RTEMS™

www.rtems.org

# RTEMS Classic API Guide

*Release 6.2 (19th December 2025)*

© 1988-2025 RTEMS Project and contributors

# CONTENTS


**1 Preface** **3**

**2 Overview** **7**
2.1 Introduction . . . 8
2.2 Real-time Application Systems . . . 9
2.3 Real-time Executive . . . 10
2.4 RTEMS Application Architecture . . . 11
2.5 RTEMS Internal Architecture . . . 12
2.6 User Customization and Extensibility . . . 14
2.7 Portability . . . 15
2.8 Memory Requirements . . . 16
2.9 Audience . . . 17
2.10 Conventions . . . 18
2.11 Manual Organization . . . 19

**3 Key Concepts** **23**
3.1 Introduction . . . 24
3.2 Objects . . . 25
3.2.1 Object Names . . . 25
3.2.2 Object Ids . . . 26
3.2.3 Local and Global Scope . . . 26
3.2.3.1 Object ID Format . . . 26
3.2.4 Object ID Description . . . 27
3.3 Communication and Synchronization . . . 28
3.4 Locking Protocols . . . 29
3.4.1 Priority Inversion . . . 29
3.4.2 Immediate Ceiling Priority Protocol (ICPP) . . . 29
3.4.3 Priority Inheritance Protocol . . . 29
3.4.4 Multiprocessor Resource Sharing Protocol (MrsP) . . . 30
3.4.5 O(m) Independence-Preserving Protocol (OMIP) . . . 30
3.5 Thread Queues . . . 31
3.6 Time . . . 32
3.7 Timer and Timeouts . . . 33
3.8 Memory Management . . . 34

**4 RTEMS Data Types** **35**
4.1 Introduction . . . 36
4.2 List of Data Types . . . 37

4.2.1 BSP_output_char_function_type . . . 37
4.2.2 BSP_polling_getchar_function_type . . . 37
4.2.3 Timer_Classes . . . 37
4.2.4 rtems_api_configuration_table . . . 37
4.2.5 rtems_asr . . . 38
4.2.6 rtems_asr_entry . . . 38
4.2.7 rtems_assert_context . . . 38
4.2.8 rtems_attribute . . . 39
4.2.9 rtems_device_driver . . . 39
4.2.10 rtems_device_driver_entry . . . 39
4.2.11 rtems_device_major_number . . . 39
4.2.12 rtems_device_minor_number . . . 39
4.2.13 rtems_driver_address_table . . . 40
4.2.14 rtems_event_set . . . 40
4.2.15 rtems_exception_frame . . . 40
4.2.16 rtems_extensions_table . . . 40
4.2.17 rtems_fatal_code . . . 40
4.2.18 rtems_fatal_extension . . . 41
4.2.19 rtems_fatal_source . . . 41
4.2.20 rtems_id . . . 41
4.2.21 rtems_initialization_tasks_table . . . 42
4.2.22 rtems_interrupt_attributes . . . 42
4.2.23 rtems_interrupt_entry . . . 44
4.2.24 rtems_interrupt_handler . . . 44
4.2.25 rtems_interrupt_level . . . 44
4.2.26 rtems_interrupt_lock . . . 44
4.2.27 rtems_interrupt_lock_context . . . 45
4.2.28 rtems_interrupt_per_handler_routine . . . 45
4.2.29 rtems_interrupt_server_action . . . 45
4.2.30 rtems_interrupt_server_config . . . 45
4.2.31 rtems_interrupt_server_control . . . 45
4.2.32 rtems_interrupt_server_entry . . . 46
4.2.33 rtems_interrupt_server_request . . . 46
4.2.34 rtems_interrupt_signal_variant . . . 46
4.2.35 rtems_interval . . . 47
4.2.36 rtems_isr . . . 47
4.2.37 rtems_isr_entry . . . 47
4.2.38 rtems_message_queue_config . . . 47
4.2.39 rtems_mode . . . 47
4.2.40 rtems_mp_packet_classes . . . 48
4.2.41 rtems_mpci_entry . . . 48
4.2.42 rtems_mpci_get_packet_entry . . . 48
4.2.43 rtems_mpci_initialization_entry . . . 48
4.2.44 rtems_mpci_receive_packet_entry . . . 48
4.2.45 rtems_mpci_return_packet_entry . . . 48
4.2.46 rtems_mpci_send_packet_entry . . . 48
4.2.47 rtems_mpci_table . . . 48
4.2.48 rtems_multiprocessing_table . . . 48
4.2.49 rtems_name . . . 48
4.2.50 rtems_object_api_class_information . . . 48
4.2.51 rtems_option . . . 49

4.2.52 rtems_packet_prefix . . . 49
4.2.53 rtems_rate_monotonic_period_states . . . 49
4.2.54 rtems_rate_monotonic_period_statistics . . . 49
4.2.55 rtems_rate_monotonic_period_status . . . 50
4.2.56 rtems_regulator_attributes . . . 50
4.2.57 rtems_regulator_deliverer . . . 51
4.2.58 rtems_regulator_statistics . . . 51
4.2.59 rtems_signal_set . . . 52
4.2.60 rtems_stack_allocate_hook . . . 52
4.2.61 rtems_stack_allocate_init_hook . . . 52
4.2.62 rtems_stack_free_hook . . . 52
4.2.63 rtems_status_code . . . 52
4.2.64 rtems_task . . . 54
4.2.65 rtems_task_argument . . . 54
4.2.66 rtems_task_begin_extension . . . 54
4.2.67 rtems_task_config . . . 54
4.2.68 rtems_task_create_extension . . . 55
4.2.69 rtems_task_delete_extension . . . 56
4.2.70 rtems_task_entry . . . 56
4.2.71 rtems_task_exitted_extension . . . 56
4.2.72 rtems_task_priority . . . 57
4.2.73 rtems_task_restart_extension . . . 57
4.2.74 rtems_task_start_extension . . . 57
4.2.75 rtems_task_switch_extension . . . 57
4.2.76 rtems_task_terminate_extension . . . 58
4.2.77 rtems_task_visitor . . . 59
4.2.78 rtems_tcb . . . 59
4.2.79 rtems_time_of_day . . . 59
4.2.80 rtems_timer_information . . . 59
4.2.81 rtems_timer_service_routine . . . 60
4.2.82 rtems_timer_service_routine_entry . . . 60
4.2.83 rtems_vector_number . . . 60

**5 Scheduling Concepts 61**
5.1 Introduction . . . 62
5.2 Background . . . 63
5.2.1 Scheduling Algorithms . . . 63
5.2.2 Priority Scheduling . . . 63
5.2.3 Scheduling Modification Mechanisms . . . 64
5.2.3.1 Task Priority and Scheduling . . . 64
5.2.3.2 Preemption . . . 64
5.2.3.3 Timeslicing . . . 64
5.2.3.4 Manual Round-Robin . . . 65
5.2.4 Dispatching Tasks . . . 65
5.2.5 Task State Transitions . . . 65
5.3 Uniprocessor Schedulers . . . 68
5.3.1 Deterministic Priority Scheduler . . . 68
5.3.2 Simple Priority Scheduler . . . 68
5.3.3 Earliest Deadline First Scheduler . . . 68
5.3.4 Constant Bandwidth Server Scheduling (CBS) . . . 69
5.4 SMP Schedulers . . . 70

5.4.1 Earliest Deadline First SMP Scheduler . . . 70
5.4.2 Deterministic Priority SMP Scheduler . . . 70
5.4.3 Simple Priority SMP Scheduler . . . 70
5.4.4 Arbitrary Processor Affinity Priority SMP Scheduler . . . 70
5.5 Directives . . . 71
5.5.1 rtems_scheduler_ident() . . . 72
5.5.2 rtems_scheduler_ident_by_processor() . . . 73
5.5.3 rtems_scheduler_ident_by_processor_set() . . . 74
5.5.4 rtems_scheduler_get_maximum_priority() . . . 76
5.5.5 rtems_scheduler_map_priority_to_posix() . . . 77
5.5.6 rtems_scheduler_map_priority_from_posix() . . . 78
5.5.7 rtems_scheduler_get_processor() . . . 79
5.5.8 rtems_scheduler_get_processor_maximum() . . . 80
5.5.9 rtems_scheduler_get_processor_set() . . . 81
5.5.10 rtems_scheduler_add_processor() . . . 82
5.5.11 rtems_scheduler_remove_processor() . . . 83

**6 Initialization Manager 85**
6.1 Introduction . . . 86
6.2 Background . . . 87
6.2.1 Initialization Tasks . . . 87
6.2.2 The Idle Task . . . 87
6.2.3 Initialization Manager Failure . . . 87
6.3 Operations . . . 88
6.3.1 Initializing RTEMS . . . 88
6.3.2 Global Construction . . . 92
6.4 Directives . . . 94
6.4.1 rtems_initialize_executive() . . . 95

**7 Task Manager 97**
7.1 Introduction . . . 98
7.2 Background . . . 99
7.2.1 Task Definition . . . 99
7.2.2 Task Control Block . . . 99
7.2.3 Task Memory . . . 99
7.2.4 Task Name . . . 100
7.2.5 Task States . . . 100
7.2.6 Task Priority . . . 100
7.2.7 Task Mode . . . 101
7.2.8 Task Life States . . . 102
7.2.9 Accessing Task Arguments . . . 102
7.2.10 Floating Point Considerations . . . 103
7.2.11 Building a Task Attribute Set . . . 103
7.2.12 Building a Mode and Mask . . . 104
7.3 Operations . . . 105
7.3.1 Creating Tasks . . . 105
7.3.2 Obtaining Task IDs . . . 105
7.3.3 Starting and Restarting Tasks . . . 105
7.3.4 Suspending and Resuming Tasks . . . 105
7.3.5 Delaying the Currently Executing Task . . . 106
7.3.6 Changing Task Priority . . . 106
7.3.7 Changing Task Mode . . . 106

7.3.8 Task Deletion . . . 106
7.3.9 Setting Affinity to a Single Processor . . . 106
7.3.10 Transition Advice for Removed Notepads . . . 107
7.3.11 Transition Advice for Removed Task Variables . . . 107
7.4 Directives . . . 108
7.4.1 rtems_task_create() . . . 109
7.4.2 rtems_task_construct() . . . 113
7.4.3 rtems_task_ident() . . . 115
7.4.4 rtems_task_self() . . . 117
7.4.5 rtems_task_start() . . . 118
7.4.6 rtems_task_restart() . . . 120
7.4.7 rtems_task_delete() . . . 122
7.4.8 rtems_task_exit() . . . 124
7.4.9 rtems_task_suspend() . . . 125
7.4.10 rtems_task_resume() . . . 126
7.4.11 rtems_task_is_suspended() . . . 127
7.4.12 rtems_task_set_priority() . . . 128
7.4.13 rtems_task_get_priority() . . . 130
7.4.14 rtems_task_mode() . . . 132
7.4.15 rtems_task_wake_after() . . . 134
7.4.16 rtems_task_wake_when() . . . 135
7.4.17 rtems_task_get_scheduler() . . . 136
7.4.18 rtems_task_set_scheduler() . . . 137
7.4.19 rtems_task_get_affinity() . . . 139
7.4.20 rtems_task_set_affinity() . . . 141
7.4.21 rtems_task_iterate() . . . 143
7.4.22 RTEMS_TASK_STORAGE_SIZE() . . . 144
7.5 Deprecated Directives . . . 145
7.5.1 ITERATE_OVER_ALL_THREADS - Iterate Over Tasks . . . 146
7.6 Removed Directives . . . 147
7.6.1 TASK_GET_NOTE - Get task notepad entry . . . 148
7.6.2 TASK_SET_NOTE - Set task notepad entry . . . 149
7.6.3 TASK_VARIABLE_ADD - Associate per task variable . . . 150
7.6.4 TASK_VARIABLE_GET - Obtain value of a per task variable . . . 151
7.6.5 TASK_VARIABLE_DELETE - Remove per task variable . . . 152

**8 Interrupt Manager 153**
8.1 Introduction . . . 154
8.2 Background . . . 157
8.2.1 Processing an Interrupt . . . 157
8.2.2 RTEMS Interrupt Levels . . . 158
8.2.3 Disabling of Interrupts by RTEMS . . . 158
8.3 Operations . . . 159
8.3.1 Establishing an ISR . . . 159
8.3.2 Directives Allowed from an ISR . . . 159
8.4 Directives . . . 161
8.4.1 rtems_interrupt_catch() . . . 162
8.4.2 rtems_interrupt_disable() . . . 164
8.4.3 rtems_interrupt_enable() . . . 166
8.4.4 rtems_interrupt_flash() . . . 167
8.4.5 rtems_interrupt_local_disable() . . . 168

