dispatch latency because extra processing is needed at DSR level. Interrupt latency should not be affected.

There is one specific problem inherent in the UDC design of which application developers should be aware: the hardware cannot fully implement the USB standard for bulk transfers. A bulk transfer typically consists of some number of full-size 64-byte packets and is terminated by a packet less than the full size. If the amount of data transferred is an exact multiple of 64 bytes then this requires a terminating packet of 0 bytes of data (plus header and checksum). The SA11x0 USB hardware does not allow a 0-byte packet to be transmitted, so the device driver is forced to substitute a 1-byte packet and the host receives more data than expected. Protocol support is needed so that the appropriate host-side device driver can allow buffer space for the extra byte, detect when it gets sent, and discard it. Consequently certain standard USB class protocols cannot be implemented using the SA11x0, and therefore custom host-side device drivers will generally have to be provided, rather than re-using existing ones that understand the standard protocol.

Endpoint Data Structures

The SA11x0 USB device driver can provide up to three data structures corresponding to the three endpoints: a usbs_control_endpoint structure usbs_sallix0_ep0; a usbs_rx_endpoint usbs_sallix0_ep1; and a usbs_tx_endpoint usbs_sallix0_ep2. The header file cyg/io/usb/usbs_sallix0.h provides declarations for these.

Not all applications will require support for all the endpoints. For example, if the intended use of the UDC only involves peripheral to host transfers then usbs_sallix0_ep1 is redundant. The device driver provides configuration options to control the presence of each endpoint:

1. Endpoint 0 is controlled by CYGFUN_DEVS_USB_SA11X0_EP0. This defaults to enabled if there are any higher-level packages that require USB hardware or if the global preference CYGGLO_IO_USB_SLAVE_APPLICATION is enabled, otherwise it is disabled. Usually this has the desired
Effect. It may be necessary to override this in special circumstances, for example if the target board uses an external USB chip in preference to the UDC and it is that external chip’s device driver that should be used rather than the on-chip UDC. It is not possible to disable endpoint 0 and at the same time enable one or both of the other endpoints, since a USB device is only usable if it can process the standard control messages.

2. Endpoint 1 is controlled by CYGPKG_DEVS_USB_SA11X0_EP1. By default it is enabled whenever endpoint 0 is enabled, but it can be disabled manually when not required.

3. Similarly endpoint 2 is controlled by CYGPKG_DEVS_USB_SA11X0_EP2. This is also enabled by default whenever endpoint 0 is enabled, but it can be disabled manually.

The SA11X0 USB device driver implements the interface specified by the common eCos USB Slave Support package. The documentation for that package should be consulted for further details. There is only one major deviation: when there is a peripheral to host transfer on endpoint 2 which is an exact multiple of the bulk transfer packet size (usually 64 bytes) the device driver has to pad the transfer with one extra byte. This is because of a hardware limitation: the UDC is incapable of transmitting 0-byte packets as required by the USB specification. Higher-level code, including the host-side device driver, needs to be aware of this and adapt accordingly.

The device driver assumes a bulk packet size of 64 bytes, so this value should be used in the endpoint descriptors in the enumeration data provided by application code. There is experimental code for running with DMA disabled, in which case the packet size will be 16 bytes rather than 64.

Devtab Entries

In addition to the endpoint data structures the SA11X0 USB device driver can also provide devtab entries for each endpoint. This allows higher-level code to use traditional I/O operations such as open/read/write rather than the USB-specific non-blocking functions like usbs_start_rx_buffer. These devtab entries are optional since they are not always required. The relevant configuration options are CYGVAR_DEVS_USB_SA11X0_EP0_DEVTAB_ENTRY, CYGVAR_DEVS_USB_SA11X0_EP1_DEVTAB_ENTRY and CYGVAR_DEVS_USB_SA11X0_EP2_DEVTAB_ENTRY. By default these devtab entries are provided if the global preference CYGGLO_USB_SLAVE_PROVIDE_DEVTAB_ENTRIES is enabled, which is usually the case. Obviously a devtab entry for a given endpoint will only be provided if the underlying endpoint is enabled. For example, there will not be a devtab entry for endpoint 1 if CYGPKG_DEVS_USB_SA11X0_EP1 is disabled.

The names for the three devtab entries are determined by using a configurable base name and appending 0c, 1r or 2w. The base name is determined by the configuration option CYGDAT_DEVS_USB_SA11X0_DEVTAB_BASENAME and has a default value of /dev/usbs, so the devtab entry for endpoint 1 would default to /dev/usbs1r. If the target hardware involves multiple USB devices then application developers may have to change the base name to prevent a name clash.

DMA Engines

The SA11X0 UDC provides only limited fifos for bulk transfers on endpoints 1 and 2; smaller than the normal 64-byte bulk packet size. Therefore a typical transfer requires the use of DMA engines. The SA11x0 provides six DMA engines that can be used for this, and the endpoints require one each (assuming both endpoints are enabled). At the time of writing there is no arbitration mechanism to control access to the DMA engines. By default the device driver will use DMA engine 4 for endpoint 1 and DMA engine 5 for endpoint 2, and it assumes that no other code uses these particular engines.

The exact DMA engines that will be used are determined by the configuration options CYGNUM_DEVS_USB_SA11X0_EP1_DMA_CHANNEL and CYGNUM_DEVS_USB_SA11X0_EP2_DMA_CHANNEL.
These options have the booldata flavor, allowing the use of DMA to be disabled completely in addition to controlling which DMA engines are used. If DMA is disabled then the device driver will attempt to work purely using the fifos, and the packet size will be limited to only 16 bytes. This limit should be reflected in the appropriate endpoint descriptors in the enumeration data. The code for driving the endpoints without DMA should be considered experimental. At best it will be suitable only for applications where the amount of data transferred is relatively small, because four times as many interrupts will be raised and performance will suffer accordingly.
XLIII. NEC uPD985xx USB Device Driver
NEC uPD985xx USB Device Driver

Name
NEC uPD985xx USB Support — Device driver for the on-chip NEC uPD985xx USB device

NEC uPD985xx USB Hardware
The NEC uPD985xx family of processors is supplied with an on-chip USB slave device, the UDC (USB Device Controller). This supports seven endpoints. Endpoint 0 can only be used for control messages. Endpoints 1 and 2 are for isochronous transmits and receives respectively. Endpoints 3 and 4 support bulk transmits and receives. Endpoints 5 and 6 normally support interrupt transmits and receives, but endpoint 5 can also be configured to support bulk transmits. At this time only the control endpoint 0, the bulk endpoints 3 and 4, and the interrupt endpoint 5 are supported.

Endpoint Data Structures
The uPD985xx USB device driver can provide up to four data structures corresponding to the four supported endpoints: a usbs_control_endpoint structure `usbs_upd985xx_ep0`; usbs_tx_endpoint structures `usbs_upd985xx_ep3` and `usbs_upd985xx_ep5`; and a usbs_rx_endpoint `usbs_upd985xx_ep4`. The header file `cyg/io/usb/usbs_nec_upd985xx.h` provides declarations for these.

Not all applications will require support for all the endpoints. For example, if the intended use of the UDC only involves peripheral to host transfers then `usbs_upd985xx_ep4` is redundant. The device driver provides configuration options to control the presence of each endpoint:

1. Endpoint 0 is controlled by `CYGFUN_DEVS_USB_UPD985XX_EP0`. This defaults to enabled if there are any higher-level packages that require USB hardware or if the global preference `CYGGLO_IO_USB_SLAVE_APPLICATION` is enabled, otherwise it is disabled. Usually this has the desired effect. It may be necessary to override this in special circumstances, for example if the target board uses an external USB chip in preference to the UDC and it is that external chip’s device driver that should be used rather than the on-chip UDC. It is not possible to disable endpoint 0 and at the same time enable one or both of the other endpoints, since a USB device is only usable if it can process the standard control messages.

2. Endpoint 3 is controlled by `CYGPKG_DEVS_USB_UPD985XX_EP3`. By default this endpoint is disabled: according to NEC erratum U3 there may be problems when attempting bulk transfers of 192 bytes or greater. As an alternative the device driver provides support for endpoint 5, configured to allow bulk transfers. Endpoint 3 can be enabled if the application only requires bulk transfers of less than 192 bytes, or if this erratum is not applicable to the system being developed for other reasons.

3. Endpoint 4 is controlled by `CYGPKG_DEVS_USB_UPD985XX_EP4`. This is enabled by default whenever endpoint 0 is enabled, but it can be disabled manually.

4. Endpoint 5 is controlled by `CYGPKG_DEVS_USB_UPD985XX_EP5`. This is enabled by default whenever endpoint 0 is enabled, but it can be disabled manually. There is also a configuration option `CYGIMP_DEVS_USB_UPD985XX_EP5_BULK`, enabled by default. This option allows the endpoint to be used for bulk transfers rather than interrupt transfers.

The uPD985xx USB device driver implements the interface specified by the common eCos USB Slave Support package. The documentation for that package should be consulted for further details.
The device driver assumes a bulk packet size of 64 bytes, so this value should be used in the endpoint descriptors in the enumeration data provided by application code. The device driver also assumes a control packet size of eight bytes, and again this should be reflected in the enumeration data. If endpoint 5 is configured for interrupt rather than bulk transfers then the maximum packet size is limited to 64 bytes by the USB standard.

Devtab Entries

In addition to the endpoint data structures the uPD985xx USB device driver can also provide devtab entries for each endpoint. This allows higher-level code to use traditional I/O operations such as `open/read/write` rather than the USB-specific non-blocking functions like `usbs_start_rx_buffer`. These devtab entries are optional since they are not always required. The relevant configuration options are `CYGVAR_DEVS_USB_UPD985XX_EP0_DEVTAB_ENTRY`, `CYGVAR_DEVS_USB_UPD985XX_EP3_DEVTAB_ENTRY`, `CYGVAR_DEVS_USB_UPD985XX_EP4_DEVTAB_ENTRY`, and `CYGVAR_DEVS_USB_UPD985XX_EP5_DEVTAB_ENTRY`. By default these devtab entries are provided if the global preference `CYGGLO_USB_SLAVE_PROVIDE_DEVTAB_ENTRIES` is enabled, which is usually the case. Obviously a devtab entry for a given endpoint will only be provided if the underlying endpoint is enabled. For example, there will not be a devtab entry for endpoint 4 if `CYGPKG_DEVS_USB_UPD985XX_EP4` is disabled.

The names for the devtab entries are determined by using a configurable base name and appending `0c`, `3w`, `4r` or `5w`. The base name is determined by the configuration option `CYGDAT_DEVS_USB_UPD985XX_DEVTAB_BASENAME` and has a default value of `/dev/usbs`, so the devtab entry for endpoint 4 would default to `/dev/usbs4r`. If the target hardware involves multiple USB devices then application developers may have to change the base name to prevent a name clash with other USB device drivers.

Restrictions

The current device driver imposes a restriction on certain bulk receives on endpoint 4. If the protocol being used involves variable-length transfers, in other words if the host is allowed to send less data than a maximum-sized transfer, then the buffer passed to the device driver for receives must be aligned to a 16-byte cacheline boundary and it must be a multiple of this 16-byte cacheline size. This restriction does not apply if the protocol only involves fixed-size transfers.

Optional Hardware Workarounds

The NEC errata list a number of other problems that affect the USB device driver. The device driver contains workarounds for these, which are enabled by default but can be disabled if the application developer knows that the errata are not relevant to the system being developed.