8.4.6 rtems_interrupt_local_enable() . . . 170
8.4.7 rtems_interrupt_is_in_progress() . . . 171
8.4.8 rtems_interrupt_lock_initialize() . . . 172
8.4.9 rtems_interrupt_lock_destroy() . . . 173
8.4.10 rtems_interrupt_lock_acquire() . . . 174
8.4.11 rtems_interrupt_lock_release() . . . 176
8.4.12 rtems_interrupt_lock_acquire_isr() . . . 177
8.4.13 rtems_interrupt_lock_release_isr() . . . 178
8.4.14 rtems_interrupt_lock_interrupt_disable() . . . 179
8.4.15 RTEMS_INTERRUPT_LOCK_DECLARE() . . . 180
8.4.16 RTEMS_INTERRUPT_LOCK_DEFINE() . . . 181
8.4.17 RTEMS_INTERRUPT_LOCK_INITIALIZER() . . . 182
8.4.18 RTEMS_INTERRUPT_LOCK_MEMBER() . . . 183
8.4.19 RTEMS_INTERRUPT_LOCK_REFERENCE() . . . 184
8.4.20 RTEMS_INTERRUPT_ENTRY_INITIALIZER() . . . 185
8.4.21 rtems_interrupt_entry_initialize() . . . 186
8.4.22 rtems_interrupt_entry_install() . . . 187
8.4.23 rtems_interrupt_entry_remove() . . . 189
8.4.24 rtems_interrupt_handler_install() . . . 190
8.4.25 rtems_interrupt_handler_remove() . . . 192
8.4.26 rtems_interrupt_vector_is_enabled() . . . 193
8.4.27 rtems_interrupt_vector_enable() . . . 194
8.4.28 rtems_interrupt_vector_disable() . . . 195
8.4.29 rtems_interrupt_is_pending() . . . 196
8.4.30 rtems_interrupt_raise() . . . 198
8.4.31 rtems_interrupt_raise_on() . . . 199
8.4.32 rtems_interrupt_clear() . . . 200
8.4.33 rtems_interrupt_get_priority() . . . 201
8.4.34 rtems_interrupt_set_priority() . . . 202
8.4.35 rtems_interrupt_get_affinity() . . . 204
8.4.36 rtems_interrupt_set_affinity() . . . 205
8.4.37 rtems_interrupt_get_attributes() . . . 207
8.4.38 rtems_interrupt_handler_iterate() . . . 208
8.4.39 rtems_interrupt_server_initialize() . . . 210
8.4.40 rtems_interrupt_server_create() . . . 212
8.4.41 rtems_interrupt_server_handler_install() . . . 213
8.4.42 rtems_interrupt_server_handler_remove() . . . 215
8.4.43 rtems_interrupt_server_set_affinity() . . . 216
8.4.44 rtems_interrupt_server_delete() . . . 218
8.4.45 rtems_interrupt_server_suspend() . . . 219
8.4.46 rtems_interrupt_server_resume() . . . 220
8.4.47 rtems_interrupt_server_move() . . . 221
8.4.48 rtems_interrupt_server_handler_iterate() . . . 222
8.4.49 rtems_interrupt_server_entry_initialize() . . . 224
8.4.50 rtems_interrupt_server_action_prepend() . . . 225
8.4.51 rtems_interrupt_server_entry_destroy() . . . 227
8.4.52 rtems_interrupt_server_entry_submit() . . . 228
8.4.53 rtems_interrupt_server_entry_move() . . . 229
8.4.54 rtems_interrupt_server_request_initialize() . . . 230
8.4.55 rtems_interrupt_server_request_set_vector() . . . 231
8.4.56 rtems_interrupt_server_request_destroy() . . . 232

8.4.57 rtems_interrupt_server_request_submit() . . . 233

**9 Clock Manager** **235**
9.1 Introduction . . . 236
9.2 Background . . . 238
9.2.1 Required Support . . . 238
9.2.2 Time and Date Data Structures . . . 238
9.2.3 Clock Tick and Timeslicing . . . 239
9.2.4 Delays . . . 239
9.2.5 Timeouts . . . 239
9.3 Operations . . . 240
9.3.1 Announcing a Tick . . . 240
9.3.2 Setting the Time . . . 240
9.3.3 Obtaining the Time . . . 240
9.3.4 Transition Advice for the Removed rtems_clock_get() . . . 240
9.4 Directives . . . 242
9.4.1 rtems_clock_set() . . . 243
9.4.2 rtems_clock_get_tod() . . . 245
9.4.3 rtems_clock_get_tod_timeval() . . . 246
9.4.4 rtems_clock_get_realtime() . . . 247
9.4.5 rtems_clock_get_realtime_bintime() . . . 248
9.4.6 rtems_clock_get_realtime_timeval() . . . 249
9.4.7 rtems_clock_get_realtime_coarse() . . . 250
9.4.8 rtems_clock_get_realtime_coarse_bintime() . . . 251
9.4.9 rtems_clock_get_realtime_coarse_timeval() . . . 252
9.4.10 rtems_clock_get_monotonic() . . . 253
9.4.11 rtems_clock_get_monotonic_bintime() . . . 254
9.4.12 rtems_clock_get_monotonic_sbintime() . . . 255
9.4.13 rtems_clock_get_monotonic_timeval() . . . 256
9.4.14 rtems_clock_get_monotonic_coarse() . . . 257
9.4.15 rtems_clock_get_monotonic_coarse_bintime() . . . 258
9.4.16 rtems_clock_get_monotonic_coarse_timeval() . . . 259
9.4.17 rtems_clock_get_boot_time() . . . 260
9.4.18 rtems_clock_get_boot_time_bintime() . . . 261
9.4.19 rtems_clock_get_boot_time_timeval() . . . 262
9.4.20 rtems_clock_get_seconds_since_epoch() . . . 263
9.4.21 rtems_clock_get_ticks_per_second() . . . 264
9.4.22 rtems_clock_get_ticks_since_boot() . . . 265
9.4.23 rtems_clock_get_uptime() . . . 266
9.4.24 rtems_clock_get_uptime_timeval() . . . 267
9.4.25 rtems_clock_get_uptime_seconds() . . . 268
9.4.26 rtems_clock_get_uptime_nanoseconds() . . . 269
9.4.27 rtems_clock_tick_later() . . . 270
9.4.28 rtems_clock_tick_later_usec() . . . 271
9.4.29 rtems_clock_tick_before() . . . 272
9.5 Removed Directives . . . 273
9.5.1 CLOCK_GET - Get date and time information . . . 274

**10 Timer Manager** **275**
10.1 Introduction . . . 276
10.2 Background . . . 277
10.2.1 Required Support . . . 277

10.2.2 Timers . . . 277
10.2.3 Timer Server . . . 277
10.2.4 Timer Service Routines . . . 277
10.3 Operations . . . 278
10.3.1 Creating a Timer . . . 278
10.3.2 Obtaining Timer IDs . . . 278
10.3.3 Initiating an Interval Timer . . . 278
10.3.4 Initiating a Time of Day Timer . . . 278
10.3.5 Canceling a Timer . . . 278
10.3.6 Resetting a Timer . . . 278
10.3.7 Initiating the Timer Server . . . 278
10.3.8 Deleting a Timer . . . 279
10.4 Directives . . . 280
10.4.1 rtems_timer_create() . . . 281
10.4.2 rtems_timer_ident() . . . 283
10.4.3 rtems_timer_cancel() . . . 284
10.4.4 rtems_timer_delete() . . . 285
10.4.5 rtems_timer_fire_after() . . . 286
10.4.6 rtems_timer_fire_when() . . . 288
10.4.7 rtems_timer_initiate_server() . . . 290
10.4.8 rtems_timer_server_fire_after() . . . 292
10.4.9 rtems_timer_server_fire_when() . . . 294
10.4.10 rtems_timer_reset() . . . 296
10.4.11 rtems_timer_get_information() . . . 297

**11 Rate Monotonic Manager 299**
11.1 Introduction . . . 300
11.2 Background . . . 301
11.2.1 Rate Monotonic Manager Required Support . . . 301
11.2.2 Period Statistics . . . 301
11.2.3 Periodicity Definitions . . . 302
11.2.4 Rate Monotonic Scheduling Algorithm . . . 302
11.2.5 Schedulability Analysis . . . 303
11.2.5.1 Assumptions . . . 303
11.2.5.2 Processor Utilization Rule . . . 304
11.2.5.3 Processor Utilization Rule Example . . . 304
11.2.5.4 First Deadline Rule . . . 304
11.2.5.5 First Deadline Rule Example . . . 305
11.2.5.6 Relaxation of Assumptions . . . 306
11.3 Operations . . . 307
11.3.1 Creating a Rate Monotonic Period . . . 307
11.3.2 Manipulating a Period . . . 307
11.3.3 Obtaining the Status of a Period . . . 307
11.3.4 Canceling a Period . . . 307
11.3.5 Deleting a Rate Monotonic Period . . . 308
11.3.6 Examples . . . 308
11.3.7 Simple Periodic Task . . . 308
11.3.8 Task with Multiple Periods . . . 309
11.4 Directives . . . 311
11.4.1 rtems_rate_monotonic_create() . . . 312
11.4.2 rtems_rate_monotonic_ident() . . . 314

11.4.3 rtems_rate_monotonic_cancel() . . . 315
11.4.4 rtems_rate_monotonic_delete() . . . 316
11.4.5 rtems_rate_monotonic_period() . . . 317
11.4.6 rtems_rate_monotonic_get_status() . . . 319
11.4.7 rtems_rate_monotonic_get_statistics() . . . 321
11.4.8 rtems_rate_monotonic_reset_statistics() . . . 323
11.4.9 rtems_rate_monotonic_reset_all_statistics() . . . 324
11.4.10 rtems_rate_monotonic_report_statistics() . . . 325
11.4.11 rtems_rate_monotonic_report_statistics_with_plugin() . . . 326

**12 Semaphore Manager 327**
12.1 Introduction . . . 328
12.2 Background . . . 329
12.2.1 Nested Resource Access . . . 329
12.2.2 Priority Inheritance . . . 329
12.2.3 Priority Ceiling . . . 330
12.2.4 Multiprocessor Resource Sharing Protocol . . . 330
12.2.5 Building a Semaphore Attribute Set . . . 330
12.2.6 Building a SEMAPHORE_OBTAIN Option Set . . . 331
12.3 Operations . . . 332
12.3.1 Creating a Semaphore . . . 332
12.3.2 Obtaining Semaphore IDs . . . 332
12.3.3 Acquiring a Semaphore . . . 332
12.3.4 Releasing a Semaphore . . . 333
12.3.5 Deleting a Semaphore . . . 333
12.4 Directives . . . 334
12.4.1 rtems_semaphore_create() . . . 335
12.4.2 rtems_semaphore_ident() . . . 339
12.4.3 rtems_semaphore_delete() . . . 341
12.4.4 rtems_semaphore_obtain() . . . 343
12.4.5 rtems_semaphore_release() . . . 346
12.4.6 rtems_semaphore_flush() . . . 348
12.4.7 rtems_semaphore_set_priority() . . . 350

**13 Barrier Manager 355**
13.1 Introduction . . . 356
13.2 Background . . . 357
13.2.1 Automatic Versus Manual Barriers . . . 357
13.2.2 Building a Barrier Attribute Set . . . 357
13.3 Directives . . . 358
13.3.1 rtems_barrier_create() . . . 359
13.3.2 rtems_barrier_ident() . . . 361
13.3.3 rtems_barrier_delete() . . . 362
13.3.4 rtems_barrier_wait() . . . 363
13.3.5 rtems_barrier_release() . . . 364

**14 Message Manager 365**
14.1 Introduction . . . 366
14.2 Background . . . 367
14.2.1 Messages . . . 367
14.2.2 Message Queues . . . 367
14.2.3 Building a Message Queue Attribute Set . . . 367

14.2.4 Building a MESSAGE_QUEUE_RECEIVE Option Set . . . 367
14.3 Operations . . . 369
14.3.1 Creating a Message Queue . . . 369
14.3.2 Obtaining Message Queue IDs . . . 369
14.3.3 Receiving a Message . . . 369
14.3.4 Sending a Message . . . 369
14.3.5 Broadcasting a Message . . . 370
14.3.6 Deleting a Message Queue . . . 370
14.4 Directives . . . 371
14.4.1 rtems_message_queue_create() . . . 372
14.4.2 rtems_message_queue_construct() . . . 375
14.4.3 rtems_message_queue_ident() . . . 377
14.4.4 rtems_message_queue_delete() . . . 379
14.4.5 rtems_message_queue_send() . . . 381
14.4.6 rtems_message_queue_urgent() . . . 383
14.4.7 rtems_message_queue_broadcast() . . . 385
14.4.8 rtems_message_queue_receive() . . . 387
14.4.9 rtems_message_queue_get_number_pending() . . . 389
14.4.10 rtems_message_queue_flush() . . . 390
14.4.11 RTEMS_MESSAGE_QUEUE_BUFFER() . . . 391

**15 Event Manager 393**
15.1 Introduction . . . 394
15.2 Background . . . 395
15.2.1 Event Sets . . . 395
15.2.2 Building an Event Set or Condition . . . 395
15.2.3 Building an EVENT_RECEIVE Option Set . . . 395
15.3 Operations . . . 397
15.3.1 Sending an Event Set . . . 397
15.3.2 Receiving an Event Set . . . 397
15.3.3 Determining the Pending Event Set . . . 397
15.3.4 Receiving all Pending Events . . . 397
15.4 Directives . . . 398
15.4.1 rtems_event_send() . . . 399
15.4.2 rtems_event_receive() . . . 401

**16 Signal Manager 403**
16.1 Introduction . . . 404
16.2 Background . . . 405
16.2.1 Signal Manager Definitions . . . 405
16.2.2 A Comparison of ASRs and ISRs . . . 405
16.2.3 Building a Signal Set . . . 405
16.2.4 Building an ASR Mode . . . 405
16.3 Operations . . . 407
16.3.1 Establishing an ASR . . . 407
16.3.2 Sending a Signal Set . . . 407
16.3.3 Processing an ASR . . . 407
16.4 Directives . . . 409
16.4.1 rtems_signal_catch() . . . 410
16.4.2 rtems_signal_send() . . . 412

**17 Partition Manager 415**

17.1 Introduction . . . 416
17.2 Background . . . 417
17.2.1 Partition Manager Definitions . . . 417
17.2.2 Building a Partition Attribute Set . . . 417
17.3 Operations . . . 418
17.3.1 Creating a Partition . . . 418
17.3.2 Obtaining Partition IDs . . . 418
17.3.3 Acquiring a Buffer . . . 418
17.3.4 Releasing a Buffer . . . 418
17.3.5 Deleting a Partition . . . 418
17.4 Directives . . . 419
17.4.1 rtems_partition_create() . . . 420
17.4.2 rtems_partition_ident() . . . 423
17.4.3 rtems_partition_delete() . . . 425
17.4.4 rtems_partition_get_buffer() . . . 427
17.4.5 rtems_partition_return_buffer() . . . 429

**18 Region Manager 431**
18.1 Introduction . . . 432
18.2 Background . . . 433
18.2.1 Region Manager Definitions . . . 433
18.2.2 Building an Attribute Set . . . 433
18.2.3 Building an Option Set . . . 433
18.3 Operations . . . 435
18.3.1 Creating a Region . . . 435
18.3.2 Obtaining Region IDs . . . 435
18.3.3 Adding Memory to a Region . . . 435
18.3.4 Acquiring a Segment . . . 435
18.3.5 Releasing a Segment . . . 435
18.3.6 Obtaining the Size of a Segment . . . 436
18.3.7 Changing the Size of a Segment . . . 436
18.3.8 Deleting a Region . . . 436
18.4 Directives . . . 437
18.4.1 rtems_region_create() . . . 438
18.4.2 rtems_region_ident() . . . 440
18.4.3 rtems_region_delete() . . . 441
18.4.4 rtems_region_extend() . . . 442
18.4.5 rtems_region_get_segment() . . . 444
18.4.6 rtems_region_return_segment() . . . 446
18.4.7 rtems_region_resize_segment() . . . 447
18.4.8 rtems_region_get_information() . . . 449
18.4.9 rtems_region_get_free_information() . . . 450
18.4.10 rtems_region_get_segment_size() . . . 452

**19 Dual-Ported Memory Manager 455**
19.1 Introduction . . . 456
19.2 Background . . . 457
19.3 Operations . . . 458
19.3.1 Creating a Port . . . 458
19.3.2 Obtaining Port IDs . . . 458
19.3.3 Converting an Address . . . 458
19.3.4 Deleting a DPMA Port . . . 458

19.4 Directives . . . 459
19.4.1 rtems_port_create() . . . 460
19.4.2 rtems_port_ident() . . . 462
19.4.3 rtems_port_delete() . . . 463
19.4.4 rtems_port_external_to_internal() . . . 464
19.4.5 rtems_port_internal_to_external() . . . 465

**20 I/O Manager 467**
20.1 Introduction . . . 468
20.2 Background . . . 469
20.2.1 Device Driver Table . . . 469
20.2.2 Major and Minor Device Numbers . . . 469
20.2.3 Device Names . . . 469
20.2.4 Device Driver Environment . . . 469
20.2.5 Runtime Driver Registration . . . 470
20.2.6 Device Driver Interface . . . 470
20.2.7 Device Driver Initialization . . . 470
20.3 Operations . . . 472
20.3.1 Register and Lookup Name . . . 472
20.3.2 Accessing an Device Driver . . . 472
20.4 Directives . . . 473
20.4.1 rtems_io_register_driver() . . . 474
20.4.2 rtems_io_unregister_driver() . . . 476
20.4.3 rtems_io_initialize() . . . 477
20.4.4 rtems_io_register_name() . . . 478
20.4.5 rtems_io_open() . . . 479
20.4.6 rtems_io_close() . . . 480
20.4.7 rtems_io_read() . . . 481
20.4.8 rtems_io_write() . . . 482
20.4.9 rtems_io_control() . . . 483

**21 Kernel Character I/O Support 485**
21.1 Introduction . . . 486
21.2 Directives . . . 487
21.2.1 rtems_putc() . . . 488
21.2.2 rtems_put_char() . . . 489
21.2.3 putk() . . . 490
21.2.4 printk() . . . 491
21.2.5 vprintk() . . . 492
21.2.6 rtems_printk_printer() . . . 493
21.2.7 getchark() . . . 494

**22 Cache Manager 495**
22.1 Introduction . . . 496
22.2 Directives . . . 497
22.2.1 rtems_cache_flush_multiple_data_lines() . . . 498
22.2.2 rtems_cache_invalidate_multiple_data_lines() . . . 499
22.2.3 rtems_cache_invalidate_multiple_instruction_lines() . . . 500
22.2.4 rtems_cache_instruction_sync_after_code_change() . . . 501
22.2.5 rtems_cache_get_maximal_line_size() . . . 502
22.2.6 rtems_cache_get_data_line_size() . . . 503
22.2.7 rtems_cache_get_instruction_line_size() . . . 504

22.2.8 rtems_cache_get_data_cache_size() . . . 505
22.2.9 rtems_cache_get_instruction_cache_size() . . . 506
22.2.10 rtems_cache_flush_entire_data() . . . 507
22.2.11 rtems_cache_invalidate_entire_data() . . . 508
22.2.12 rtems_cache_invalidate_entire_instruction() . . . 509
22.2.13 rtems_cache_enable_data() . . . 510
22.2.14 rtems_cache_disable_data() . . . 511
22.2.15 rtems_cache_enable_instruction() . . . 512
22.2.16 rtems_cache_disable_instruction() . . . 513
22.2.17 rtems_cache_aligned_malloc() . . . 514

**23 Fatal Error Manager 515**
23.1 Introduction . . . 516
23.2 Background . . . 517
23.2.1 Overview . . . 517
23.2.2 System Termination Procedure . . . 517
23.2.3 Fatal Sources . . . 518
23.2.4 Internal Error Codes . . . 519
23.3 Operations . . . 524
23.3.1 Announcing a Fatal Error . . . 524
23.4 Directives . . . 525
23.4.1 rtems_fatal() . . . 526
23.4.2 rtems_panic() . . . 527
23.4.3 rtems_shutdown_executive() . . . 528
23.4.4 rtems_exception_frame_print() . . . 529
23.4.5 rtems_fatal_source_text() . . . 530
23.4.6 rtems_internal_error_text() . . . 531
23.4.7 rtems_fatal_error_occurred() . . . 532

**24 Board Support Packages 533**
24.1 Introduction . . . 534
24.2 Reset and Initialization . . . 535
24.2.1 Interrupt Stack Requirements . . . 536
24.2.2 Processors with a Separate Interrupt Stack . . . 536
24.2.3 Processors Without a Separate Interrupt Stack . . . 536
24.3 Device Drivers . . . 537
24.3.1 Clock Tick Device Driver . . . 537
24.4 User Extensions . . . 538
24.5 Multiprocessor Communications Interface (MPCI) . . . 539
24.5.1 Tightly-Coupled Systems . . . 539
24.5.2 Loosely-Coupled Systems . . . 539
24.5.3 Systems with Mixed Coupling . . . 539
24.5.4 Heterogeneous Systems . . . 539

**25 User Extensions Manager 541**
25.1 Introduction . . . 542
25.2 Background . . . 543
25.2.1 Extension Sets . . . 543
25.2.2 TCB Extension Area . . . 543
25.2.3 Order of Invocation . . . 544
25.2.4 Thread Create Extension . . . 545
25.2.5 Thread Start Extension . . . 545

25.2.6 Thread Restart Extension . . . 545
25.2.7 Thread Switch Extension . . . 546
25.2.8 Thread Begin Extension . . . 546
25.2.9 Thread Exitted Extension . . . 547
25.2.10 Thread Termination Extension . . . 547
25.2.11 Thread Delete Extension . . . 547
25.2.12 Fatal Error Extension . . . 548
25.3 Directives . . . 549
25.3.1 rtems_extension_create() . . . 550
25.3.2 rtems_extension_delete() . . . 552
25.3.3 rtems_extension_ident() . . . 553

**26 Configuring a System 555**
26.1 Introduction . . . 556
26.2 Default Value Selection Philosophy . . . 559
26.3 Sizing the RTEMS Workspace . . . 560
26.4 Potential Issues with RTEMS Workspace Size Estimation . . . 561
26.5 Configuration Example . . . 562
26.6 Unlimited Objects . . . 564
26.6.1 Unlimited Objects by Class . . . 565
26.6.2 Unlimited Objects by Default . . . 565
26.7 General System Configuration . . . 566
26.7.1 CONFIGURE_DIRTY_MEMORY . . . 567
26.7.2 CONFIGURE_DISABLE_BSP_SETTINGS . . . 568
26.7.3 CONFIGURE_DISABLE_NEWLIB_REENTRANCY . . . 569
26.7.4 CONFIGURE_EXECUTIVE_RAM_SIZE . . . 570
26.7.5 CONFIGURE_EXTRA_TASK_STACKS . . . 571
26.7.6 CONFIGURE_INIT . . . 572
26.7.7 CONFIGURE_INITIAL_EXTENSIONS . . . 573
26.7.8 CONFIGURE_INTERRUPT_STACK_SIZE . . . 574
26.7.9 CONFIGURE_MALLOC_DIRTY . . . 575
26.7.10 CONFIGURE_MAXIMUM_FILE_DESCRIPTORS . . . 576
26.7.11 CONFIGURE_MAXIMUM_PROCESSORS . . . 577
26.7.12 CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE . . . 578
26.7.13 CONFIGURE_MAXIMUM_THREAD_NAME_SIZE . . . 579
26.7.14 CONFIGURE_MEMORY_OVERHEAD . . . 580
26.7.15 CONFIGURE_MESSAGE_BUFFER_MEMORY . . . 581
26.7.16 CONFIGURE_MICROSECONDS_PER_TICK . . . 583
26.7.17 CONFIGURE_MINIMUM_TASK_STACK_SIZE . . . 584
26.7.18 CONFIGURE_STACK_CHECKER_ENABLED . . . 585
26.7.19 CONFIGURE_TICKS_PER_TIMESLICE . . . 586
26.7.20 CONFIGURE_UNIFIED_WORK_AREAS . . . 587
26.7.21 CONFIGURE_UNLIMITED_ALLOCATION_SIZE . . . 588
26.7.22 CONFIGURE_UNLIMITED_OBJECTS . . . 589
26.7.23 CONFIGURE_VERBOSE_SYSTEM_INITIALIZATION . . . 590
26.7.24 CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY . . . 591
26.8 Device Driver Configuration . . . 592
26.8.1 CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER . . . 593
26.8.2 CONFIGURE_APPLICATION_EXTRA_DRIVERS . . . 594
26.8.3 CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER . . . 595
26.8.4 CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER . . . 596