Erratum S1 lists a possible problem if the device driver attempts multiple writes to the USB hardware. This is circumvented by a dummy read operation after every write. If the workaround is not required then the configuration option `CYGIMP_DEVS_USB_UPD985XX_IBUS_WRITE_LIMIT` can be disabled.

Errata U3 and U4 describe various problems related to concurrent transmissions on different endpoints. By default the device driver works around this by serializing all transmit operations. For example if the device driver needs to send a response to a control message on endpoint 0 while there is an ongoing bulk transfer on endpoint 5, the response is delayed until the bulk transfer has completed. Under typical operating conditions this does not cause any problems: endpoint 0 traffic usually happens only during initialization, when the target is connected to the host, while endpoint 5 traffic only happens after initialization. However if transmit serial-
ization is inappropriate for the system being developed then it can be disabled using the configuration option CYGIMP_DEVS_USB_UPD985XX_SERIALIZE_TRANSMITS.

Platform Dependencies

On some platforms it is necessary for the low-level USB device driver to perform some additional operations during start-up. For example it may be necessary to manipulate one of the processor’s GPIO lines before the host can detect a new USB peripheral and attempt to communicate with it. This avoids problems if the target involves a significant amount of work prior to device driver initialization, for example a power-on self-test sequence. If the USB host attempted to contact the target before the USB device driver had been initialized, it would fail to get the expected responses and conclude that the target was not a functional USB peripheral.

Platform-specific initialization code can be provided via a macro UPD985XX_USB_PLATFORM_INIT. Typically this macro would be defined in the platform HAL’s header file cyg/hal/plf_io.h. If the current platform defines such a macro, the USB device driver will invoke it during the endpoint 0 start-up operation.
NEC uPD985xx USB Device Driver
XLIV. Synthetic Target Ethernet Driver
NEC uPD985xx USB Device Driver
Synthetic Target Ethernet Driver

Name

Synthetic Target Ethernet Support — Allow synthetic target applications to perform ethernet I/O

Overview

The synthetic target ethernet package can provide up to four network devices, eth0 to eth3. These can be used directly by the eCos application or, more commonly, by a TCP/IP stack that is linked with the eCos application. Each eCos device can be mapped on to a real Linux network device. For example, if the Linux PC has two ethernet cards and eth1 is not currently being used by Linux itself, then one of the eCos devices can be mapped on to this Linux device. Alternatively, it is possible to map some or all of the eCos devices on to the ethertap support provided by the Linux kernel.

The ethernet package depends on the I/O auxiliary provided by the synthetic target architectural HAL package. During initialization the eCos application will attempt to instantiate the desired devices, by sending a request to the auxiliary. This will load a Tcl script ethernet.tcl that is responsible for handling the instantiation request and subsequent I/O operations, for example transmitting an ethernet packet. However, some of the low-level I/O operations cannot conveniently be done by a Tcl script so ethernet.tcl will actually run a separate program rawether to interact with the Linux network device.

On the target-side there are configuration options to control which network devices should be present. For many applications a single device will be sufficient, but if the final eCos application is something like a network bridge then the package can support multiple devices. On the host-side each eCos network device needs to be mapped on to a Linux one, either a real ethernet device or an ethertap device. This is handled by an entry in the target definition file:

```plaintext
synth_device ethernet {
    eth0 real eth1
    eth1 ethertap tap3 00:01:02:03:FE:05
    ...
}
```

The ethernet package also comes with support for packet logging, and provides various facilities for use by user Tcl scripts.

Installation

Before a synthetic target eCos application can access ethernet devices it is necessary to build and install host-side support. The relevant code resides in the host subdirectory of the synthetic target ethernet package, and building it involves the standard configure, make and make install steps. The build involves a new executable rawether which must be able to access a raw Linux network device. This is achieved by installing it suid root, so the make install step has to be run with superuser privileges.
There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the ethernet support, in a single build tree. This involves using the configure script at the toplevel of the eCos repository. For more information on this, see the README.host file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target ethernet support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The configure options are much the same as for a build from the toplevel, and the README.host file can be consulted for more details. It is essential that the ethernet support be configured with the same --prefix option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the ethernet support.

**Target-side Configuration Options**

The target-side code can be configured to support up to four ethernet devices, eth0 to eth3. By default eth0 is enabled if the configuration includes a TCP/IP stack, otherwise it is disabled. The other three devices are always disabled by default. If any of the devices are enabled then there will also be the usual configuration options related to building this package. Other options related to network devices, for example whether or not to use DHCP, are provided by the generic network device package.

**Real Ethernet**

One obvious way of providing a synthetic target eCos application with ethernet I/O is to use a real ethernet device in the PC: transmitted packets go out on a real network, and packets on the network addressed to the right MAC address are passed on to eCos. This way synthetic target networking behaves just like networking on a real target with ethernet hardware. For example, if there is a DHCP server anywhere on the network then eCos will be able to contact it during networking startup and get hold of IP address information.

Configuring the ethernet support to use a real ethernet device requires a simple entry in the target definition file:

```c
synth_device ethernet {  
   <eCos device> real <linux device>  
   ...  
}
```

For example, to map the eCos network device eth0 to the Linux device eth1:

```c
synth_device ethernet {  
   eth0 real eth1  
   ...  
}
```
It is not possible for an ethernet device to be shared by both the eCos TCP/IP stack and the Linux one: there would be no simple way to work out which stack incoming packets are intended for. In theory it might be possible to do some demultiplexing using distinct IP addresses, but it would be impossible to support some functionality such as DHCP. Therefore the rawether program will refuse to access any ethernet device already in use. On a typical Linux system eth0 will be used for Linux networking, and the PC will have to be equipped with additional ethernet devices for use by eCos.

The rawether program will access the hardware via the appropriate Linux device driver, so it is important that the system is set up such that the relevant module will be automatically loaded or is already loaded. The details of this will depend on the installed distribution and version, but typically it will involve an entry in /etc/modules.conf.

**Ethertap**

The Linux kernel’s ethertap facility provides a virtual network interface. A Linux application, for example the rawether program, can open a special character device /dev/net/tun, perform various ioctl calls, and then write and read ethernet packets. When the device is opened the Linux kernel automatically creates a new network interface, for example tap0. The Linux TCP/IP stack can be made to use this network interface like any other interface, receiving and transmitting ethernet packets. The net effect is a virtual network connecting just the Linux and eCos TCP/IP stacks, with no other nodes attached. By default all traffic remains inside this virtual network and is never forwarded to a real network.

Support for the ethertap facility may or may not be provided automatically, depending on your Linux distribution and version. If your system does not have a device /dev/net/tun or a module tun.o then the appropriate kernel documentation should be consulted, for example/usr/src/linux-2.4/Documentation/networking/tuntap.txt. If you are using an old Linux kernel then the ethertap functionality may be missing completely. When the rawether program is configured and built, the configure script will check for a file /usr/include/linux/if_tun.h. If that file is missing then rawether will be built without ethertap functionality, and only real ethernet interfaces will be supported.

The target definition file is used to map eCos network devices on to ethertap devices. The simplest usage is:

```
synth_device ethernet {
    eth0 ethertap
    ...
}
```

The Linux kernel will automatically allocate the next available tap network interface. Usually this will be tap0 but if other software is using the ethertap facility, for example to implement a VPN, then a different number may be allocated. Usually it will be better to specify the particular tap device that should be used for each eCos device, for example:

```
synth_device ethernet {
    eth0 ethertap tap3
    eth1 ethertap tap4
    ...
}
```

The user now knows exactly which eCos device is mapped onto which Linux device, avoiding much potential confusion. Because the virtual devices are emulated ethernet devices, they require MAC addresses. There is no physical hardware to provide these addresses, so normally MAC addresses will be invented. That means that each time the eCos application is run it will have different MAC addresses, which makes it more difficult to compare the results of different runs. To get more deterministic behaviour it is possible to specify the MAC addresses in the target definition file:
During the initialization phase the eCos application will instantiate the various network devices. This will cause the I/O auxiliary to load the ethernet.tcl script and spawn rawether processes, which in turn will open /dev/net/tun and perform the appropriate ioctl calls. On the Linux side there will now be new network interfaces such as tap3, and these can be configured like any other network interface using commands such as ifconfig. In addition, if the Linux system is set up with hotplug support then it may be possible to arrange for the network interface to become active automatically. On a Red Hat Linux system this would require files such as /etc/sysconfig/network-scripts/ifcfg-tap3, containing data like:

```
DEVICE="tap3"
BOOTPROTO="none"
BROADCAST=10.2.2.255
IPADDR="10.2.2.1"
NETMASK="255.255.255.0"
NETWORK=10.2.2.0
ONBOOT="no"
```

This gives the Linux interface the address 10.2.2.1 on the network 10.2.2.0. The eCos network device should be configured with a compatible address. One way of doing this would be to enable CYGHWR_NET_DRIVER_ETH0_ADDRS, set CYGHWR_NET_DRIVER_ETH0_ADDR_IP to 10.2.2.2, and similarly update the NETMASK, BROADCAST, GATEWAY and SERVER configuration options.

It should be noted that the ethertap facility provides a virtual network, and any packets transmitted by the eCos application will not appear on a real network. Therefore usually there will no accessible DHCP server, and eCos cannot use DHCP or BOOTP to obtain IP address information. Instead the eCos configuration should use manual or static addresses.

When rawether exits, the tap interface is removed by the kernel. By adding the parameter persistent rawether will set the persistent flag on the tap device.

```
synth_device ethernet {
   eth0 ethertap tap3 00:01:02:03:FE:05
   eth1 ethertap tap4 00:01:02:03:FE:06 persistent
   ...
}
```

With this flag set the kernel will not remove the interface when rawether exits. This means applications such as dhcpcd, radvd, and tcpdump will continue to run on the interface between invocations of synthetic targets. As a result the target can dynamically obtain its IP addresses from these daemons. Note it is a good idea to specify a MAC address otherwise a different random MAC address will be used each time and the dhcpcd daemon will not be able to reissue the same IP address.

Host daemons like dhcpcd, ntpd, radvd etc are started at boot time. Since the tap device does not exists at this point in time it is not possible for these daemons to bind to the tap device. A simple solution is to use the program install/bin/mktap. This takes one parameter, the name of the tap device it should create. eg. tap3.

An alternative approach would be to set up the Linux box as a network bridge, using commands like brctl to connect the virtual network interface tap3 to a physical network interface such as eth0. Any packets sent by the eCos application will get forwarded automatically to the real network, and some packets on the real network will get forwarded over the virtual network to the eCos application. Note that the eCos application might also get some packets that were not intended for it, but usually those will just be discarded by the eCos TCP/IP stack. The exact details of setting up a network bridge are left as an exercise to the reader.
Packet Logging

The ethernet support comes with support for logging the various packets that are transferred, including a simple protocol analyser. This generates simple text output using the filter mechanisms provided by the I/O auxiliary, so it is possible to control the appearance and visibility of different types of output. For example the user might want to see IPv4 headers and all ICMPv4 and ARP operations, but not TCP headers or any of the packet data.

The protocol analyser is not intended to be a fully functional analyser with knowledge of many different TCP/IP protocols, advanced search facilities, graphical traffic displays, and so on. Functionality like that is already provided by other tools such as ethereal and tcpdump. Achieving similar levels of functionality would require a lot of work, for very little gain. It is still useful to have some protocol analysis functionality available because the output will be interleaved with other output, for example printf calls from the application. That may make it easier to understand the sequence of events.

One problem with logging ethernet traffic is that it can involve very large amounts of data. If the application is expected to run for a long time or is very I/O intensive then it is easy to end up with many megabytes. When running in graphical mode all the logging data will be held in memory, even data that is not currently visible. At some point the system will begin to run low on memory and performance will suffer. To avoid problems, the ethernet script maintains a flag that controls whether or not packet logging is active. The default is to run with logging disabled, but this can be changed in the target definition file:

```plaintext
synth_device ethernet {
    ...
    logging 1
}
```

The ethernet script will add a toolbar button that allows this flag to be changed at run-time, allowing the user to capture traffic for certain periods of time while the application continues running.

The target definition file can contain the following entries for the various packet logging filters:

```plaintext
synth_device ethernet {
    ...
    filter ether -hide 0 -background LightBlue -foreground "#000080"
    filter arp -hide 0 -background LightBlue -foreground "#000050"
    filter ipv4 -hide 0 -background LightBlue -foreground "#000040"
    filter ipv6 -hide 1 -background LightBlue -foreground "#000040"
    filter icmpv4 -hide 0 -background LightBlue -foreground "#000070"
    filter icmpv6 -hide 1 -background LightBlue -foreground "#000070"
    filter udp -hide 0 -background LightBlue -foreground "#000030"
    filter tcp -hide 0 -background LightBlue -foreground "#000020"
    filter hexdata -hide 1 -background LightBlue -foreground "#000080"
    filter asciidata -hide 1 -background LightBlue -foreground "#000080"
}
```

All output will show the eCos network device, for example eth0, and the direction relative to the eCos application. Some of the filters will show packet headers, for example ether gives details of the ethernet packet header and tcp gives information about TCP headers such as whether or not the SYN flag is set. The TCP and UDP filters will also show source and destination addresses, using numerical addresses and if possible host names. However, host names will only be shown if the host appears in /etc/hosts: doing full DNS lookups while the data is being captured would add significantly to complexity and overhead. The hexdata and asciidata filters show the remainder of the packets after the ethernet, IP and TCP or UDP headers have been stripped.

Some of the filters will provide raw dumps of some of the packet data. Showing up to 1500 bytes of data for each packet would be expensive, and often the most interesting information is near the start of the packet. Therefore it is possible to set a limit on the number of bytes that will be shown using the target definition file. The default limit is 64 bytes.
User Interface Additions

When running in graphical mode the ethernet script extends the user interface in two ways: a button is added to the toolbar so that users can enable or disable packet logging; and an entry is added to the Help menu for the ethernet-specific documentation.

Command Line Arguments

The synthetic target ethernet support does not use any command line arguments. All configuration is handled through the target definition file.

Hooks

The ethernet support defines two hooks that can be used by other scripts, especially user scripts: ethernet_tx and ethernet_rx. The tx hook is called whenever eCos tries to transmit a packet. The rx hook is called whenever an incoming packet is passed to the eCos application. Note that this may be a little bit after the packet was actually received by the I/O auxiliary since it can buffer some packets. Both hooks are called with two arguments, the name of the network device and the packet being transferred. Typical usage might look like:

```tcl
proc my_tx_hook {arg_list} {
    set dev [lindex $arg_list 0]
    incr ::my_ethernet_tx_packets($dev)
    incr ::my_ethernet_tx_bytes($dev) [string length [lindex $arg_list 1]]
}
proc my_rx_hook {arg_list} {
    set dev [lindex $arg_list 0]
    incr ::my_ethernet_rx_packets($dev)
    incr ::my_ethernet_rx_bytes($dev) [string length [lindex $arg_list 1]]
}
synth::hook_add "ethernet_tx" my_tx_hook
synth::hook_add "ethernet_rx" my_rx_hook
```

The global arrays `my_ethernet_tx_packets` etc. will now be updated whenever there is ethernet traffic. Other code, probably running at regular intervals by use of the Tcl `after` procedure, can then use this information to update a graphical monitor of some sort.

Additional Tcl Procedures

The ethernet support provides one additional Tcl procedure that can be used by other scripts:

```tcl```
eternet::devices_get_list```

This procedure returns a list of the ethernet devices that have been instantiated, for example `{eth0 eth1}`.
XLV. Freescale ColdFire Ethernet Driver
Synthetic Target Ethernet Driver
**Freescale ColdFire Ethernet Driver**

**Name**

CYGPKG_DEVS_ETH_MCFxxxx — eCos Support for Freescale ColdFire On-chip Ethernet Devices

**Description**

Some members of the Freescale ColdFire family of processors come with an on-chip ethernet device. This package provides an eCos driver for that device. The driver supports both polled mode for use by RedBoot and interrupt-driven mode for use by a full TCP/IP stack.

The original version of the driver was written specifically for the MCF5272 processor. It has since been made to work on other members of the ColdFire family.

**Configuration Options**

This ethernet package should be loaded automatically when selecting a target containing a ColdFire processor with on-chip ethernet, and it should never be necessary to load it explicitly. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but the net, redboot and lwip_eth templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver.

All eCos network devices need a unique name. By default the on-chip ethernet device is assigned the name eth0 but can be changed through the configuration option CYGDAT_DEVS_ETH_MCFxxxx_NAME. This is useful if for example the target hardware includes a number of additional off-chip ethernet devices.

The hardware requires that incoming ethernet frames are received into one of a small number of buffers, arranged in a ring. Once a frame has been received and its size is known the driver will pass it up to higher-level code for further processing. The number of these buffers is configurable via the option CYGNUM_DEVS_ETH_MCFxxxx_RXBUFFERS. Each receive buffer requires 1528 bytes of memory. A smaller number of buffers increases the probability that incoming ethernet frames have to be discarded. TCP/IP stacks are designed to cope with the occasional lost packet, but if too many frames are discarded then this will greatly affect performance. A key issue here is that passing the incoming frames up to higher-level code typically happens at thread level and hence the system behaviour is defined in large part by the priority of various threads running in the TCP/IP stack. If application code has high-priority threads that take up much of the available cpu time and the TCP/IP stack gets little chance to run then there will be little opportunity to pass received frames up the stack. Balancing out the various thread priorities and the number of receive buffers is the responsibility of the application developer.

By default the ethernet driver will raise interrupts using a low priority level. The exact value will depend on the processor being used, for example the MCF5282 interrupt controllers impose specific constraints on interrupt priorities. The driver does very little at interrupt level, instead the real work is done via threads inside the TCP/IP stack. Hence the interrupt priority has little or no effect on the system’s behaviour. If the default priorities are inappropriate for some reason then they can be changed through the configuration options CYGNUM_DEVS_ETH_MCFxxxx_ISR_RX_PRIORITY and CYGNUM_DEVS_ETH_MCFxxxx_ISR_TX_PRIORITY.

There is an option related to the default network MAC address, CYGDAT_DEVS_ETH_MCFxxxx_PLATFORM_MAC. This is discussed in more detail below.
Optionally the ethernet driver can maintain statistics about the number of incoming and transmitted ethernet frames, receive overruns, collisions, and other conditions. Maintaining and providing these statistics involves some overhead, and is controlled by the configuration option `CYGFUN_DEVS_ETH_MCFxxxx_STATISTICS`. Typically these statistics are only accessed through SNMP, so by default statistics gathering is enabled if the configuration includes `CYGPKG_SNMPAGENT` and disabled otherwise.

**MAC Address**

The ColdFire processors do not have a built-in unique network MAC address since that would require slightly different manufacturing for each chip. All ethernet devices should have a unique address so this has to come from elsewhere. There are a number of possibilities:

1. The platform HAL can provide the address. For example the target board may have a small serial EPROM or similar which is initialized during board manufacture. The platform HAL can read the serial EPROM during system startup and provide the information to the ethernet driver. If this is the case then the platform HAL should implement the CDL interface `CYGINT_DEVS_ETH_MCFxxxx_PLATFORM_MAC` and provide a macro `HAL_MCFxxxx_ETH_GET_MAC_ADDRESS` in the exported header `cyg/hal/plf_arch.h`.

2. There is a configuration option `CYGDAT_DEVS_ETH_MCFxxxx_PLATFORM_MAC` which specifies the default MAC address. Manipulating this option is fine if the configuration will only be used on a single board. However if multiple boards run applications with the same configuration then they would all have the same MAC address, and the resulting behaviour is undefined.

3. If the target hardware boots via RedBoot and uses a block of flash to hold configuration variables then one of these variables will be the MAC address. It can be manipulated at the RedBoot prompt using the `fconfig` command, thus giving each board a unique address. An eCos application containing the ethernet driver will automatically pick up this address.

When designing a new target board it is recommended that the board comes with a unique network address supported by the platform HAL, rather than relying on users to change the address. The latter approach can be error-prone and will lead to failures that are difficult to track down.

**Platform-specific PHY**

The on-chip ethernet hardware relies on an external media independent interface (MII), also known as a PHY chip. This separate chip handles the low-level details of ethernet communication, for example negotiating a link speed with the hub. In most scenarios the PHY chip simply does the right thing and needs no support from the ethernet driver. If there are special requirements, for example if the board has to be hardwired to communicate at 10Mbps rather than autonegotiate the link speed, then usually this is handled by fixed logic levels on some of the PHY pins or by using jumpers.

The eCos ethernet driver assumes that the PHY is already fully operational and does not interact with it in any way. If the target hardware does require software initialization of the PHY chip then usually this will be done in the platform HAL, because the choice of PHY chip is a characteristic of the platform.
XLVI. Freescale MCFxxxx Serial Driver
MCFxxxx Serial Driver

Name

CYGPKG_DEVS_SERIAL_MCFxxxx — eCos Support for the MCFxxxx On-chip Serial Devices

Description

All members of the Freescale MCFxxxx ColdFire family of processors contain a number of on-chip UARTs for serial communication. They all use very similar hardware. There are some variations such as different fifo sizes, and some processors contain extra functionality such as autobaud detection, but a single eCos device driver can cope with most of these differences. The CYGPKG_DEVS_SERIAL_MCFxxxx package provides this driver. It will use definitions provided by the variant HAL CYGPKG_HAL_M68K_MCFxxxx, the processor HAL and the platform HAL.

The driver provides partial support for hardware flow control and for serial line status. Only CTS/RTS hardware flow control is supported since the UART does not provide DTR/DSR lines. Similarly only line breaks, and certain communication errors are supported for line status since the UART does not provide other lines such as DCD or RI. On some platforms it should be possible to emulate these lines using GPIO pins, but currently there is no support for this.

Once application code accesses a UART through the serial driver, for example by opening a device /dev/ser0, the driver assumes that it has sole access to the hardware. This means that the UART should not be used for any other purpose, for example HAL diagnostics or gdb debug traffic. Instead such traffic has to go via another communication channel such as ethernet.

Configuration Options

The MCFxxxx serial driver should be loaded automatically when selecting a platform containing a suitable processor, and it should never be necessary to load it explicitly. The driver as a whole is inactive unless the generic serial support, CYGPKG_IO_SERIAL_DEVICES, is enabled. Exactly which UART or UARTs are accessible on a given platform is determined by the platform because even if the processor contains a UART the platform may not provide a connector. Support for a given UART, say uart0, is controlled by a configuration option CYGPKG_DEVS_SERIAL_MCFxxxx_SERIAL0. The device driver configuration option in turn depends on a HAL configuration option CYGHWR_HAL_M68K_MCFxxxx_UART0 to indicate that the UART is actually present and connected on the target hardware. If a given UART is of no interest to an application developer then it is possible to save some memory by disabling this option.

For every enabled UART there are a further four configuration options:

CYGDAT_DEVS_SERIAL_MCFxxxx_SERIAL0_NAME

Each serial device should have a unique name so that application code can open it. The default device names are /dev/ser0, /dev/ser1, and so on. It is only necessary to change these if the platform contains additional off-chip UARTs with clashing names.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY

By default the driver arranges for the UARTs to interrupt at a low interrupt priority. Usually there will be no need to change this because the driver does not actually do very much processing at ISR level, and anyway UARTs are not especially fast devices so do not require immediate attention. On some Coldfires with MCF5282-compatible interrupt controllers care has to be taken that all interrupt priorities are unique.
MCFxxxx Serial Driver

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_BAUD

Each UART will be initialized to a given baud rate. The default baud rate is 38400 because in most scenarios this is fast enough yet does not suffer from excess data corruption. Lower baud rates can be used if the application will operate in an electrically noisy environment, or higher baud rates up to 230400 can be used if 38400 does not provide sufficient throughput.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_BUFSIZE

The serial driver will maintain software buffers for incoming and outgoing data. The former allows data to continue to arrive even if the application is still busy processing the previous transfer, and thus potentially improves throughput. The latter allows the application to transmit data without immediately blocking until the transfer is complete, often eliminating the need for a separate thread. The size of these buffers can be controlled via this configuration option, or alternatively these buffers can be disabled completely to save memory.

There are additional options in the generic serial I/O package CYGPKG_IO_SERIAL which will affect this driver. For example CYGPKG_IO_SERIAL_FLOW_CONTROL and its sub-options determine what flow control mechanism (if any) should be used.

This package also defines some configuration options related to testing. Usually these options are of no interest to application developers and can be ignored.

Porting

The generic driver needs some information from other packages about the exact hardware, for example how many UARTs are available and where in memory they can be accessed.

1. Another package, usually the processor HAL, should provide one or more options CYGHWR_HAL_M68K_MCFxxxx_UART0, CYGHWR_HAL_M68K_MCFxxxx_UART1 or CYGHWR_HAL_M68K_MCFxxxx_UART2. These may be calculated or user-configurable depending on the processor.

2. The device driver will also look for symbol definitions CYGHWR_HAL_M68K_MCFxxxx_UART0_RTS and CYGHWR_HAL_M68K_MCFxxxx_UART0_CTS, and the equivalents for the other UARTs, to determine whether or not these handshake lines are connected. These may be configuration options or they may be statically defined in a HAL I/O header file. The platform HAL should also implement the generic serial package’s interface CYGINT_IO_SERIAL_FLOW_CONTROL_HW if appropriate.

3. If RTS is connected then the driver will also look for a symbol CYGHWR_HAL_M68K_MCFxxxx_UART0_RS485_RTS. This enables partial support for RS485 communication in that the device driver will arrange for the RTS line to be asserted during a transmit. The driver has no support for more advanced RS485 functionality such as multidrop.

In addition the driver assumes the standard MCFxxxx HAL macros are defined for the UART base addresses and the registers. The driver primarily targets MCF5282-compatible UARTs but there is also some support for functionality available on other members of the Coldfire range, for example the MCF5272’s fractional baud rate support.
XLVII. Synthetic Target Watchdog Device
MCFxxxx Serial Driver
Synthetic Target Watchdog Device

Name

Synthetic Target Watchdog Device — Emulate watchdog hardware in the synthetic target

Overview

Some target hardware comes equipped with a watchdog timer. Application code can start this timer and after a certain period of time, typically a second, the watchdog will trigger. Usually this causes the hardware to reboot. The application can prevent this by regularly resetting the watchdog. An automatic reboot can be very useful when deploying hardware in the field: a hardware glitch could cause the unit to hang; or the software could receive an unexpected sequence of inputs, never seen in the laboratory, causing the system to lock up. Often the hardware is still functional, and a reboot sorts out the problem with only a brief interruption in service.

The synthetic target watchdog package emulates watchdog hardware. During system initialization watchdog device will be instantiated, and the watchdog.tcl script will be loaded by the I/O auxiliary. When the eCos application starts the watchdog device, the watchdog.tcl script will start checking the state of the eCos application at one second intervals. A watchdog reset call simply involves a message to the I/O auxiliary. If the watchdog.tcl script detects that a second has elapsed without a reset then it will send a SIGPWR signal to the eCos application, causing the latter to terminate. If gdb is being used to run the application, the user will get a chance to investigate what is happening. This behaviour is different from real hardware in that there is no automatic reboot, but the synthetic target is used only for development purposes, not deployment in the field: if a reboot is desired then this can be achieved very easily by using gdb commands to run another instance of the application.

Installation

Before a synthetic target eCos application can use a watchdog device it is necessary to build and install host-side support. The relevant code resides in the host subdirectory of the synthetic target watchdog package, and building it involves the standard configure, make and make install steps. The implementation of the watchdog support does not require any executables, just a Tcl script watchdog.tcl and some support files, so the make step is a no-op.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the watchdog support, in a single build tree. This involves using the configure script at the toplevel of the eCos repository. For more information on this, see the README.host file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target watchdog support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The configure options are much the same as for a build from the toplevel, and the README.host file can be consulted for more details. It is essential that the watchdog support be configured with the same --prefix option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the watchdog support.
**Target-side Configuration**

The watchdog device depends on the generic watchdog support, `CYGPKG_IO_WATCHDOG`: if the generic support is absent then the watchdog device will be inactive. Some templates include this generic package by default, but not all. If the configuration does not include the generic package then it can be added using the eCos configuration tools, for example:

```
$ ecosconfig add CYGPKG_IO_WATCHDOG
```

By default the configuration will use the hardware-specific support, i.e. this package. However the generic watchdog package contains an alternative implementation using the kernel alarm facility, and that implementation can be selected if desired. However usually it will be better to rely on an external watchdog facility as provided by the I/O auxiliary and the `watchdog.tcl` script: if there are serious problems within the application, for example memory corruption, then an internal software-only implementation will not be reliable.

The watchdog resolution is currently fixed to one second: if the device does not receive a reset signal at least once a second then the watchdog will trigger and the eCos application will be terminated with a `SIGPWR` signal. The current implementation does not allow this resolution to be changed.

On some targets the watchdog device does not perform a hard reset. Instead the device works more or less via the interrupt subsystem, allowing application code to install action routines that will be called when the watchdog triggers. The synthetic target watchdog support effectively does perform a hard reset, by sending a `SIGPWR` signal to the eCos application, and there is no support for action routines.

The synthetic target watchdog package provides some configuration options for manipulating the compiler flags used for building the target-side code. That code is fairly simple, so for nearly all applications the default flags will suffice.

It should be noted that the watchdog device is subject to selective linking. Unless some code explicitly refers- ences the device, for example by calling the start and reset functions, the watchdog support will not appear in the final executable. This is desirable because a watchdog device has no effect until started.

**Wallclock versus Elapsed Time**

On real hardware the watchdog device uses wallclock time: if the device does not receive a reset signal within a set period of time then the watchdog will trigger. When developing for the synthetic target this is not always appropriate. There may be other processes running, using up some or most of the cpu time. For example, the application may be written such that it will issue a reset after some calculations which are known to complete within half a second, well within the one-second resolution of the watchdog device. However if other Linux processes are running then the synthetic target application may get timesliced, and half a second of computation may take several seconds of wallclock time.

Another problem with using wallclock time is that it interferes with debugging: if the application hits a break- point then it is unlikely that the user will manage to restart it in less than a second, and the watchdog will not get reset in time.

To avoid these problems the synthetic target watchdog normally uses consumed cpu time rather than wallclock time. If the application is timesliced or if it is halted inside gdb then it does not consume any cpu time. The application actually has to spend a whole second’s worth of cpu cycles without issuing a reset before the watchdog triggers.

However using consumed cpu time is not a perfect solution either. If the application makes blocking system calls then it is not using cpu time. Interaction with the I/O auxiliary involves system calls, but these should take only a short amount of time so their effects can be ignored. If the application makes direct system calls such as `cyg_hal_sys_read` then the system behaviour becomes undefined. In addition by default the idle thread will make blocking `select` system calls, effectively waiting until an interrupt occurs. If an application spends
much of its time idle then the watchdog device may take much longer to trigger than expected. It may be desirable to enable the synthetic target HAL configuration option CYGIMP_HAL_IDLE_THREAD_SPIN, causing the idle thread to spin rather than block, at the cost of wasted cpu cycles.

The default is to use consumed cpu time, but this can be changed in the target definition file:

```plaintext
synth_device watchdog {
    use wallclock_time
...
}
```

**User Interface**

When the synthetic target is run in graphical mode the watchdog device extends the user interface in two ways. The Help menu is extended with an entry for the watchdog-specific documentation. There is also a graphical display of the current state of the watchdog. Initially the watchdog is asleep:

![Initial state of the watchdog](image1)

When application code starts the device the watchdog will begin to keep an eye on things (or occasionally both eyes).

![View with eyes](image2)

If the watchdog triggers the display will change again, and optionally the user can receive an audible alert. The location of the watchdog display within the I/O auxiliary’s window can be controlled via a `watchdog_pack` entry in the target definition file. For example the following can be used to put the watchdog display to the right of the central text window:

```plaintext
synth_device watchdog {
    watchdog_pack -in .main.e -side top
    ...
}
```

The user interface section of the generic synthetic target HAL documentation can be consulted for more information on window packing.

By default the watchdog support will not generate an audible alert when the watchdog triggers, to avoid annoying colleagues. Sound can be enabled in the target definition file, and two suitable files `sound1.au` and `sound2.au` are supplied as standard:

```plaintext
synth_device watchdog {
    sound sound1.au
    ...
}
```
Synthetic Target Watchdog Device

An absolute path can be specified if desired:

```tcl
synth_device watchdog {
    sound /usr/share/emacs/site-lisp/emacspeak/sounds/default-8k/alarm.au

...}
```

Sound facilities are not built into the I/O auxiliary itself, instead an external program is used. The default player is `play`, a front-end to the sox application shipped with some Linux distributions. If another player should be used then this can be specified in the target definition file:

```tcl
synth_device watchdog {
...
    sound_player my_sound_player
```

The specified program will be run in the background with a single argument, the sound file.

Command Line Arguments

The watchdog support does not use any command line arguments. All configuration is handled through the target definition file.

Hooks

The watchdog support does not provide any hooks for use by other scripts. There is rarely any need for customizing the system’s behaviour when a watchdog triggers because those should be rare events, even during application development.

Additional Tcl Procedures

The watchdog support does not provide any additional Tcl procedures or variables for use by other scripts.
XLVIII. Dallas DS1307 Wallclock Device Driver
Synthetic Target Watchdog Device
Dallas DS1307 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 — eCos Support for the Dallas DS1307 Serial Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 provides a device driver for the wallclock device in the Dallas DS1307 Serial Real-Time Clock chips. This combines a real-time clock and 56 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a date command which interacts with the wallclock device.

Porting

DS1307 platform support can be implemented in one of two ways. The preferred approach involves the generic I2C API, as defined by the package CYGPKG_IO_I2C. The platform HAL can just provide a cyg_i2c_device structure cyg_i2c_wallclock_ds1307 and implement the CDL interface CYGINT_DEVICES_WALLCLOCK_DALLAS_DS1307_I2C. The DS1307 driver will now use I2C rx and tx operations to interact with the chip.

Alternatively the DS1307 driver can use macros or functions provided by another package to access the chip. This is intended primarily for older platforms that predate the CYGPKG_IO_I2C package. The other package should export a header file containing macros DS_GET and DS_PUT that transfer the eight bytes corresponding to the chip’s clock registers. It should also export the name of this header via a #define CYGDAT_DEVS_WALLCLOCK_DS1307_INL in the global configuration header pkgconf/system.h. For full details see the source code.

In addition the DS1307 device driver package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.
XLIX. MMC MultiMedia Card Disk Driver
Dallas DS1307 Wallclock Device Driver
Device Driver for MMC MultiMedia Cards

**Name**

CYGPKG_DEVS_DISK_MMC — eCos Support for MMC MultiMedia Cards

**Description**

This package provides a disk device driver for MultiMediaCards (MMC). A MultiMediaCard provides non-volatile storage in a small footprint (24mm * 32mm * 1.4mm), and weighing less than 2 grams. Typical card sizes are 16MB to 128MB, with an upper limit of 4GB. It should be noted that these sizes are measured in millions of bytes, not $2^{20}$. The MultiMediaCard Association (http://www.mmca.org) defines the standard for these cards.

At the hardware level there are two ways of accessing an MMC card, either using a custom interface or via an SPI bus. A card will detect the interface in use at run-time. The custom interface allows for better performance but requires additional hardware. Currently only SPI mode is supported by this package.

Theoretically an MMC card can be used with any file system. In practice all cards are formatted for PC compatibility, with a partition table in the first block and a single FAT file system on the rest of the card. Currently this package checks the format of the MMC card and will only allow access to a card if it is formatted this way.

An MMC socket allows cards to be removed and inserted at any time. The device driver will detect removal events when the next I/O operation happens and will report them to higher-level code via an error code ENODEV. However there are no guarantees that the higher-level code will be able to recover from this error. The expected usage is that application code will explicitly mount the card before attempting any file I/O, and will umount the card before it is removed. Between these operations the system is likely to keep some disk blocks cached, for performance reasons. If the card is removed before the umount then it may end up with a corrupted file system, and the disk subsystem may be left in an inconsistent state. Regular uses of sync will reduce the risk of file system corruption.

**Configuration Options**

CYGPKG_DEVS_DISK_MMC is a hardware package which should get loaded automatically when you configure for a suitable eCos target platform. In this case suitable means that the hardware has an MMC socket connected to an SPI bus, that an SPI bus driver package exists and is also automatically loaded, and that the platform HAL provides information on how the card is connected to the SPI bus.

The package depends on support from the generic disk package CYGPKG_IO_DISK. That will not be loaded automatically: the presence of an MMC socket on the board does not mean that the application has any need for a file system. Hence by default CYGPKG_DEVS_DISK_MMC will be inactive and will not contribute any code or data to the application’s memory footprint. To activate the driver it will be necessary to add one or more packages to the configuration using ecosconfig add or the graphical configuration tool: the generic disk support CYGPKG_IO_DISK; usually a file system, CYGPKG_FS_FAT; support for the file I/O API CYGPKG_IO_FILEIO; and possibly additional support packages that may be needed by the file system, for example CYGPKG_LINUX_COMPAT. Depending on the template used to create the initial configuration some of these may be loaded already.

The package provides two main configuration options. CYGDAT_DEVS_DISK_MMC_SPI_DISK0_NAME specifies the name of the raw disk device, for example /dev/hd0. Allowing for partition tables that makes /dev/hd0/1 the first argument that should be passed to a mount call. If the hardware has multiple disk devices then each one
Device Driver for MMC MultiMedia Cards

will need a unique name. `CYGIMP_DEVS_DISK_MMC_SPI_POLLED` controls whether the SPI bus will be accessed in interrupt-driven or polled mode. It will default to interrupt-driven if the application is multi-threaded, which is assumed to be the case if the kernel is present. If the kernel is absent, for example in a RedBoot configuration, then the driver will default to polled mode. With some hardware polled mode may significantly increase disk throughput even in a multi-threaded application, but will consume cpu cycles that could be used by other threads.

**Additional Functionality**

The disk driver package exports a variable `cyg_mmc_spi_polled`. This defaults to true or false depending on the configuration option `CYGIMP_DEVS_DISK_MMC_SPI_POLLED`. If the default mode is interrupt-driven then file I/O, including mount operations, are only allowed when the scheduler has started and interrupts have been enabled. Any attempts at file I/O earlier during system initialization, for example inside a C++ static constructor, will lock up. If it is necessary to perform file I/O at this time then the driver can be temporarily switched to polling mode before the I/O operation by setting `cyg_mmc_spi_polled`, and clearing it again after the I/O. Alternatively the default mode can be changed to polling by editing the configuration, and then the `main()` thread can change the mode to interrupt-driven once the scheduler has started.

**Porting to New Hardware**

Assuming that the MMC connector is hooked up to a standard SPI bus and that there is already an eCos SPI bus driver, porting the MMC disk driver package should be straightforward. Some other package, usually the platform HAL, should provide a cyg_spi_device structure `cyg_spi_mmc_dev0`. That structure contains the information needed by this package to interact with the MMC card via the usual SPI interface, for example how to activate the appropriate chip select. The platform HAL should also implement the CDL interface `CYGINT_DEVS_DISK_MMC_SPI_CONNECTORS`.

When defining `cyg_spi_mmc_dev0` special care must be taken with the chip select. The MMC protocol is transaction-oriented. For example a read operation involves an initial command sent to the card, then a reply, then the actual data, and finally a checksum. The card’s chip select must be kept asserted for the entire operation, and there can be no interactions with other devices on the same SPI bus during this time.

Optionally the platform HAL may define a macro `HAL_MMC_SPI_INIT` which will be invoked during a mount operation. This can take any hardware-specific actions that may be necessary, for example manipulating GPIO pins. Usually no such macro is needed because the hardware is set up during platform initialization.

Currently the package does not provide any support for accessing MMC cards using an interface other than SPI. On some targets there may be additional hardware to detect events such as card insertion or removal, but there is no support for exploiting such hardware at present.

Only a single MMC socket is supported. Extending the package to support multiple sockets would be straightforward but it seems unlikely that any hardware would come with multiple MMC sockets. Given the nature of SPI buses there is a problem if the MMC socket is hooked up via an expansion connector rather than being attached to the main board. The platform HAL would not know about the socket so would not implement the CDL interface `CYGINT_DEVS_DISK_MMC_SPI_CONNECTORS`, and the ecos.db target entry would not include `CYGPKG_DEVS_DISK_MMC`. Because this is a hardware package it cannot easily be added by hand. Instead this scenario would require some editing of the existing platform HAL and target entry.
L. Synthetic Target Framebuffer Device
Device Driver for MMC MultiMedia Cards
Synthetic Target Framebuffer Device

Name

Synthetic Target Framebuffer Device — Emulate framebuffer hardware in the synthetic target

Overview

This package CYGPKG_DEVS_FRAMEBUF_SYNTH provides a framebuffer device driver for the eCos synthetic target.

The driver supports up to four framebuffer devices `fb0`, `fb1`, `fb2`, and `fb3`. The width, height, depth, and display format of each framebuffer can be controlled via configuration options. It is also possible to set a viewport for each device and to enable page flipping.

To use the framebuffer support the eCos application must run inside an X session, not from the console, and it must be started with `--io` to enable the I/O auxiliary. The I/O auxiliary will start a separate instance of a host-side utility framebuffer for each target-side framebuffer device. The framebuffer utility can access the eCos.
framebuffer data via a shared memory region and draw it to the screen using X library calls. It needs the X server to run with a TrueColor visual and a display of depth of 24 or 32 bits per pixel.

Installation

The synthetic target framebuffer driver depends on host-side support which must be built and installed. The relevant code resides in the host subdirectory of the synthetic target framebuffer package, and building it involves the standard configure, make and make install steps. This will build and install a utility program framebuf that does the actual drawing of the eCos framebuffer contents to the host-side X display. It will also install a Tcl script and some support files. framebuf is an X11 application so can only be built on Linux systems with the appropriate X11 development package or packages.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the framebuffer support, in a single build tree. This involves using the configure script at the toplevel of the eCos repository. For more information on this, see the README.host file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target framebuffer support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The configure options are much the same as for a build from the toplevel, and the README.host file can be consulted for more details. It is essential that the framebuffer support be configured with the same --prefix option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the framebuffer support.

Configuration

The package is loaded automatically when creating a configuration for the synthetic target. However it is inactive unless the generic framebuffer support CYGPKG_IO_FRAMEBUF is also added to the configuration, for example by ecosconfig add framebuf.

By default the package enables a single framebuffer device fb0 with a corresponding cyg_fb data structure cyg_synth_fb0. The default settings for this device are 320 by 240 pixels, a depth of 8 bits per pixel, a paletted display, no viewport support, and no page flipping. All of these settings can be changed by configuration options inside the CDL component CYGPKG_DEVS_FRAMEBUF_SYNTH_FB0. The supported display formats are: 8 bpp paletted; 8bpp true colour 332; 16bpp true 565; 16bpp true 555; and 32bpp 0888. This allows the synthetic target to match the actual display capabilities of the hardware that is being emulated. If the actual hardware has more than one framebuffer device then this can be emulated by enabling additional components CYGPKG_DEVS_FRAMEBUF_SYNTH_FB1 ..., and setting the appropriate options.

Customization

In addition to the target-side configurability it is possible to customize the host-side behaviour. For example, the default behaviour is for fb0 to be drawn inside the I/O auxiliary’s main window, if it is not too large. fb1, fb2 and fb3 will be drawn inside separate toplevel windows, as will fb0 if that has been configured too large for embedding in the main window. This behaviour can be changed by providing a custom Tcl/Tk procedure that creates the containing frame for the framebuffer device.

Customization involves adding a synth_device framebuf section to the .tdf target definition file, usually default.tdf or ~/.ecos/synth/default.tdf.
The pixel size on the host display may be rather smaller than on the final hardware, causing a serious mismatch between the application’s appearance when using synthetic target emulation and when using real hardware. To reduce this problem the host-side can magnify the target-side framebuffer devices. In the example above each target-side pixel in device `fb2` will be drawn using 2x2 pixels on the host side. Valid magnifications are 1, 2, 3 and 4. With a magnification of 4 an eCos framebuffer device of 320x240 pixels will be drawn in an X window of 1280x960 pixels.

The `create_frame_proc` entry can be used to specify a custom Tcl/Tk procedure that will create the containing Tk frames for the host-side displays. This procedure can be written for a specific configuration, but it is supplied with all the parameters associated with the framebuffer device so can be more generic. An example is supplied in the package’s `misc` subdirectory:

1. Create a configuration for the synthetic target with the default template.
2. Import the `example.ecm` configuration fragment from the `misc` subdirectory. This will add the generic framebuffer support package, enable all four framebuffer devices, and configure each device. Build the resulting configuration.
3. Compile the `example.c` program and link it against the eCos configuration.
4. Incorporate the `example.tdf` fragment into the appropriate target definition file, typically `default.tdf` or `~/.ecos/synth/default.tdf`.
5. Run the example executable. The four framebuffer devices should get instantiated in a separate window in a single column. `FB0` just contains a static display. `FB1` supports two pages, one with vertical stripes and one with horizontal stripes, and the two pages are flipped at regular intervals. `FB2` has a static display similar to `FB0`, but is drawn in a viewport of only 160x120 pixels. However `example.tdf` magnifies this by 2 so it appears the same size as the other devices. The application moves the viewport around the underlying framebuffer device. `FB3` is also a static display, a simple set of vertical stripes. However this framebuffer is paletted and the palette is changed at regular intervals, causing apparent movement.
Synthetic Target Framebuffer Device
LI. AMD AM29xxxxx Flash Device Driver
Synthetic Target Framebuffer Device
Overview

Name
Overview — eCos Support for AMD AM29xxxx Flash Devices and Compatibles

Description
The CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2 AMD AM29xxxxx V2 flash driver package implements support for the AM29xxxx family of flash devices and compatibles. Normally the driver is not accessed directly. Instead application code will use the API provided by the generic flash driver package CYGPKG_IO_FLASH, for example by calling functions like cyg_flash_program.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note: Many eCos targets with AM29xxxxx or compatible flash devices will still use the older driver package CYGPKG_DEVS_FLASH_AMD_AM29XXXXX. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options
The AM29xxxxx flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the AM29xxxxx driver package.

There are a number of configuration options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_REGIONS may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.
Overview
Instantiating an AM29xxxxx Device

Name
Instantiating — including the driver in an eCos target

Synopsis

#include <cyg/io/am29xxxxx_dev.h>

int cyg_am29xxxxx_init_check_devid_XX(struct cyg_flash_dev* device);
int cyg_am29xxxxx_init_cfi_XX(struct cyg_flash_dev* device);
int cyg_am29xxxxx_erase_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr);
int cyg_am29xxxxx_program_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr, const void* data, size_t len);
int cyg_at49xxxx_softlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_at49xxxx_hardlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_at49xxxx_unlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_am29xxxxx_read_devid_XX(struct cyg_flash_dev* device);

Description

The AM29xxxxx family contains some hundreds of different flash devices, all supporting the same basic set of operations but with various common or uncommon extensions. The devices vary in capacity, performance, boot block layout, and width. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The AM29xxxxx driver package cannot know the details of every chip and every platform. Instead it is the responsibility of another package, usually the platform HAL, to supply the necessary information by instantiating some data structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor’s bus logic determines which combinations are possible, and there will be a trade off between cost and performance: two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8
A single 8-bit flash device on an 8-bit bus.

16
A single 16-bit flash device on a 16-bit bus.

32
A single 32-bit flash device on a 32-bit bus.
Instantiating an AM29xxxx Device

88
Two parallel 8-bit devices on an 16-bit bus.

8888
Four parallel 8-bit devices on a 32-bit bus.

1616
Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the top two bytes.

16as8
A single 16-bit flash device connected to an 8-bit bus.

These configuration all require slightly different code to manipulate the hardware. The AM29xxxxx driver package provides separate functions for each configuration, for example cyg_am29xxxxx_erase_16 and cyg_am29xxxxx_program_1616.

Caution
At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors - both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so many flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have a single 16K block, followed by two 8K blocks, then a 32K block, and finally 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.

Example
In most cases flash support is specific to a platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code should be placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL’s CDL:

cdl_package CYGPKG_HAL_M68K_ALAIA {
    ...
    compile -library=libextras.a alaia_flash.c
    ...
}

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into libextras.a.

The actual file alaia_flash.c will look something like:

#include <pkgconf/system.h>
#undef CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2
The bulk of the file is protected by an #ifdef for the AM29xxxx flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so instantiating the data structures would serve no purpose. The rest of the file is split into three structure definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic flash code in cyg/io/flash_dev.h. The relevant ones have an _16 suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second provides information specific to AM29xxxx flash devices. The third provides the cyg_flash_dev structure needed by the generic flash code, which contains pointers to the previous two.

Functions

All eCos flash device drivers must implement a standard interface, defined by the generic flash code CYGPKG_IO_FLASH. This interface includes a table of seven function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation cyg_flash_devfn_query_nop. AM29xxxx flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared const. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. const should not be used if the table may be modified by a platform-specific initialization routine.
Instantiating an AM29xxxx Device

Initialization

There is a choice of three main initialization functions. The simplest is `cyg_flash_devfn_init_nop`, which does nothing. It can be used if the `cyg_am29xxxxx_dev` and `cyg_flash_dev` structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the nop function involves the smallest code size and run-time overheads.

The next step up is `cyg_am29xxxxx_init_check_devid_XX`, where `XX` will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the `devid` field in the `cyg_am29xxxxx_dev` structure. This initialization function will attempt to query the flash device and check that the provided device id matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a `cyg_am29xxxxx_init_cfi_XX` function. This queries the flash device as per the Common Flash Memory Interface Specification, supported by all current devices (although not necessarily by older devices). The `block_info` field in the `cyg_am29xxxxx_dev` structure and the `end` and `num_block_infos` fields in the `cyg_flash_dev` structure will be filled in. It is still necessary to supply the `start` field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it increases the code size.

Caution

If CFI is used then the `cyg_am29xxxxx_dev` structure must not be declared `const`. The CFI code will attempt to update the structure and will fail if the structure is held in read-only memory. This would leave the flash driver non-functional.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The AM29xxxxx driver provides a utility function to read the device id, `cyg_am29xxxxx_read_devid_XX`:

```c
static int
alaia_flash_init(struct cyg_flash_dev* dev)
{
    int devid = cyg_am29xxxxx_read_devid_1616(dev);
    switch(devid) {
        case 0x0042 :
            ...
        case 0x0084 :
            ...
        default:
            return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
}
```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this problem was fixed in the next revision. The platform-specific initialization function can figure out which model board it is running on and replace the default `16as8` functions with faster `16` ones.
Erase and Program

The AM29xxxxx driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. As another example some flash device may require a higher voltage to be applied during an erase or program operation. or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```c
static int
alaia_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
    int result;
    ... // Set up the hardware for an erase
    result = cyg_am29xxxxx_erase_32(dev, addr);
    ... // Revert the hardware change
    return result;
}
```

There are two configurations which affect the erase and program functions, and which a platform HAL may wish to change: CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_TIMEOUT and CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_PROGRAM_TIMEOUT. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, so a timeout indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL’s CDL can require appropriate values for these options.

Locking

There is no single way of implementing the block lock and unlock operations on all AM29xxxxx devices. If these operations are supported at all then usually they involve manipulating the voltages on certain pins. This would not be able to be handled by generic driver code since it requires knowing how these pins can be manipulated via the processor’s GPIO lines. Therefore the AM29xxxxx driver does not usually provide lock and unlock functions, and instead the generic dummy functions cyg_flash_devfn_lock_nop and cyg_flash_devfn_unlock_nop should be used. An exception exists for the AT49xxxx family of devices which are sufficiently AMD compatible in other respects. Otherwise, if a platform does provide a way of implementing the locking then this can be handled by platform-specific functions.

```c
static int
alaia_lock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    ...
}

static int
alaia_unlock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    ...
}
```
Instantiating an AM29xxxx Device

If real locking functions are implemented then the platform HAL’s CDL script should implement the CDL interface CYGHWR_IO_FLASH_BLOCK_LOCKING. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

AT49xxxx locking

As locking is standardised across the AT49xxxx family of AMD AM29xxxx compatible Flash parts, a method supporting this is included within this driver. cyg_at49xxxx_softlock_XX provides a means of locking a Flash sector such that it may be subsequently unlocked. cyg_at49xxxx_hardlock_XX locks a sector such that it cannot be unlocked until after reset or a power cycle. cyg_at49xxxx_unlock_XX unlocks a sector that has previously been softlocked. At power on or Flash device reset, all sectors default to being softlocked.

Other

The driver provides a set of functions cyg_am29xxxx_read_devid_XX, one per supported bus configuration. These functions take a single argument, a pointer to the cyg_flash_dev structure, and return the chip’s device id. For older devices this id is a single byte. For more recent devices the id is a 3-byte value, 0x7E followed by a further two bytes that actually identify the device. cyg_am29xxxx_read_devid_XX is usually called only from inside a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board.

Device-Specific Structure

The cyg_am29xxxx_dev structure provides information specific to AM29xxxx flash devices, as opposed to the more generic flash information which goes into the cyg_flash_dev structure. There are only two fields: devid and block_info.

devid is only needed if the driver’s initialization function is set to cyg_am29xxxx_init_check_devid_XX. That function will extract the actual device info from the flash chip and compare it with the devid field. If there is a mismatch then subsequent operations on the device will fail.

The block_info field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:

1. If the driver initialization function is set to cyg_flash_devfn_init_nop or cyg_am29xxxx_init_check_devid_XX then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.
2. If cyg_am29xxxx_init_cfi_XX is used then this will fill in the block info table. Hence there is no need for static initialization.
3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the block_info table is determined by the configuration option CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_REGIONS. This has a default value of 4, which should suffice for nearly all AM29xxxxx flash devices. If more entries are needed then the platform HAL’s CDL script should require a larger value.
If the cyg_am29xxxxx_dev structure is statically initialized then it can be `const`. This saves a small amount of memory in ROM startup applications. If the structure is updated at run-time, either by `cyg_am29xxxxx_init_cfi_XX` or by a platform-specific initialization routine, then it cannot be `const`.

### Flash Structure

Internally the generic flash code works in terms of `cyg_flash_dev` structures, and the platform HAL should define one of these. The structure should be placed in the `cyg_flashdev` table. The following fields need to be provided:

- **funs**
  
  This should point at the table of functions.

- **start**
  
  The base address of the flash in the address map. On some boards the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The `start` field should correspond to the cached address.

- **end**
  
  The address of the last byte in the flash. It can either be statically initialized, or `cyg_am29xxxxx_init_cfi_XX` will calculate its value at run-time.

- **num_block_infos**
  
  This should be the number of entries in the `block_info` table. It can either be statically initialized or it will be filled in by `cyg_am29xxxxx_init_cfi_XX`.

- **block_info**
  
  The table with the block information is held in the `cyg_am29xxxxx_dev` structure, so this field should just point into that structure.

- **priv**
  
  This field is reserved for use by the device driver. For the AM29xxxxx driver it should point at the appropriate `cyg_am29xxxxx_dev` structure.

The `cyg_flash_dev` structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared `const`.

### Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two `cyg_flash_dev` structures. The table of function pointers can usually be shared, as can the `cyg_am29xxxxx_dev` structure. Another approach is to define a single `cyg_flash_dev` structure but with a larger `block_info` table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate `cyg_flash_dev` structures.
Instantiating an AM29xxxx Device

If more than one cyg_flash_dev structure is instantiated then the platform HAL’s CDL script should implement the CDL interface CYGHWR_IO_FLASH_DEVICE once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The AM29xxxxx driver source code includes the header files cyg/hal/hal_arch.h and cyg/hal/hal_io.h, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the AM29xxxxx driver assumes that the flash can be accessed uncached, and it will use the HAL CYGARC_UNCACHED_ADDRESS macro to map the cached address in the start field of the cyg_flash_dev structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro HAL_AM29XXXXX_UNCACHED_ADDRESS can be defined instead. However fixing the CYGARC_UNCACHED_ADDRESS macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface CYGHWR_DEVS_FLASH_AMD_AM29XXXXX_V2_CACHED_ONLY. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```
AM29_INTSCACHE_STATE;
AM29_INTSCACHE_BEGIN();
while ( ! finished ) {
    program data
}
AM29_INTSCACHE_END();
```

The default implementations of these INTSCACHE macros are as follows: STATE defines any local variables that may be needed, e.g. to save the current interrupt state; BEGIN disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; END re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data.

Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example HAL_DCACHE_DISABLE may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros HAL_AM29XXXXX_INTSCACHE_STATE, HAL_AM29XXXXX_INTSCACHE_BEGIN and HAL_AM29XXXXX_INTSCACHE_END, using these instead of the defaults.
LII. Intel Strata Flash Device Driver
Instantiating an AM29xxxx Device
Overview

Name
Overview — eCos Support for Intel Strata Flash Devices and Compatibles

Description
The CYGPKG_DEVS_FLASH_STRATA_V2 flash driver package implements support for the Intel Strata family of flash devices and compatibles. The driver is not normally accessed directly. Instead application code will use the API provided by the generic flash driver package CYGPKG_IO_FLASH, for example by calling functions like cyg_flash_program. There are a small number of additional functions specific to Strata devices.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note: Many eCos targets with Strata or compatible flash devices will still use the older driver package CYGPKG_DEVS_FLASH_STRATA. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options
The Strata flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the Strata driver package.

There are a number of configuration options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_REGIONS may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.
Instantiating a Strata Device

Name

Instantiating — including the driver in an eCos target

Synopsis