26.8.5 CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER . . . . . 597
26.8.6 CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER . . . . . 598
26.8.7 CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER . . . . . 599
26.8.8 CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER . . . . . 600
26.8.9 CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER . . . . . 601
26.8.10 CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER . . . . 602
26.8.11 CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER 603
26.8.12 CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER . . . . . 604
26.8.13 CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER . . . . . 605
26.8.14 CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER . . . . . 606
26.8.15 CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER . . . . . 607
26.8.16 CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS . . . . . 608
26.8.17 CONFIGURE_ATA_DRIVER_TASK_PRIORITY . . . . . 609
26.8.18 CONFIGURE_EXCEPTION_TO_SIGNAL_MAPPING . . . . . 610
26.8.19 CONFIGURE_MAXIMUM_DRIVERS . . . . . 611
26.9 Classic API Configuration . . . . . 613
26.9.1 CONFIGURE_MAXIMUM_BARRIERS . . . . . 614
26.9.2 CONFIGURE_MAXIMUM_MESSAGE_QUEUES . . . . . 615
26.9.3 CONFIGURE_MAXIMUM_PARTITIONS . . . . . 616
26.9.4 CONFIGURE_MAXIMUM_PERIODS . . . . . 617
26.9.5 CONFIGURE_MAXIMUM_PORTS . . . . . 618
26.9.6 CONFIGURE_MAXIMUM_REGIONS . . . . . 619
26.9.7 CONFIGURE_MAXIMUM_SEMAPHORES . . . . . 620
26.9.8 CONFIGURE_MAXIMUM_TASKS . . . . . 621
26.9.9 CONFIGURE_MAXIMUM_TIMERS . . . . . 622
26.9.10 CONFIGURE_MAXIMUM_USER_EXTENSIONS . . . . . 623
26.9.11 CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE . . 624
26.10 Classic API Initialization Task Configuration . . . . . 625
26.10.1 CONFIGURE_INIT_TASK_ARGUMENTS . . . . . 626
26.10.2 CONFIGURE_INIT_TASK_ATTRIBUTES . . . . . 627
26.10.3 CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE . . . . . 628
26.10.4 CONFIGURE_INIT_TASK_ENTRY_POINT . . . . . 629
26.10.5 CONFIGURE_INIT_TASK_INITIAL_MODES . . . . . 630
26.10.6 CONFIGURE_INIT_TASK_NAME . . . . . 631
26.10.7 CONFIGURE_INIT_TASK_PRIORITY . . . . . 632
26.10.8 CONFIGURE_INIT_TASK_STACK_SIZE . . . . . 633
26.10.9 CONFIGURE_RTEMS_INIT_TASKS_TABLE . . . . . 634
26.11 POSIX API Configuration . . . . . 635
26.11.1 CONFIGURE_MAXIMUM_POSIX_KEYS . . . . . 636
26.11.2 CONFIGURE_MAXIMUM_POSIX_KEY_VALUE_PAIRS . . . . . 637
26.11.3 CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES . . . . . 638
26.11.4 CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS . . . . . 639
26.11.5 CONFIGURE_MAXIMUM_POSIX_SEMAPHORES . . . . . 640
26.11.6 CONFIGURE_MAXIMUM_POSIX_SHMS . . . . . 641
26.11.7 CONFIGURE_MAXIMUM_POSIX_THREADS . . . . . 642
26.11.8 CONFIGURE_MAXIMUM_POSIX_TIMERS . . . . . 643
26.11.9 CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE . . . . . 644
26.12 POSIX Initialization Thread Configuration . . . . . 645
26.12.1 CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT . . . . . 646
26.12.2 CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE . . . . . 647
26.12.3 CONFIGURE_POSIX_INIT_THREAD_TABLE . . . . . 648

26.13 Event Recording Configuration . . . 649
26.13.1 CONFIGURE_RECORD_EXTENSIONS_ENABLED . . . 650
26.13.2 CONFIGURE_RECORD_FATAL_DUMP_BASE64 . . . 651
26.13.3 CONFIGURE_RECORD_FATAL_DUMP_BASE64_ZLIB . . . 652
26.13.4 CONFIGURE_RECORD_INTERRUPTS_ENABLED . . . 653
26.13.5 CONFIGURE_RECORD_PER_PROCESSOR_ITEMS . . . 654
26.14 Filesystem Configuration . . . 655
26.14.1 CONFIGURE_APPLICATION_DISABLE_FILESYSTEM . . . 656
26.14.2 CONFIGURE_FILESYSTEM_ALL . . . 657
26.14.3 CONFIGURE_FILESYSTEM_DOSFS . . . 658
26.14.4 CONFIGURE_FILESYSTEM_FTPFS . . . 659
26.14.5 CONFIGURE_FILESYSTEM_IMFS . . . 660
26.14.6 CONFIGURE_FILESYSTEM_JFFS2 . . . 661
26.14.7 CONFIGURE_FILESYSTEM_NFS . . . 662
26.14.8 CONFIGURE_FILESYSTEM_RFS . . . 663
26.14.9 CONFIGURE_FILESYSTEM_TFTPFS . . . 664
26.14.10 CONFIGURE_IMFS_DISABLE_CHMOD . . . 665
26.14.11 CONFIGURE_IMFS_DISABLE_CHOWN . . . 666
26.14.12 CONFIGURE_IMFS_DISABLE_LINK . . . 667
26.14.13 CONFIGURE_IMFS_DISABLE_MKNOD . . . 668
26.14.14 CONFIGURE_IMFS_DISABLE_MKNOD_DEVICE . . . 669
26.14.15 CONFIGURE_IMFS_DISABLE_MKNOD_FILE . . . 670
26.14.16 CONFIGURE_IMFS_DISABLE_MOUNT . . . 671
26.14.17 CONFIGURE_IMFS_DISABLE_READDIR . . . 672
26.14.18 CONFIGURE_IMFS_DISABLE_READLINK . . . 673
26.14.19 CONFIGURE_IMFS_DISABLE_RENAME . . . 674
26.14.20 CONFIGURE_IMFS_DISABLE_RMNOD . . . 675
26.14.21 CONFIGURE_IMFS_DISABLE_SYMLINK . . . 676
26.14.22 CONFIGURE_IMFS_DISABLE_UNMOUNT . . . 677
26.14.23 CONFIGURE_IMFS_DISABLE_UTIME . . . 678
26.14.24 CONFIGURE_IMFS_ENABLE_MKFIFO . . . 679
26.14.25 CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK . . . 680
26.14.26 CONFIGURE_JFFS2_DELAYED_WRITE_TASK_PRIORITY . . . 681
26.14.27 CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM . . . 682
26.14.28 CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM . . . 683
26.15 Block Device Cache Configuration . . . 684
26.15.1 CONFIGURE_APPLICATION_NEEDS_LIBBLOCK . . . 685
26.15.2 CONFIGURE_BDBUF_BUFFER_MAX_SIZE . . . 686
26.15.3 CONFIGURE_BDBUF_BUFFER_MIN_SIZE . . . 687
26.15.4 CONFIGURE_BDBUF_CACHE_MEMORY_SIZE . . . 688
26.15.5 CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS . . . 689
26.15.6 CONFIGURE_BDBUF_MAX_WRITE_BLOCKS . . . 690
26.15.7 CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY . . . 691
26.15.8 CONFIGURE_BDBUF_TASK_STACK_SIZE . . . 692
26.15.9 CONFIGURE_SWAPOUT_BLOCK_HOLD . . . 693
26.15.10 CONFIGURE_SWAPOUT_SWAP_PERIOD . . . 694
26.15.11 CONFIGURE_SWAPOUT_TASK_PRIORITY . . . 695
26.15.12 CONFIGURE_SWAPOUT_WORKER_TASKS . . . 696
26.15.13 CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY . . . 697
26.16 Task Stack Allocator Configuration . . . 698
26.16.1 CONFIGURE_TASK_STACK_ALLOCATOR . . . 699

26.16.2 CONFIGURE_TASK_STACK_ALLOCATOR_AVOIDS_WORK_SPACE . . . . 700
26.16.3 CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE . . . . . 701
26.16.4 CONFIGURE_TASK_STACK_ALLOCATOR_INIT . . . . . 702
26.16.5 CONFIGURE_TASK_STACK_DEALLOCATOR . . . . . 703
26.16.6 CONFIGURE_TASK_STACK_FROM_ALLOCATOR . . . . . 704
26.17 Idle Task Configuration . . . . . 705
26.17.1 CONFIGURE_IDLE_TASK_BODY . . . . . 706
26.17.2 CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION . . . . . 707
26.17.3 CONFIGURE_IDLE_TASK_STACK_SIZE . . . . . 708
26.17.4 CONFIGURE_IDLE_TASK_STORAGE_SIZE . . . . . 709
26.18 General Scheduler Configuration . . . . . 710
26.18.1 CONFIGURE_CBS_MAXIMUM_SERVERS . . . . . 711
26.18.2 CONFIGURE_MAXIMUM_PRIORITY . . . . . 712
26.18.3 CONFIGURE_SCHEDULER_ASSIGNMENTS . . . . . 713
26.18.4 CONFIGURE_SCHEDULER_CBS . . . . . 714
26.18.5 CONFIGURE_SCHEDULER_EDF . . . . . 715
26.18.6 CONFIGURE_SCHEDULER_EDF_SMP . . . . . 716
26.18.7 CONFIGURE_SCHEDULER_NAME . . . . . 717
26.18.8 CONFIGURE_SCHEDULER_PRIORITY . . . . . 718
26.18.9 CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP . . . . . 719
26.18.10 CONFIGURE_SCHEDULER_PRIORITY_SMP . . . . . 720
26.18.11 CONFIGURE_SCHEDULER_SIMPLE . . . . . 721
26.18.12 CONFIGURE_SCHEDULER_SIMPLE_SMP . . . . . 722
26.18.13 CONFIGURE_SCHEDULER_STRONG_APA . . . . . 723
26.18.14 CONFIGURE_SCHEDULER_TABLE_ENTRIES . . . . . 724
26.18.15 CONFIGURE_SCHEDULER_USER . . . . . 725
26.19 Clustered Scheduler Configuration . . . . . 726
26.19.1 Configuration Step 1 - Scheduler Algorithms . . . . . 726
26.19.2 Configuration Step 2 - Schedulers . . . . . 726
26.19.3 Configuration Step 3 - Scheduler Table . . . . . 727
26.19.4 Configuration Step 4 - Processor to Scheduler Assignment . . . . . 727
26.19.5 Configuration Example . . . . . 727
26.19.6 Configuration Errors . . . . . 728
26.20 FACE Technical Standard Related Configuration . . . . . 730
26.20.1 CONFIGURE_POSIX_TIMERS_FACE_BEHAVIOR . . . . . 731
26.21 Multiprocessing Configuration . . . . . 732
26.21.1 CONFIGURE_EXTRA_MPCI_RECEIVE_SERVER_STACK . . . . . 733
26.21.2 CONFIGURE_MP_APPLICATION . . . . . 734
26.21.3 CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS . . . . . 735
26.21.4 CONFIGURE_MP_MAXIMUM_NODES . . . . . 736
26.21.5 CONFIGURE_MP_MAXIMUM_PROXIES . . . . . 737
26.21.6 CONFIGURE_MP_MPCI_TABLE_POINTER . . . . . 738
26.21.7 CONFIGURE_MP_NODE_NUMBER . . . . . 739
26.22 PCI Library Configuration . . . . . 740
26.23 Ada Configuration . . . . . 741
26.24 Directives . . . . . 742
26.24.1 rtems_get_build_label() . . . . . 743
26.24.2 rtems_get_copyright_notice() . . . . . 744
26.24.3 rtems_get_target_hash() . . . . . 745
26.24.4 rtems_get_version_string() . . . . . 746
26.24.5 rtems_configuration_get_do_zero_of_workspace() . . . . . 747