```c
#include <cyg/io/strata_dev.h>

int cyg_strata_init_nop(struct cyg_flash_dev* device);
int cyg_strata_init_check_devid_XX(struct cyg_flash_dev* device);
int cyg_strata_init_cfi_XX(struct cyg_flash_dev* device);
int cyg_strata_erase_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr);
int cyg_strata_program_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr, const void* data, size_t len);
int cyg_strata_bufprogram_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr, const void* data, size_t len);
int cyg_strata_lock_j3_XX(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_strata_unlock_j3_XX(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_strata_lock_k3_XX(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_strata_unlock_k3_XX(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
```

Description

The Strata family contains a number of different devices, all supporting the same basic set of operations but with various common or uncommon extensions. The range includes:

28FxxxB3 Boot Block

These support 8 8K boot blocks as well as the usual 64K blocks. There is no buffered write capability. The only locking mechanism available involves manipulating voltages on certain pins.

28FxxxC3

These also have boot blocks. There is no buffered write capability. Individual blocks can be locked and unlocked in software.

28FxxxJ3

These are uniform devices where all blocks are 128K. Buffered writes are supported. Blocks can be locked individually, but the only unlock operation is a global unlock-all.

28FxxxK3

These are also uniform devices with 128K blocks. Buffered writes are supported. Individual blocks can be locked and unlocked in software.

Each of these comes in a range of sizes and bus widths. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The Strata driver package cannot know all this information. Instead it is the responsibility of another package, usually the
platform HAL, to instantiate some flash device structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor’s bus logic determines which combinations are possible, and usually there will be a trade off between cost and performance. For example two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8
 achieving eight bits on a 16-bit bus.

16
 A single 16-bit flash device on a 16-bit bus.

32
 A single 32-bit flash device on a 32-bit bus.

88
 Two parallel 8-bit devices on an 16-bit bus.

8888
 Four parallel 8-bit devices on a 32-bit bus.

1616
 Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the upper two bytes.

16as8
 A single 16-bit flash device connected to an 8-bit bus.

These configuration all require slightly different code to manipulate the hardware. The Strata driver package provides separate functions for each configuration, for example `cyg_strata_erase_16` and `cyg_strata_program_1616`.

Caution

At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors, both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so some flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have eight 8K blocks and 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.
Instantiating a Strata Device

Example

Flash support is usually specific to each platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code is generally placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL’s CDL:

cdl_package CYGPKG_HAL_M68K_KIKOO {
    ...
    compile -library=libextras.a kikoo_flash.c
    ...
}

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into libextras.a.

The actual file kikoo_flash.c will look something like:

```c
#include <pkgconf/system.h>
#ifdef CYGPKG_DEVS_FLASH_STRATA_V2
#include <cyg/io/flash.h>
#include <cyg/io/strata_dev.h>

static const CYG_FLASH_FUNS(hal_kikoo_flash_strata_funs,
    &cyg_strata_init_check_devid_16,
    &cyg_flash_devfn_query_nop,
    &cyg_strata_erase_16,
    &cyg_strata_bufprogram_16,
    (int (*)(struct cyg_flash_dev*, const cyg_flashaddr_t, void*, size_t))0,
    &cyg_strata_lock_j3_16,
    &cyg_strata_unlock_j3_16);

static const cyg_strata_dev hal_kikoo_flash_priv = {
    .manufacturer_code = CYG_FLASH_STRATA_MANUFACTURER_INTEL,
    .device_code = 0x0017,
    .bufsize = 16,
    .block_info = {
        { 0x00020000, 64 } // 64 * 128K blocks
    }
};

CYG_FLASH_DRIVER(hal_kikoo_flash,
    &hal_kikoo_flash_strata_funs,
    0,
    0x60000000,
    0x601FFFFF,
    1,
    hal_kikoo_flash_priv.block_info,
    &hal_kikoo_flash_priv
);
#endif
```

The bulk of the file is protected by an ifdef for the Strata flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so there is no point in instantiating the device. The rest of the file is split into three definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic
flash code in cyg/io/flash_dev.h. The relevant ones have an _16 suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second definition provides information specific to Strata flash devices. The third provides the cyg_flash_dev structure needed by the generic flash code, which contains pointers to the previous two.

**Functions**

All eCos flash device drivers must implement a standard interface, defined by the generic flash code CYGPKG_IO_FLASH. This interface includes a table of 7 function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation cyg_flash_devfn_query_nop. Strata flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared const. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. const should not be used if the table may be modified by a platform-specific initialization routine.

**Initialization**

There is a choice of three main initialization functions. The simplest is cyg_flash_devfn_init_nop, which does nothing. It can be used if the cyg_strata_dev and cyg_flash_dev structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the nop function involves the smallest code size and run-time overheads.

The next step up is cyg_strata_init_check_devid_XX, where XX will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the devid field in the cyg_strata_dev structure. However this initialization function will attempt to query the flash device and check that the provided manufacturer and device codes matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a cyg_strata_init_cfi_XX function. This queries the flash device as per the Common Flash Memory Interface Specification, supported by all current devices (although not necessarily by older devices). The block_info field in the cyg_strata_dev structure and the end and num_block_infos fields in the cyg_flash_dev structure will be filled in. It is still necessary to supply the start field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it will increase the code size.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The Strata driver provides a utility function to read the device id, cyg_strata_read_devid_XX:

```
static int
kikoo_flash_init(struct cyg_flash_dev* dev)
{
  int manufacturer_code, device_code;
  cyg_strata_read_devid_1616(dev, &manufacturer_code, &device_code);
  if (manufacturer_code != CYG_FLASH_STRATA_MANUFACTURER_STMICRO) {
    return CYG_FLASH_ERR_DRV_WRONG_PART;
  }
  switch(device_code) {
    case 0x0042 :
      ...
```
Instantiating a Strata Device

```c
    case 0x0084 :
      ...
    default:
      return CYG_FLASH_ERR_DRV_WRONG_PART;
  }
}
```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this was fixed in the next revision. The platform-specific initialization function could figure out which model board it is running on and replace the default 16as8 functions with 16 ones.

Erase and Program

The Strata driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. For another example some flash device may require a higher voltage to be applied during an erase or program operation, or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```c
static int
kikoo_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
  int result;
  ...
  result = cyg_strata_erase_32(dev, addr);
  ...
  return result;
}
```

There are two versions of the program function. cyg_strata_bufprogram_xx uses the buffered write capability of some strata chips. This allows the flash chip to perform the writes in parallel, thus greatly improving performance. It requires that the bufsize field of the cyg_strata_dev structure is set correctly to the number of words in the write buffer. The usual value for this is 16, corresponding to a 32-byte write buffer. The alternative cyg_strata_program_xx writes the data one word at a time so is significantly slower. It should be used only with strata chips that do not support buffered writes, for example the b3 and c3 series.

There are two configuration options which affect the erase and program functions, and which a platform HAL may wish to change: CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_TIMEOUT and CYGNUM_DEVS_FLASH_STRATA_V2_PROGRAM_TIMEOUT. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, and hence a timeout probably indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL’s CDL can require appropriate values for these options.
Locking

Current Strata devices implement locking in three different ways, requiring different sets of functions:

28FxxxB3

There is no software locking support. The cyg_flash_devfn_lock_nop and cyg_flash_devfn_unlock_nop functions should be used.

28FxxxC3
28FxxxK3

These support locking and unlocking individual blocks. The cyg_strata_lock_k3_XX and cyg_strata_unlock_k3_XX functions should be used. All blocks are locked following power-up or reset, so the unlock function must be used before any erase or program operation. Theoretically the lock function is optional and cyg_flash_devfn_lock_nop can be used instead, saving a small amount of code space.

28FxxxJ3

Individual blocks can be locked using cyg_strata_lock_j3_XX, albeit using a slightly different algorithm from the C3 and K3 series. However the only unlock support is a global unlock of all blocks. Hence the only way to unlock a single block is to check the locked status of every block, unlock them all, and relock the ones that should still be locked. This time-consuming operation is implemented by cyg_strata_unlock_j3_XX. Worse, unlocking all blocks can take approximately a second. During this time the flash is unusable so normally interrupts have to be disabled, affecting real-time responsiveness. There is no way of suspending this operation.

Unlike the C3 and K3 chips, on a J3 blocks are not automatically locked following power-up or reset. Hence lock and unlock support is optional, and cyg_flash_devfn_lock_nop and cyg_flash_devfn_unlock_nop can be used.

If real locking functions are used then the platform HAL’s CDL script should implement the CDL interface CYGHWR_IO_FLASH_BLOCK_LOCKING. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

Device-Specific Structure

The cyg_strata_dev structure provides information specific to Strata flash devices, as opposed to the more generic flash information which goes into the cyg_flash_dev structure. There are only two fields: devid and block_info.

manufacturer_code and device_code are needed only if the driver’s initialization function is set to cyg_strata_init_check_devid_XX. That function will extract the actual device info from the flash chip and compare it with these fields. If there is a mismatch then subsequent operations on the device will fail. Definitions of CYG_FLASH_STRATA_MANUFACTURER_INTEL and CYG_FLASH_STRATA_MANUFACTURER_STMICRO are provided for convenience.

The bufsize field is needed only if a buffered program function cyg_strata_bufprogram_XX is used. It should give the size of the buffer in words. Typically Strata devices have a 32-byte buffer, so when attached to an 8-bit bus bufsize should be 32 and when attached to a 16-bit bus it should be 16.

The block_info field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:
Instantiating a Strata Device

1. If the driver initialization function is set to `cyg_strata_init_nop` or `cyg_strata_init_check_devid_XX` then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.

2. If `cyg_strata_init_cfi_XX` is used then this will fill in the block info table. Hence there is no need for static initialization.

3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the `block_info` table is determined by the configuration option `CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_REGIONS`. This has a default value of 2, which should suffice for nearly all Strata flash devices. If more entries are needed then the platform HAL’s CDL script should require a larger value.

If the `cyg_strata_dev` structure is statically initialized then it can be `const`. This saves a small amount of memory in ROM startup applications. If the structure may be updated at run-time, either by `cyg_strata_init_cfi_XX` or by a platform-specific initialization routine, then it cannot be `const`.

**Flash Structure**

Internally the flash code works in terms of `cyg_flash_dev` structures, and the platform HAL should define one of these. The structure should be placed in the `cyg_flashdev` table. The following fields need to be provided:

- **funs**
  
  This should point at the table of functions.

- **start**
  
  The base address of the flash in the address map. On some board the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The `start` field should correspond to the cached address.

- **end**
  
  The address of the last byte in the flash. It can either be statically initialized, or `cyg_strata_init_cfi_XX` will calculate its value at run-time.

- **num_block_infos**
  
  This should be the number of entries in the `block_info` table. It can either be statically initialized or it will be filled in by `cyg_strata_init_cfi_XX`.