26.24.6 rtems_configuration_get_idle_task_stack_size() . . . . . 748
26.24.7 rtems_configuration_get_idle_task() . . . . . 749
26.24.8 rtems_configuration_get_interrupt_stack_size() . . . . . 750
26.24.9 rtems_configuration_get_maximum_barriers() . . . . . 751
26.24.10 rtems_configuration_get_maximum_extensions() . . . . . 752
26.24.11 rtems_configuration_get_maximum_message_queues() . . . . . 753
26.24.12 rtems_configuration_get_maximum_partitions() . . . . . 754
26.24.13 rtems_configuration_get_maximum_periods() . . . . . 755
26.24.14 rtems_configuration_get_maximum_ports() . . . . . 756
26.24.15 rtems_configuration_get_maximum_processors() . . . . . 757
26.24.16 rtems_configuration_get_maximum_regions() . . . . . 758
26.24.17 rtems_configuration_get_maximum_semaphores() . . . . . 759
26.24.18 rtems_configuration_get_maximum_tasks() . . . . . 760
26.24.19 rtems_configuration_get_maximum_timers() . . . . . 761
26.24.20 rtems_configuration_get_microseconds_per_tick() . . . . . 762
26.24.21 rtems_configuration_get_milliseconds_per_tick() . . . . . 763
26.24.22 rtems_configuration_get_nanoseconds_per_tick() . . . . . 764
26.24.23 rtems_configuration_get_number_of_initial_extensions() . . . . . 765
26.24.24 rtems_configuration_get_stack_allocate_for_idle_hook() . . . . . 766
26.24.25 rtems_configuration_get_stack_allocate_hook() . . . . . 767
26.24.26 rtems_configuration_get_stack_allocate_init_hook() . . . . . 768
26.24.27 rtems_configuration_get_stack_allocator_avoids_work_space() . . . . . 769
26.24.28 rtems_configuration_get_stack_free_hook() . . . . . 770
26.24.29 rtems_configuration_get_stack_space_size() . . . . . 771
26.24.30 rtems_configuration_get_ticks_per_timeslice() . . . . . 772
26.24.31 rtems_configuration_get_unified_work_area() . . . . . 773
26.24.32 rtems_configuration_get_user_extension_table() . . . . . 774
26.24.33 rtems_configuration_get_user_multiprocessing_table() . . . . . 775
26.24.34 rtems_configuration_get_work_space_size() . . . . . 776
26.24.35 rtems_configuration_get_rtems_api_configuration() . . . . . 777
26.24.36 rtems_resource_is_unlimited() . . . . . 778
26.24.37 rtems_resource_maximum_per_allocation() . . . . . 779
26.24.38 rtems_resource_unlimited() . . . . . 780
26.25 Obsolete Configuration Options . . . . . 781
26.25.1 CONFIGURE_BDBUF_BUFFER_COUNT . . . . . 781
26.25.2 CONFIGURE_BDBUF_BUFFER_SIZE . . . . . 781
26.25.3 CONFIGURE_DISABLE_CLASSIC_API_NOTEPADS . . . . . 781
26.25.4 CONFIGURE_ENABLE_GO . . . . . 781
26.25.5 CONFIGURE_GNAT_RTEMS . . . . . 781
26.25.6 CONFIGURE_HAS_OWN_CONFIGURATION_TABLE . . . . . 781
26.25.7 CONFIGURE_HAS_OWN_BDBUF_TABLE . . . . . 781
26.25.8 CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE . . . . . 781
26.25.9 CONFIGURE_HAS_OWN_INIT_TASK_TABLE . . . . . 781
26.25.10 CONFIGURE_HAS_OWN_MOUNT_TABLE . . . . . 781
26.25.11 CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE . . . . . 781
26.25.12 CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS . . . . . 782
26.25.13 CONFIGURE_MAXIMUM_ADA_TASKS . . . . . 782
26.25.14 CONFIGURE_MAXIMUM_DEVICES . . . . . 782
26.25.15 CONFIGURE_MAXIMUM_FAKE_ADA_TASKS . . . . . 782
26.25.16 CONFIGURE_MAXIMUM_GO_CHANNELS . . . . . 782
26.25.17 CONFIGURE_MAXIMUM_GOROUTINES . . . . . 782

26.25.18 CONFIGURE_MAXIMUM_MRSP_SEMAPHORES . . . 782
26.25.19 CONFIGURE_NUMBER_OF_TERMIOS_PORTS . . . 782
26.25.20 CONFIGURE_MAXIMUM_POSIX_BARRIERS . . . 782
26.25.21 CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES . . . 782
26.25.22 CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTORS . . . 782
26.25.23 CONFIGURE_MAXIMUM_POSIX_MUTEXES . . . 782
26.25.24 CONFIGURE_MAXIMUM_POSIX_RWLOCKS . . . 783
26.25.25 CONFIGURE_MAXIMUM_POSIX_SPINLOCKS . . . 783
26.25.26 CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE . . . 783
26.25.27 CONFIGURE_SMP_APPLICATION . . . 783
26.25.28 CONFIGURE_SMP_MAXIMUM_PROCESSORS . . . 783
26.25.29 CONFIGURE_TERMIOS_DISABLED . . . 783

**27 Self-Contained Objects 785**
27.1 Introduction . . . 786
27.2 RTEMS Thread API . . . 788
27.3 Mutual Exclusion . . . 789
27.3.1 Static mutex initialization . . . 790
27.3.2 Run-time mutex initialization . . . 791
27.3.3 Lock the mutex . . . 792
27.3.4 Try to lock the mutex . . . 793
27.3.5 Unlock the mutex . . . 794
27.3.6 Set mutex name . . . 795
27.3.7 Get mutex name . . . 796
27.3.8 Mutex destruction . . . 796
27.4 Condition Variables . . . 797
27.4.1 Static condition variable initialization . . . 798
27.4.2 Run-time condition variable initialization . . . 799
27.4.3 Wait for condition signal . . . 800
27.4.4 Signals a condition change . . . 801
27.4.5 Broadcasts a condition change . . . 802
27.4.6 Set condition variable name . . . 803
27.4.7 Get condition variable name . . . 804
27.4.8 Condition variable destruction . . . 804
27.5 Counting Semaphores . . . 805
27.5.1 Static counting semaphore initialization . . . 806
27.5.2 Run-time counting semaphore initialization . . . 807
27.5.3 Wait for a counting semaphore . . . 808
27.5.4 Wait for a counting semaphore with timeout in ticks . . . 809
27.5.5 Tries to wait for a counting semaphore . . . 810
27.5.6 Post a counting semaphore . . . 811
27.5.7 Set counting semaphore name . . . 812
27.5.8 Get counting semaphore name . . . 813
27.5.9 Counting semaphore destruction . . . 813
27.6 Binary Semaphores . . . 814
27.6.1 Static binary semaphore initialization . . . 815
27.6.2 Run-time binary semaphore initialization . . . 816
27.6.3 Wait for a binary semaphore . . . 817
27.6.4 Wait for a binary semaphore with timeout in ticks . . . 818
27.6.5 Tries to wait for a binary semaphore . . . 819
27.6.6 Post a binary semaphore . . . 820

27.6.7 Set binary semaphore name . . . 821
27.6.8 Get binary semaphore name . . . 822
27.6.9 Binary semaphore destruction . . . 822
27.7 Threads . . . 823

**28 Regulator Manager** **825**
28.1 Introduction . . . 826
28.2 Background . . . 827
28.2.1 Regulator Buffering . . . 827
28.2.2 Message Delivery Rate . . . 827
28.3 Operations . . . 829
28.3.1 Application Sourcing Data . . . 829
28.3.2 Delivery Function . . . 829
28.4 Directives . . . 831
28.4.1 rtems_regulator_create() . . . 832
28.4.2 rtems_regulator_delete() . . . 834
28.4.3 rtems_regulator_obtain_buffer() . . . 836
28.4.4 rtems_regulator_release_buffer() . . . 837
28.4.5 rtems_regulator_send() . . . 838
28.4.6 rtems_regulator_get_statistics() . . . 840

**29 Multiprocessing Manager** **843**
29.1 Introduction . . . 844
29.2 Background . . . 845
29.2.1 Nodes . . . 845
29.2.2 Global Objects . . . 845
29.2.3 Global Object Table . . . 845
29.2.4 Remote Operations . . . 846
29.2.5 Proxies . . . 847
29.2.6 Multiprocessor Configuration Table . . . 847
29.3 Multiprocessor Communications Interface Layer . . . 848
29.3.1 INITIALIZATION . . . 848
29.3.2 GET_PACKET . . . 849
29.3.3 RETURN_PACKET . . . 849
29.3.4 RECEIVE_PACKET . . . 849
29.3.5 SEND_PACKET . . . 850
29.3.6 Supporting Heterogeneous Environments . . . 850
29.4 Operations . . . 852
29.4.1 Announcing a Packet . . . 852
29.5 Directives . . . 853
29.5.1 rtems_multiprocessing_announce() . . . 854

**30 Symmetric Multiprocessing (SMP)** **855**
30.1 Introduction . . . 856
30.2 Background . . . 857
30.2.1 Application Configuration . . . 857
30.2.2 Examples . . . 857
30.2.3 Uniprocessor versus SMP Parallelism . . . 857
30.2.4 Task Affinity . . . 858
30.2.5 Task Migration . . . 858
30.2.6 Clustered Scheduling . . . 858
30.2.7 OpenMP . . . 859

30.2.8 Atomic Operations . . . 860
30.3 Application Issues . . . 861
30.3.1 Task variables . . . 861
30.3.2 Highest Priority Thread Never Walks Alone . . . 861
30.3.3 Disabling of Thread Preemption . . . 861
30.3.4 Disabling of Interrupts . . . 862
30.3.5 Interrupt Service Routines Execute in Parallel With Threads . . . 863
30.3.6 Timers Do Not Stop Immediately . . . 863
30.3.7 False Sharing of Cache Lines Due to Objects Table . . . 863
30.4 Implementation Details . . . 864
30.4.1 Low-Level Synchronization . . . 864
30.4.2 Internal Locking . . . 865
30.4.3 Profiling . . . 866
30.4.4 Scheduler Helping Protocol . . . 867
30.4.5 Thread Dispatch Details . . . 867
30.4.6 Per-Processor Data . . . 868
30.4.7 Thread Pinning . . . 868

**31 PCI Library 871**
31.1 Introduction . . . 872
31.2 Background . . . 873
31.2.1 Software Components . . . 873
31.2.2 PCI Configuration . . . 874
31.2.2.1 RTEMS Configuration selection . . . 874
31.2.2.2 Auto Configuration . . . 874
31.2.2.3 Read Configuration . . . 875
31.2.2.4 Static Configuration . . . 875
31.2.2.5 Peripheral Configuration . . . 876
31.2.3 PCI Access . . . 876
31.2.3.1 Configuration space . . . 876
31.2.3.2 I/O space . . . 877
31.2.3.3 Registers over Memory space . . . 877
31.2.3.4 Access functions . . . 877
31.2.3.5 PCI address translation . . . 878
31.2.4 PCI Interrupt . . . 878
31.2.5 PCI Shell command . . . 879

**32 Stack Bounds Checker 881**
32.1 Introduction . . . 882
32.2 Background . . . 883
32.2.1 Task Stack . . . 883
32.2.2 Execution . . . 883
32.3 Operations . . . 884
32.3.1 Initializing the Stack Bounds Checker . . . 884
32.3.2 Checking for Blown Task Stack . . . 884
32.3.3 Reporting Task Stack Usage . . . 884
32.3.4 When a Task Overflows the Stack . . . 884
32.4 Routines . . . 885
32.4.1 STACK_CHECKER_IS_BLOWN - Has Current Task Blown Its Stack . . . 885
32.4.2 STACK_CHECKER_REPORT_USAGE - Report Task Stack Usage . . . 885

**33 CPU Usage Statistics 887**

33.1 Introduction . . . 888
33.2 Background . . . 889
33.3 Operations . . . 890
33.3.1 Report CPU Usage Statistics . . . 890
33.3.2 Reset CPU Usage Statistics . . . 890
33.4 Directives . . . 891
33.4.1 cpu_usage_report - Report CPU Usage Statistics . . . 892
33.4.2 cpu_usage_reset - Reset CPU Usage Statistics . . . 893

**34 Object Services** **895**
34.1 Introduction . . . 896
34.2 Background . . . 897
34.2.1 APIs . . . 897
34.2.2 Object Classes . . . 897
34.2.3 Object Names . . . 897
34.3 Operations . . . 898
34.3.1 Decomposing and Recomposing an Object Id . . . 898
34.3.2 Printing an Object Id . . . 898
34.4 Directives . . . 900
34.4.1 rtems_build_id() . . . 901
34.4.2 rtems_build_name() . . . 902
34.4.3 rtems_object_get_classic_name() . . . 903
34.4.4 rtems_object_get_name() . . . 904
34.4.5 rtems_object_set_name() . . . 905
34.4.6 rtems_object_id_get_api() . . . 906
34.4.7 rtems_object_id_get_class() . . . 907
34.4.8 rtems_object_id_get_node() . . . 908
34.4.9 rtems_object_id_get_index() . . . 909
34.4.10 rtems_object_id_api_minimum() . . . 910
34.4.11 rtems_object_id_api_maximum() . . . 911
34.4.12 rtems_object_api_minimum_class() . . . 912
34.4.13 rtems_object_api_maximum_class() . . . 913
34.4.14 rtems_object_get_api_name() . . . 914
34.4.15 rtems_object_get_api_class_name() . . . 915
34.4.16 rtems_object_get_class_information() . . . 916
34.4.17 rtems_object_get_local_node() . . . 917
34.4.18 RTEMS_OBJECT_ID_INITIAL() . . . 918

**35 Chains** **919**
35.1 Introduction . . . 920
35.2 Background . . . 921
35.2.1 Nodes . . . 921
35.2.2 Controls . . . 921
35.3 Operations . . . 922
35.3.1 Multi-threading . . . 922
35.3.2 Creating a Chain . . . 922
35.3.3 Iterating a Chain . . . 922
35.4 Directives . . . 924
35.4.1 Initialize Chain With Nodes . . . 925
35.4.2 Initialize Empty . . . 926
35.4.3 Is Null Node ? . . . 927
35.4.4 Head . . . 928

35.4.5 Tail . . . 929
35.4.6 Are Two Nodes Equal ? . . . 930
35.4.7 Is the Chain Empty . . . 931
35.4.8 Is this the First Node on the Chain ? . . . 932
35.4.9 Is this the Last Node on the Chain ? . . . 933
35.4.10 Does this Chain have only One Node ? . . . 934
35.4.11 Returns the node count of the chain (unprotected) . . . 935
35.4.12 Is this Node the Chain Head ? . . . 936
35.4.13 Is this Node the Chain Tail ? . . . 937
35.4.14 Extract a Node . . . 938
35.4.15 Extract a Node (unprotected) . . . 939
35.4.16 Get the First Node . . . 940
35.4.17 Get the First Node (unprotected) . . . 941
35.4.18 Insert a Node . . . 942
35.4.19 Insert a Node (unprotected) . . . 943
35.4.20 Append a Node . . . 944
35.4.21 Append a Node (unprotected) . . . 945
35.4.22 Prepend a Node . . . 946
35.4.23 Prepend a Node (unprotected) . . . 947

**36 Red-Black Trees 949**
36.1 Introduction . . . 950
36.2 Background . . . 951
36.2.1 Nodes . . . 951
36.2.2 Controls . . . 951
36.3 Operations . . . 952
36.4 Directives . . . 953
36.4.1 Documentation for the Red-Black Tree Directives . . . 953

**37 Timespec Helpers 955**
37.1 Introduction . . . 956
37.2 Background . . . 957
37.2.1 Time Storage Conventions . . . 957
37.3 Operations . . . 958
37.3.1 Set and Obtain Timespec Value . . . 958
37.3.2 Timespec Math . . . 958
37.3.3 Comparing struct timespec Instances . . . 958
37.3.4 Conversions and Validity Check . . . 958
37.4 Directives . . . 959
37.4.1 TIMESPEC_SET - Set struct timespec Instance . . . 960
37.4.2 TIMESPEC_ZERO - Zero struct timespec Instance . . . 961
37.4.3 TIMESPEC_IS_VALID - Check validity of a struct timespec instance . . . 962
37.4.4 TIMESPEC_ADD_TO - Add Two struct timespec Instances . . . 963
37.4.5 TIMESPEC_SUBTRACT - Subtract Two struct timespec Instances . . . 964
37.4.6 TIMESPEC_DIVIDE - Divide Two struct timespec Instances . . . 965
37.4.7 TIMESPEC_DIVIDE_BY_INTEGER - Divide a struct timespec Instance by an Integer . . . 966
37.4.8 TIMESPEC_LESS_THAN - Less than operator . . . 967
37.4.9 TIMESPEC_GREATER_THAN - Greater than operator . . . 968
37.4.10 TIMESPEC_EQUAL_TO - Check equality of timespecs . . . 969
37.4.11 TIMESPEC_GET_SECONDS - Get Seconds Portion of struct timespec Instance . . . 970

37.4.12 TIMESPEC_GET_NANOSECONDS - Get Nanoseconds Portion of the struct timespec Instance . . . 971
37.4.13 TIMESPEC_TO_TICKS - Convert struct timespec Instance to Ticks . . . 972
37.4.14 TIMESPEC_FROM_TICKS - Convert Ticks to struct timespec Representation . . . 973

**38 Constant Bandwidth Server Scheduler API 975**
38.1 Introduction . . . 976
38.2 Background . . . 977
38.2.1 Constant Bandwidth Server Definitions . . . 977
38.2.2 Handling Periodic Tasks . . . 977
38.2.3 Registering a Callback Function . . . 977
38.2.4 Limitations . . . 978
38.3 Operations . . . 979
38.3.1 Setting up a server . . . 979
38.3.2 Attaching Task to a Server . . . 979
38.3.3 Detaching Task from a Server . . . 979
38.3.4 Examples . . . 979
38.4 Directives . . . 981
38.4.1 CBS_INITIALIZE - Initialize the CBS library . . . 982
38.4.2 CBS_CLEANUP - Cleanup the CBS library . . . 983
38.4.3 CBS_CREATE_SERVER - Create a new bandwidth server . . . 984
38.4.4 CBS_ATTACH_THREAD - Attach a thread to server . . . 985
38.4.5 CBS_DETACH_THREAD - Detach a thread from server . . . 986
38.4.6 CBS_DESTROY_SERVER - Destroy a bandwidth server . . . 987
38.4.7 CBS_GET_SERVER_ID - Get an ID of a server . . . 988
38.4.8 CBS_GET_PARAMETERS - Get scheduling parameters of a server . . . 989
38.4.9 CBS_SET_PARAMETERS - Set scheduling parameters . . . 990
38.4.10 CBS_GET_EXECUTION_TIME - Get elapsed execution time . . . 991
38.4.11 CBS_GET_REMAINING_BUDGET - Get remaining execution time . . . 992
38.4.12 CBS_GET_APPROVED_BUDGET - Get scheduler approved execution time 993

**39 Ada Support 995**
39.1 Introduction . . . 996
39.2 Ada Programming Language Support . . . 997
39.3 Classic API Ada Bindings . . . 998

**40 Linker Sets 999**
40.1 Introduction . . . 1000
40.2 Background . . . 1002
40.3 Directives . . . 1003
40.3.1 RTEMS_LINKER_SET_BEGIN - Designator of the linker set begin marker 1004
40.3.2 RTEMS_LINKER_SET_END - Designator of the linker set end marker . . 1005
40.3.3 RTEMS_LINKER_SET_SIZE - The linker set size in characters . . . 1006
40.3.4 RTEMS_LINKER_SET_ITEM_COUNT - The linker set item count . . . 1007
40.3.5 RTEMS_LINKER_SET_IS_EMPTY - Is the linker set empty? . . . 1008
40.3.6 RTEMS_LINKER_SET_FOREACH - Iterate through the linker set items . . 1009
40.3.7 RTEMS_LINKER_ROSET_DECLARE - Declares a read-only linker set . . . 1010
40.3.8 RTEMS_LINKER_ROSET - Defines a read-only linker set . . . 1011
40.3.9 RTEMS_LINKER_ROSET_ITEM_DECLARE - Declares a read-only linker set item . . . 1012

40.3.10 RTEMS_LINKER_ROSET_ITEM_ORDERED_DECLARE - Declares an ordered read-only linker set item . . . 1013
40.3.11 RTEMS_LINKER_ROSET_ITEM_REFERENCE - References a read-only linker set item . . . 1014
40.3.12 RTEMS_LINKER_ROSET_ITEM - Defines a read-only linker set item . . . 1015
40.3.13 RTEMS_LINKER_ROSET_ITEM_ORDERED - Defines an ordered read-only linker set item . . . 1016
40.3.14 RTEMS_LINKER_ROSET_CONTENT - Marks a declaration as a read-only linker set content . . . 1017
40.3.15 RTEMS_LINKER_RWSET_DECLARE - Declares a read-write linker set . . . 1018
40.3.16 RTEMS_LINKER_RWSET - Defines a read-write linker set . . . 1019
40.3.17 RTEMS_LINKER_RWSET_ITEM_DECLARE - Declares a read-write linker set item . . . 1020
40.3.18 RTEMS_LINKER_RWSET_ITEM_ORDERED_DECLARE - Declares an ordered read-write linker set item . . . 1021
40.3.19 RTEMS_LINKER_RWSET_ITEM_REFERENCE - References a read-write linker set item . . . 1022
40.3.20 RTEMS_LINKER_RWSET_ITEM - Defines a read-write linker set item . . . 1023
40.3.21 RTEMS_LINKER_RWSET_ITEM_ORDERED - Defines an ordered read-write linker set item . . . 1024
40.3.22 RTEMS_LINKER_RWSET_CONTENT - Marks a declaration as a read-write linker set content . . . 1025

**41 Directive Status Codes** **1027**
41.1 Introduction . . . 1028
41.2 Directives . . . 1029
41.2.1 STATUS_TEXT - Returns the enumeration name for a status code . . . 1030

**42 Example Application** **1031**

**43 Glossary** **1033**

**Bibliography** **1053**

**Index** **1057**

**Copyrights and License**

© 2017 Chris Johns
© 2017 Kuan-Hsun Chen
© 2015, 2020 embedded brains GmbH & Co. KG
© 2015, 2020 Sebastian Huber
© 2011 Petr Benes
© 2010 Gedare Bloom
© 1988, 2018 On-Line Applications Research Corporation (OAR)

This document is available under the Creative Commons Attribution-ShareAlike 4.0 International Public License.

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

The RTEMS Project is hosted at https://www.rtems.org. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

**RTEMS Online Resources**

| | |
|---|---|
| Home | https://www.rtems.org |
| Documentation | https://docs.rtems.org |
| Mailing Lists | https://lists.rtems.org |
| Bug Reporting | https://gitlab.rtems.org |
| Git Repositories | https://gitlab.rtems.org |
| Developers | https://gitlab.rtems.org |

CHAPTER

**ONE**

# PREFACE

In recent years, the cost required to develop a software product has increased significantly while the target hardware costs have decreased. Now a larger portion of money is expended in developing, using, and maintaining software. The trend in computing costs is the complete dominance of software over hardware costs. Because of this, it is necessary that formal disciplines be established to increase the probability that software is characterized by a high degree of correctness, maintainability, and portability. In addition, these disciplines must promote practices that aid in the consistent and orderly development of a software system within schedule and budgetary constraints. To be effective, these disciplines must adopt standards which channel individual software efforts toward a common goal.

The push for standards in the software development field has been met with various degrees of success. The Microprocessor Operating Systems Interfaces (MOSI) effort has experienced only limited success. As popular as the UNIX operating system has grown, the attempt to develop a standard interface definition to allow portable application development has only recently begun to produce the results needed in this area. Unfortunately, very little effort has been expended to provide standards addressing the needs of the real-time community. Several organizations have addressed this need during recent years.

The Real Time Executive Interface Definition (RTEID) was developed by Motorola with technical input from Software Components Group [Mot88]. RTEID was adopted by the VMEbus International Trade Association (VITA) as a baseline draft for their proposed standard multiprocessor, real-time executive interface, Open Real-Time Kernel Interface Definition (ORKID) [VIT90]. These two groups worked together with the IEEE P1003.4 committee to ensure that the functionality of their proposed standards is adopted as the real-time extensions to POSIX.

This proposed standard defines an interface for the development of real-time software to ease the writing of real-time application programs that are directly portable across multiple real-time executive implementations. This interface includes both the source code interfaces and run-time behavior as seen by a real-time application. It does not include the details of how a kernel implements these functions. The standard's goal is to serve as a complete definition of external interfaces so that application code that conforms to these interfaces will execute properly in all real-time executive environments. With the use of a standards compliant executive, routines that acquire memory blocks, create and manage message queues, establish and use semaphores, and send and receive signals need not be redeveloped for a different real-time environment as long as the new environment is compliant with the standard. Software developers need only concentrate on the hardware dependencies of the real-time system. Furthermore, most hardware dependencies for real-time applications can be localized to the device drivers.

A compliant executive provides simple and flexible real-time multiprocessing. It easily lends itself to both tightly-coupled and loosely-coupled configurations (depending on the system hard-

ware configuration). Objects such as tasks, queues, events, signals, semaphores, and memory blocks can be designated as global objects and accessed by any task regardless of which processor the object and the accessing task reside.

The acceptance of a standard for real-time executives will produce the same advantages enjoyed from the push for UNIX standardization by AT&T's System V Interface Definition and IEEE's POSIX efforts. A compliant multiprocessing executive will allow close coupling between UNIX systems and real-time executives to provide the many benefits of the UNIX development environment to be applied to real-time software development. Together they provide the necessary laboratory environment to implement real-time, distributed, embedded systems using a wide variety of computer architectures.

A study was completed in 1988, within the Research, Development, and Engineering Center, U.S. Army Missile Command, which compared the various aspects of the Ada programming language as they related to the application of Ada code in distributed and/or multiple processing systems. Several critical conclusions were derived from the study. These conclusions have a major impact on the way the Army develops application software for embedded applications. These impacts apply to both in-house software development and contractor developed software.

A conclusion of the analysis, which has been previously recognized by other agencies attempting to utilize Ada in a distributed or multiprocessing environment, is that the Ada programming language does not adequately support multiprocessing. Ada does provide a mechanism for multi-tasking, however, this capability exists only for a single processor system. The language also does not have inherent capabilities to access global named variables, flags or program code. These critical features are essential in order for data to be shared between processors. However, these drawbacks do have workarounds which are sometimes awkward and defeat the intent of software maintainability and portability goals.

Another conclusion drawn from the analysis, was that the run time executives being delivered with the Ada compilers were too slow and inefficient to be used in modern missile systems. A run time executive is the core part of the run time system code, or operating system code, that controls task scheduling, input/output management and memory management. Traditionally, whenever efficient executive (also known as kernel) code was required by the application, the user developed in-house software. This software was usually written in assembly language for optimization.

Because of this shortcoming in the Ada programming language, software developers in research and development and contractors for project managed systems, are mandated by technology to purchase and utilize off-the-shelf third party kernel code. The contractor, and eventually the Government, must pay a licensing fee for every copy of the kernel code used in an embedded system.

The main drawback to this development environment is that the Government does not own, nor has the right to modify code contained within the kernel. V&V techniques in this situation are more difficult than if the complete source code were available. Responsibility for system failures due to faulty software is yet another area to be resolved under this environment.