- **block_info**
  
  The table with the block information is held in the `cyg_strata_dev` structure, so this field should just point into that structure.

- **priv**
  
  This field is reserved for use by the device driver. For the Strata driver it should point at the appropriate `cyg_strata_dev` structure.

The `cyg_flash_dev` structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared `const`. 

841
Instantiating a Strata Device

Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two cyg_flash_dev structures. The table of function pointers can usually be shared, as can the cyg_strata_dev structure. Another approach is to define a single cyg_flash_dev structure but with a larger block_info table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate cyg_flash_dev structures.

If more than one cyg_flash_dev structure is instantiated then the platform HAL’s CDL script should implement the CDL interface CYGHWR_IO_FLASH_DEVICE once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The Strata driver source code includes the header files cyg/hal/hal_arch.h and cyg/hal/hal_io.h, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the strata driver assumes that the flash can be accessed uncached, and it will use the HAL CYGARC_UNCACHED_ADDRESS macro to map the cached address in the start field of the cyg_flash_dev structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro HAL_STRATA_UNCACHED_ADDRESS can be defined instead. However fixing the CYGARC_UNCACHED_ADDRESS macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface CYGHWR_DEVS_FLASH_STRATA_V2_CACHED_ONLY. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```
STRATA_INTSCACHE_STATE;
STRATA_INTSCACHE_BEGIN();
while (! finished ) {
    program data
}
STRATA_INTSCACHE_END();
```

The default implementations of these INTSCACHE macros are as follows: STATE defines any local variables that may be needed, e.g. to save the current interrupt state; BEGIN disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; END re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data.
Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example `HAL_DCACHE_DISABLE` may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros `HAL_STRATA_INTSCACHE_STATE`, `HAL_STRATA_INTSCACHE_BEGIN` and `HAL_STRATA_INTSCACHE_END`, using these instead of the defaults.
Instantiating a Strata Device
Strata-Specific Functions

Name

Strata — driver-specific functions

Synopsis

```c
#include <cyg/io/strata_dev.h>

void cyg_strata_read_devid_XX(struct cyg_flash_dev* device, cyg_uint32* manufacturer, cyg_uint32* device);
int cyg_strata_unlock_all_j3_XX(struct cyg_flash_dev* device);
```

Description

The driver provides two sets of functions specific to Strata devices and not accessible via the standard eCos flash API. Both may be used safely before the flash subsystem is initialized using `cyg_flash_init`.

cyg_strata_read_devid_XX can be used to get the manufacturer and device codes. Typically it is called from a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board. This may be useful if a board may get manufactured with several different and somewhat incompatible chips, although usually cyg_strata_init_cfi is the better approach. It may also be used during testing and porting to check that the chip is working correctly.

cyg_strata_unlock_all_j3_XX is only useful with 28FxxxJ3 chips and compatibles. These do not allow individual blocks to be unlocked. Hence the standard block unlock functionality is expensive: it requires checking the locked state of every block, unlocking every block, and then relocking all the blocks that should still be blocked. Worse, unlocking every block is a time-consuming operation, taking approximately a second, that needs to run with interrupts disabled. For many applications it is better to just ignore the chip’s locking capabilities and run with all blocks permanently unlocked. Invoking cyg_strata_unlock_all_j3_XX during manufacture or when the board is commissioned achieves this.
Strata-Specific Functions
LIII. Motorola MCF52xx ColdFire I\(^2\)C Bus Driver
Strata-Specific Functions
Motorola MCF52xx Coldfire I²C Bus Driver

Name

CYGPKG_DEVS_I2C_MCF52xx — eCos Support for the Motorola Coldfire I²C Bus

Description

Several processors in the Motorola ColdFire family come with one or more on-chip I²C bus devices. This package provides an eCos I²C bus driver. It was originally developed on an MCF5280 but should work with any ColdFire processor that uses a compatible bus device. The driver implements the functionality defined by the generic I²C package CYGPKG_IO_I2C.

Caution

The hardware does not support DMA or fifos, so usually a transfer will involve an interrupt for every byte transferred. Since the I²C bus typically runs at 100KHz large transfers will consume much of the available cpu time.

This package does not provide any cyg_i2c_bus structures. The number of I²C buses varies between ColdFire processors. If multiple buses are available then exactly which one(s) are in use on a given hardware platform depends entirely on that platform. The desired I²C bus speed also depends on the platform, and there may be other issues such as how the processor pins should be set up. Hence it is left to other code, usually the platform HAL, to instantiate the bus structure(s). This driver package supplies the necessary functions and utility macros. Similarly this package does not provide any cyg_i2c_device structures. Which I²C devices are hooked up to which I²C bus is entirely a characteristic of the hardware platform, so again it is up to the platform HAL to instantiate the necessary structures.

The driver will operate in interrupt-driven mode if interrupts are enabled when a transfer is initiated. Otherwise it will operate in polled mode. This allows the driver to be used in a variety of configurations including inside RedBoot.

Configuration Options

The I²C bus driver package should be loaded automatically when selecting a target containing a suitable ColdFire processor, and it should never be necessary to load the package explicitly. If the application does not use any of the I²C functionality, directly or indirectly, then all the I²C code should be removed at link-time and the application does not suffer any overheads.

By default the driver assumes a single I²C bus and optimizes for that case. For example options like the ISR vector and priority are handled by compile-time #define’s in the platform HAL’s exported header files rather than by per-bus structure fields. This helps to reduce both code and data overheads. If the driver should support multiple I²C buses then CYGHWR_DEVS_I2C_MCF52xx_MULTIPLE_BUSES should be enabled. Typically this will be done by the platform HAL using a CDL requires property. If bus instantiation happens outside the platform HAL and hence the HAL’s header files do not provide the appropriate definitions, then this configuration option should also be defined.

The only other configuration options in this package provide control over the compiler flags used to build the driver code.
Defining the Bus and Devices

For most hardware targets the platform HAL will instantiate the cyg_i2c_bus and cyg_i2c_device structures, and it will also initialize the hardware so that the I²C-related pins are connected appropriately. Some development boards have no I²C devices, but the I²C bus signals are accessible via an expansion connector and I²C devices can be put on a daughter board. In such cases it may be necessary for the application to instantiate both the bus and all the device structures. Alternatively the platform HAL may provide a configuration option to enable just the bus, with the devices still left to application code.

To facilitate bus instantiation the header file cyg/io/i2c_mcf52xx.h provides a utility macro CYG_MCF52xx_I2C_BUS. This takes six parameters:

1. The name of the bus, for example hal_dnp5280_i2c_bus. This name will be used when instantiating the I²C devices.
2. An initialization function. If no platform-specific initialization is needed then this can be the cyg_mcf52xx_i2c_init function exported by this driver. Otherwise it can be a platform-specific function which, for example, sets up the relevant pins appropriately and then chains into cyg_mcf52xx_i2c_init.
3. The base address of the I²C bus. For example on an MCF5282 with the IPSBAR set to its usual value of 0x40000000, the I²C bus is at location 0x40000300.
4. The interrupt vector, for example CYGNUM_HAL_ISR_I2C_IIF on an MCF5282.
5. The interrupt priority. Typically this will be a configurable option within the platform HAL.
6. A value for the I²C bus’s I2FDR register. That register controls the bus speed. Typical bus speeds are 100KHz and 400KHz, depending on the capabilities of the attached devices. There is no simple relationship between the system clock speed, the desired bus speed, and the FDR register. Although the driver could determine the FDR setting using a lookup table and appropriate code, it is better to determine the correct value once during the porting process and avoid unnecessary run-time overheads.

For the common case where only a single I²C bus should be supported (CYGHWR_DEVS_I2C_MCF52xx_MULTIPLE_BUSES is disabled), the last four parameters should be provided by preprocessor #define’s, typically in cyg/hal/plf_io.h which gets #include’d automatically via cyg/hal/hal_io.h. This header can also define the HAL_I2CEXPORTED_DEVICES macro as per the generic I²C package:

```c
#include <pkgconf/hal_m68k_dnp5280.h>

 ifndef CYGHWR_HAL_M68K_DNP5280_I2C
#define HAL_MCF52xx_I2C_SINGLETON_BASE (HAL_MCF52xx_MBAR+HAL_MCF5282_I2C0_BASE)
#define HAL_MCF52xx_I2C_SINGLETON_ISRVEC CYGNUM_HAL_ISR_I2C_IIF
#define HAL_MCF52xx_I2C_SINGLETON_ISRPRI CYGNUM_HAL_M68K_DNP5280_I2C_ISRPRI
#define HAL_MCF52xx_I2C_SINGLETON_FDR CYGNUM_HAL_M68K_DNP5280_I2C_FDR

#define HAL_I2CEXPORTED_DEVICES \ 
    extern cyg_i2c_bus hal_dnp5280_i2c_bus;
#endif
```

On this particular platform the I²C bus is only accessible on an expansion connector so the support is conditional on a configuration option CYGHWR_HAL_M68K_DNP5280_I2C. The interrupt priority and I2FDR values are also controlled by configuration options. On other platforms the I²C support may not be conditional and the priority and/or FDR values may be hard-wired.
The I²C bus instantiation should happen in an ordinary C or C++ file, typically in the platform HAL. The corresponding object file should go into libtarget.a and the file should only contain I²C-related code to get the maximum benefit of linker garbage collection.

```c
#include <cyg/infra/cyg_type.h>
#include <cyg/hal/hal_io.h>
#include <cyg/io/i2c.h>
#include <cyg/io/i2c_mcf52xx.h>

static void
dnp5280_i2c_init(struct cyg_i2c_bus* bus)
{
    cyg_uint16 paspar;
    // Reset GPIO pins PAS0/1 to their alternative SCL/SDA settings
    HAL_READ_UINT16(HAL_MCF5282_IPSBAR + HAL_MCF5282_GPIO_PASPAR, paspar);
    paspar &= ~(HAL_MCF5282_GPIO_PASPAR_A0_MASK | HAL_MCF5282_GPIO_PASPAR_A1_MASK);
    paspar |= (HAL_MCF5282_GPIO_PASPAR_A0_SCL | HAL_MCF5282_GPIO_PASPAR_A1_SDA);
    HAL_WRITE_UINT16(HAL_MCF5282_IPSBAR + HAL_MCF5282_GPIO_PASPAR, paspar);

    // And leave the driver to take care of the rest.
    cyg_mcf52xx_i2c_init(bus);
}

CYG_MCF52xx_I2C_BUS(hal_dnp5280_i2c_bus,
    &dnp5280_i2c_init,
    HAL_MCF52xx_I2C_SINGLETON_BASE,
    HAL_MCF52xx_I2C_SINGLETON_ISRVEC,
    HAL_MCF52xx_I2C_SINGLETON_ISRPRI,
    HAL_MCF52xx_I2C_SINGLETON_FDR);
```

Obviously if CYGHWR_DEVS_I2C_MCF52xx_MULTIPLE_BUSES is enabled then the singleton macros may not be defined and the appropriate numbers should be used directly. This example uses a custom initialization function which sets up the relevant pins and then chains into the I²C drivers' cyg_mcf52xx_i2c_init function. If the platform HAL has already set up the pins correctly then cyg_mcf52xx_i2c_init could be used directly in the bus instantiation, saving a small amount of code for the custom initialization function.

I²C device structures can be instantiated in the usual way, for example:

```c
CYG_I2C_DEVICE(cyg_i2c_wallclock_ds1307,
    &hal_dnp5280_i2c_bus,
    0x68,
    0x00,
    CYG_I2C_DEFAULT_DELAY);
```