The Guidance and Control Directorate began a software development effort to address these problems. A project to develop an experimental run time kernel was begun that will eliminate the major drawbacks of the Ada programming language mentioned above. The Real Time Executive for Multiprocessor Systems (RTEMS) provides full capabilities for management of tasks, interrupts, time, and multiple processors in addition to those features typical of generic operating systems. The code is Government owned, so no licensing fees are necessary. RTEMS has been implemented in both the Ada and C programming languages. It has been ported to the following processor families:

- Adapteva Epiphany
- Altera NIOS II
- Analog Devices Blackfin
- Atmel AVR
- ARM
- Freescale (formerly Motorola) MC68xxx
- Freescale (formerly Motorola) MC683xx
- Freescale (formerly Motorola) ColdFire
- Intel i386 and above
- Lattice Semiconductor LM32
- NEC V850
- MIPS
- Moxie Processor
- OpenRISC
- PowerPC
- Renesas (formerly Hitachi) SuperH
- Renesas (formerly Hitachi) H8/300
- Renesas M32C
- SPARC v7, v8, and V9

Since almost all of RTEMS is written in a high level language, ports to additional processor families require minimal effort.

RTEMS multiprocessor support is capable of handling either homogeneous or heterogeneous systems. The kernel automatically compensates for architectural differences (byte swapping, etc.) between processors. This allows a much easier transition from one processor family to another without a major system redesign.

Since the proposed standards are still in draft form, RTEMS cannot and does not claim compliance. However, the status of the standard is being carefully monitored to guarantee that RTEMS provides the functionality specified in the standard. Once approved, RTEMS will be made compliant.

This document is a detailed users guide for a functionally compliant real-time multiprocessor executive. It describes the user interface and run-time behavior of Release 4.10.99.0 of the C interface to RTEMS.

CHAPTER

**TWO**

# OVERVIEW

## 2.1 Introduction

RTEMS, Real-Time Executive for Multiprocessor Systems, is a real-time executive (kernel) which provides a high performance environment for embedded military applications including the following features:

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling
- optional rate monotonic scheduling
- intertask communication and synchronization
- priority inheritance
- responsive interrupt management
- dynamic memory allocation
- high level of user configurability

This manual describes the usage of RTEMS for applications written in the C programming language. Those implementation details that are processor dependent are provided in the Applications Supplement documents. A supplement document which addresses specific architectural issues that affect RTEMS is provided for each processor type that is supported.

## 2.2 Real-time Application Systems

Real-time application systems are a special class of computer applications. They have a complex set of characteristics that distinguish them from other software problems. Generally, they must adhere to more rigorous requirements. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real-time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints referred to as deadlines. Systems can be buried by an avalanche of interdependent, asynchronous or cyclical event streams.

Deadlines can be further characterized as either hard or soft based upon the value of the results when produced after the deadline has passed. A deadline is hard if the results have no value or if their use will result in a catastrophic event. In contrast, results which are produced after a soft deadline may have some value.

Another distinguishing requirement of real-time application systems is the ability to coordinate or manage a large number of concurrent activities. Since software is a synchronous entity, this presents special problems. One instruction follows another in a repeating synchronous cycle. Even though mechanisms have been developed to allow for the processing of external asynchronous events, the software design efforts required to process and manage these events and tasks are growing more complicated.

The design process is complicated further by spreading this activity over a set of processors instead of a single processor. The challenges associated with designing and building real-time application systems become very complex when multiple processors are involved. New requirements such as interprocessor communication channels and global resources that must be shared between competing processors are introduced. The ramifications of multiple processors complicate each and every characteristic of a real-time system.

## 2.3 Real-time Executive

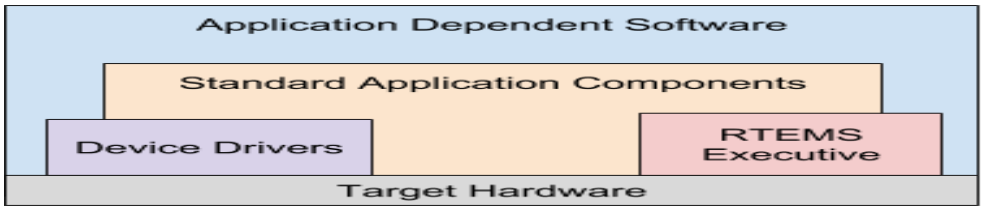
Fortunately, real-time operating systems or real-time executives serve as a cornerstone on which to build the application system. A real-time multitasking executive allows an application to be cast into a set of logical, autonomous processes or tasks which become quite manageable. Each task is internally synchronous, but different tasks execute independently, resulting in an asynchronous processing stream. Tasks can be dynamically paused for many reasons resulting in a different task being allowed to execute for a period of time. The executive also provides an interface to other system components such as interrupt handlers and device drivers. System components may request the executive to allocate and coordinate resources, and to wait for and trigger synchronizing conditions. The executive system calls effectively extend the CPU instruction set to support efficient multitasking. By causing tasks to travel through well-defined state transitions, system calls permit an application to demand-switch between tasks in response to real-time events.

By proper grouping of responses to stimuli into separate tasks, a system can now asynchronously switch between independent streams of execution, directly responding to external stimuli as they occur. This allows the system design to meet critical performance specifications which are typically measured by guaranteed response time and transaction throughput. The multiprocessor extensions of RTEMS provide the features necessary to manage the extra requirements introduced by a system distributed across several processors. It removes the physical barriers of processor boundaries from the world of the system designer, enabling more critical aspects of the system to receive the required attention. Such a system, based on an efficient real-time, multiprocessor executive, is a more realistic model of the outside world or environment for which it is designed. As a result, the system will always be more logical, efficient, and reliable.

By using the directives provided by RTEMS, the real-time applications developer is freed from the problem of controlling and synchronizing multiple tasks and processors. In addition, one need not develop, test, debug, and document routines to manage memory, pass messages, or provide mutual exclusion. The developer is then able to concentrate solely on the application. By using standard software components, the time and cost required to develop sophisticated real-time applications is significantly reduced.

## 2.4 RTEMS Application Architecture

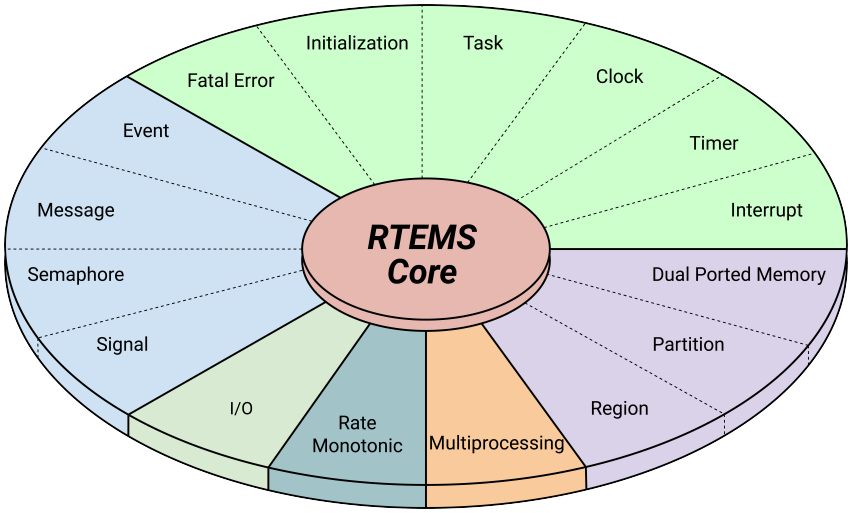
One important design goal of RTEMS was to provide a bridge between two critical layers of typical real-time systems. As shown in the following figure, RTEMS serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real-time applications can be localized to the low level device drivers.




The RTEMS I/O interface manager provides an efficient tool for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code that accesses them. A well designed real-time system can benefit from this architecture by building a rich library of standard application components which can be used repeatedly in other real-time projects.

## 2.5 RTEMS Internal Architecture

RTEMS can be viewed as a set of layered components that work in harmony to provide a set of services to a real-time application system. The executive interface presented to the application is formed by grouping directives into logical sets called resource managers. Functions utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core. The executive core depends on a small set of CPU dependent routines. Together these components provide a powerful run time environment that promotes the development of efficient real-time application systems. The following figure illustrates this organization:




Subsequent chapters present a detailed description of the capabilities provided by each of the following RTEMS managers:

- initialization
- task
- interrupt
- clock
- timer
- semaphore
- message
- event
- signal
- partition
- region
- dual ported memory
- I/O
- fatal error
- rate monotonic
- user extensions

- multiprocessing

## 2.6 User Customization and Extensibility

As thirty-two bit microprocessors have decreased in cost, they have become increasingly common in a variety of embedded systems. A wide range of custom and general-purpose processor boards are based on various thirty-two bit processors. RTEMS was designed to make no assumptions concerning the characteristics of individual microprocessor families or of specific support hardware. In addition, RTEMS allows the system developer a high degree of freedom in customizing and extending its features.

RTEMS assumes the existence of a supported microprocessor and sufficient memory for both RTEMS and the real-time application. Board dependent components such as clocks, interrupt controllers, or I/O devices can be easily integrated with RTEMS. The customization and extensibility features allow RTEMS to efficiently support as many environments as possible.

## 2.7 Portability

The issue of portability was the major factor in the creation of RTEMS. Since RTEMS is designed to isolate the hardware dependencies in the specific board support packages, the real-time application should be easily ported to any other processor. The use of RTEMS allows the development of real-time applications which can be completely independent of a particular microprocessor architecture.

## 2.8 Memory Requirements

Since memory is a critical resource in many real-time embedded systems, RTEMS was specifically designed to automatically leave out all services that are not required from the run-time environment. Features such as networking, various fileystems, and many other features are completely optional. This allows the application designer the flexibility to tailor RTEMS to most efficiently meet system requirements while still satisfying even the most stringent memory constraints. As a result, the size of the RTEMS executive is application dependent.

RTEMS requires RAM to manage each instance of an RTEMS object that is created. Thus the more RTEMS objects an application needs, the more memory that must be reserved. See *Configuring a System* (page 555).

RTEMS utilizes memory for both code and data space. Although RTEMS' data space must be in RAM, its code space can be located in either ROM or RAM.

## 2.9 Audience

This manual was written for experienced real-time software developers. Although some background is provided, it is assumed that the reader is familiar with the concepts of task management as well as intertask communication and synchronization. Since directives, user related data structures, and examples are presented in C, a basic understanding of the C programming language is required to fully understand the material presented. However, because of the similarity of the Ada and C RTEMS implementations, users will find that the use and behavior of the two implementations is very similar. A working knowledge of the target processor is helpful in understanding some of RTEMS' features. A thorough understanding of the executive cannot be obtained without studying the entire manual because many of RTEMS' concepts and features are interrelated. Experienced RTEMS users will find that the manual organization facilitates its use as a reference document.

## 2.10 Conventions

The following conventions are used in this manual:

- Significant words or phrases as well as all directive names are printed in bold type.
- Items in bold capital letters are constants defined by RTEMS. Each language interface provided by RTEMS includes a file containing the standard set of constants, data types, and structure definitions which can be incorporated into the user application.
- A number of type definitions are provided by RTEMS and can be found in rtems.h.
- The characters "0x" preceding a number indicates that the number is in hexadecimal format. Any other numbers are assumed to be in decimal format.

## 2.11 Manual Organization

This first chapter has presented the introductory and background material for the RTEMS executive. The remaining chapters of this manual present a detailed description of RTEMS and the environment, including run time behavior, it creates for the user.

A chapter is dedicated to each manager and provides a detailed discussion of each RTEMS manager and the directives which it provides. The presentation format for each directive includes the following sections:

- Calling sequence
- Directive status codes
- Description
- Notes

The following provides an overview of the remainder of this manual:

**Chapter 3:**
Key Concepts: presents an introduction to the ideas which are common across multiple RTEMS managers.

**Chapter 4:**
RTEMS Data Types: describes the fundamental data types shared by the services in the RTEMS Classic API.

**Chapter 5:**
Scheduling Concepts: details the various RTEMS scheduling algorithms and task state transitions.

**Chapter 6:**
Initialization Manager: describes the functionality and directives provided by the Initialization Manager.

**Chapter 7:**
Task Manager: describes the functionality and directives provided by the Task Manager.

**Chapter 8:**
Interrupt Manager: describes the functionality and directives provided by the Interrupt Manager.

**Chapter 9:**
Clock Manager: describes the functionality and directives provided by the Clock Manager.

**Chapter 10:**
Timer Manager: describes the functionality and directives provided by the Timer Manager.

**Chapter 11:**
Rate Monotonic Manager: describes the functionality and directives provided by the Rate Monotonic Manager.

**Chapter 12:**
Semaphore Manager: describes the functionality and directives provided by the Semaphore Manager.

**Chapter 13:**
Barrier Manager: describes the functionality and directives provided by the Barrier Manager.

**Chapter 14:**
Message Manager: describes the functionality and directives provided by the Message Manager.

**Chapter 15:**
Event Manager: describes the functionality and directives provided by the Event Manager.

**Chapter 16:**
Signal Manager: describes the functionality and directives provided by the Signal Manager.

**Chapter 17:**
Partition Manager: describes the functionality and directives provided by the Partition Manager.

**Chapter 18:**
Region Manager: describes the functionality and directives provided by the Region Manager.

**Chapter 19:**
Dual-Ported Memory Manager: describes the functionality and directives provided by the Dual-Ported Memory Manager.

**Chapter 20:**
I/O Manager: describes the functionality and directives provided by the I/O Manager.

**Chapter 21:**
Fatal Error Manager: describes the functionality and directives provided by the Fatal Error Manager.

**Chapter 22:**
Board Support Packages: defines the functionality required of user-supplied board support packages.

**Chapter 23:**
User Extensions: shows the user how to extend RTEMS to incorporate custom features.

**Chapter 24:**
Configuring a System: details the process by which one tailors RTEMS for a particular single-processor or multiprocessor application.

**Chapter 25:**
Self-Contained Objects: contains information about objects like threads, mutexes and semaphores.

**Chapter 26:**
Multiprocessing Manager: presents a conceptual overview of the multiprocessing capabilities provided by RTEMS as well as describing the Multiprocessing Communications Interface Layer and Multiprocessing Manager directives.

**Chapter 27:**
Symmetric Multiprocessing (SMP): information regarding the SMP features.

**Chapter 28:**
PCI Library: information about using the PCI bus in RTEMS.

**Chapter 29:**
Stack Bounds Checker: presents the capabilities of the RTEMS task stack checker which can report stack usage as well as detect bounds violations.

**Chapter 30:**
CPU Usage Statistics: presents the capabilities of the CPU Usage statistics gathered on a per task basis along with the mechanisms for reporting and resetting the statistics.

**Chapter 31:**
Object Services: presents a collection of helper services useful when manipulating RTEMS objects. These include methods to assist in obtaining an object's name in printable form. Additional services are provided to decompose an object Id and determine which API and object class it belongs to.

**Chapter 32:**
Chains: presents the methods provided to build, iterate and manipulate doubly-linked chains. This manager makes the chain implementation used internally by RTEMS to user space applications.

**Chapter 33:**
Red-Black Trees: information about how to use the Red-Black Tree API.

**Chapter 34:**
Timespec Helpers: presents a set of helper services useful when manipulating POSIX `struct timespec` instances.

**Chapter 35:**
Constant Bandwidth Server Scheduler API.

**Chapter 36:**
Ada Support: information about Ada programming language support.

**Chapter 37:**
Directive Status Codes: provides a definition of each of the directive status codes referenced in this manual.

**Chapter 38:**
Linker Sets: information about linker set features.

**Chapter 39:**
Example Application: provides a template for simple RTEMS applications.

**Chapter 40:**
Glossary: defines terms used throughout this manual.

**Chapter 41:**
References: References.

**Chapter 42:**
Index: Index.

CHAPTER

**THREE**

# KEY CONCEPTS

## 3.1 Introduction

The facilities provided by RTEMS are built upon a foundation of very powerful concepts. These concepts must be understood before the application developer can efficiently utilize RTEMS. The purpose of this chapter is to familiarize one with these concepts.

## 3.2 Objects

RTEMS provides directives which can be used to dynamically create, delete, and manipulate a set of predefined object types. These types include tasks, message queues, semaphores, memory regions, memory partitions, timers, ports, and rate monotonic periods. The object-oriented nature of RTEMS encourages the creation of modular applications built upon re-usable "building block" routines.

All objects are created on the local node as required by the application and have an RTEMS assigned ID. All objects have a user-assigned name. Although a relationship exists between an object's name and its RTEMS assigned ID, the name and ID are not identical. Object names are completely arbitrary and selected by the user as a meaningful "tag" which may commonly reflect the object's use in the application. Conversely, object IDs are designed to facilitate efficient object manipulation by the executive.

### 3.2.1 Object Names

An object name is an unsigned thirty-two bit entity associated with the object by the user. The data type `rtems_name` is used to store object names.

Although not required by RTEMS, object names are often composed of four ASCII characters which help identify that object. For example, a task which causes a light to blink might be called "LITE". The `rtems_build_name` routine is provided to build an object name from four ASCII characters. The following example illustrates this:

```
rtems_name my_name;
my_name = rtems_build_name( 'L', 'I', 'T', 'E' );
```

However, it is not required that the application use ASCII characters to build object names. For example, if an application requires one-hundred tasks, it would be difficult to assign meaningful ASCII names to each task. A more convenient approach would be to name them the binary values one through one-hundred, respectively.

RTEMS provides a helper routine, `rtems_object_get_name`, which can be used to obtain the name of any RTEMS object using just its ID. This routine attempts to convert the name into a printable string.

The following example illustrates the use of this method to print an object name:

```
#include <rtems.h>
#include <rtems/bspIo.h>
void print_name(rtems_id id)
{
    char  buffer[10];   /* name assumed to be 10 characters or less */
    char *result;
    result = rtems_object_get_name( id, sizeof(buffer), buffer );
    printk( "ID=0x%08x name=%s\n", id, ((result) ? result : "no name") );
}
```

### 3.2.2 Object Ids

an object id is a unique 32-bit unsigned integer value which uniquely identifies an object instance. object ids are passed as arguments to many directives in rtems and rtems translates the id to an internal object pointer. the efficient manipulation of object ids is critical to the performance of some rtems services.

There are multiple directives with names of the form `rtems_@CLASS@_ident` that take a name as argument and return the associated id if the name is found. The following is the set of name to id services: which can look up an object

- `rtems_extension_ident()`
- `rtems_barrier_ident()`
- `rtems_port_ident()`
- `rtems_message_queue_ident()`
- `rtems_partition_ident()`
- `rtems_region_ident()`
- `rtems_semaphore_ident()`
- `rtems_task_ident()`
- `rtems_timer_ident()`

### 3.2.3 Local and Global Scope

RTEMS supports uniprocessing, distributed multiprocessing, and Symmetric Multiprocessing (SMP) configurations. A uniprocessor system includes only a single processor in a single node. Distributed multiprocessor systems include multiple nodes, each of which is a single processor and is usually referred to as just multiprocessor mode for historical reasons. SMP systems consist of multiple processors cores in a single node.

In distributed multiprocessing configurations, there are multiple nodes in the system and object instances may be visible on just the creating node or to all nodes. If visible only to the creating node, this is referred to as **local scope** and corresponds to the RTEMS_LOCAL attribute setting which is the default. If RTEMS GLOBAL is specified as part of the object attributes, then the object instance has **global scope** and the object id can be used anywhere in the system to identify that object instance.

In uniprocessing and SMP configurations, there is only one node in the system and object instances are locally scoped to that node. Any attempt to create with the RTEMS_GLOBAL attribute is an error.

#### 3.2.3.1 Object ID Format

The thirty-two bit format for an object ID is composed of four parts: API, object class, node, and index. The data type `rtems_id` is used to store object IDs.

```
31      27 26   24 23          16 15                            0
+---------+-------+--------------+-------------------------------+
|         |       |              |                               |
|  Class  |  API  |     Node     |             Index             |
```

(continues on next page)

(continued from previous page)

```
|          |        |               |                                |
+---------+-------+--------------+-------------------------------+
```

The most significant five bits are the object class. The next three bits indicate the API to which the object class belongs. The next eight bits (16-23) are the number of the node on which this object was created. The node number is always one (1) in a single processor system. The least significant sixteen bits form an identifier within a particular object type. This identifier, called the object index, ranges in value from 1 to the maximum number of objects configured for this object type.

None of the fields in an object id may be zero except for the special case of RTEMS_SELF to indicate the currently running thread.

### 3.2.4 Object ID Description

The components of an object ID make it possible to quickly locate any object in even the most complicated multiprocessor system. Object ID's are associated with an object by RTEMS when the object is created and the corresponding ID is returned by the appropriate object create directive. The object ID is required as input to all directives involving objects, except those which create an object or obtain the ID of an object.

The object identification directives can be used to dynamically obtain a particular object's ID given its name. This mapping is accomplished by searching the name table associated with this object type. If the name is non-unique, then the ID associated with the first occurrence of the name will be returned to the application. Since object IDs are returned when the object is created, the object identification directives are not necessary in a properly designed single processor application.

In addition, services are provided to portably examine the subcomponents of an RTEMS ID. These services are described in detail later in this manual but are prototyped as follows:

```
Objects_APIs rtems_object_id_get_api( rtems_id );
uint32_t rtems_object_id_get_class( rtems_id );
uint32_t rtems_object_id_get_node( rtems_id );
uint16_t rtems_object_id_get_index( rtems_id );
```

An object control block is a data structure defined by RTEMS which contains the information necessary to manage a particular object type. For efficiency reasons, the format of each object type's control block is different. However, many of the fields are similar in function. The number of each type of control block is application dependent and determined by the values specified in the user's Configuration Table. An object control block is allocated at object create time and freed when the object is deleted. With the exception of user extension routines, object control blocks are not directly manipulated by user applications.

## 3.3 Communication and Synchronization

In real-time multitasking applications, the ability for cooperating execution threads to communicate and synchronize with each other is imperative. A real-time executive should provide an application with the following capabilities:

- Data transfer between cooperating tasks
- Data transfer between tasks and ISRs
- Synchronization of cooperating tasks
- Synchronization of tasks and ISRs

Most RTEMS managers can be used to provide some form of communication and/or synchronization. However, managers dedicated specifically to communication and synchronization provide well established mechanisms which directly map to the application's varying needs. This level of flexibility allows the application designer to match the features of a particular manager with the complexity of communication and synchronization required. The following managers were specifically designed for communication and synchronization:

- Semaphore
- Message Queue
- Event
- Signal

The semaphore manager supports mutual exclusion involving the synchronization of access to one or more shared user resources. Binary semaphores may utilize the optional priority inheritance algorithm to avoid the problem of priority inversion. The message manager supports both communication and synchronization, while the event manager primarily provides a high performance synchronization mechanism. The signal manager supports only asynchronous communication and is typically used for exception handling.

## 3.4 Locking Protocols

RTEMS supports the four locking protocols

- *Immediate Ceiling Priority Protocol (ICPP)* (page 29),
- *Priority Inheritance Protocol* (page 29),
- *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30), and
- *O(m) Independence-Preserving Protocol (OMIP)* (page 30)

for synchronization objects providing mutual-exclusion (mutex). The OMIP is only available in SMP configurations and replaces the priority inheritance protocol in this case. One aim of the locking protocols is to avoid priority inversion.

Since RTEMS 5.1, priority updates due to the locking protocols take place immediately and are propagated recursively. The mutex owner and wait for mutex relationships define a directed acyclic graph (DAG). The run-time of the mutex obtain, release and timeout operations depend on the complexity of this resource dependency graph.

### 3.4.1 Priority Inversion

Priority inversion is a form of indefinite postponement which is common in multitasking, preemptive executives with shared resources. Priority inversion occurs when a high priority tasks requests access to shared resource which is currently allocated to a low priority task. The high priority task must block until the low priority task releases the resource. This problem is exacerbated when the low priority task is prevented from executing by one or more medium priority tasks. Because the low priority task is not executing, it cannot complete its interaction with the resource and release that resource. The high priority task is effectively prevented from executing by lower priority tasks.

### 3.4.2 Immediate Ceiling Priority Protocol (ICPP)

Each mutex using the Immediate Ceiling Priority Protocol (ICPP) has a ceiling priority. The priority of the mutex owner is immediately raised to the ceiling priority of the mutex. In case the thread owning the mutex releases the mutex, then the normal priority of the thread is restored. This locking protocol is beneficial for schedulability analysis, see also [BW01].

This protocol avoids the possibility of changing the priority of the mutex owner multiple times since the ceiling priority must be set to the one of highest priority thread which will ever attempt to acquire that mutex. This requires an overall knowledge of the application as a whole. The need to identify the highest priority thread which will attempt to obtain a particular mutex can be a difficult task in a large, complicated system. Although the priority ceiling protocol is more efficient than the priority inheritance protocol with respect to the maximum number of thread priority changes which may occur while a thread owns a particular mutex, the priority inheritance protocol is more forgiving in that it does not require this apriori information.

### 3.4.3 Priority Inheritance Protocol

The priority of the mutex owner is raised to the highest priority of all threads that currently wait for ownership of this mutex [SRL90]. Since RTEMS 5.1, priority updates due to the priority inheritance protocol take place immediately and are propagated recursively. This means the priority inheritance is transitive since RTEMS 5.1. If a task A owning a priority inheritance mutex blocks on another priority inheritance mutex, then the owner of this mutex inherits the priority of the task A.

### 3.4.4 Multiprocessor Resource Sharing Protocol (MrsP)

The Multiprocessor Resource Sharing Protocol (MrsP) is a generalization of the priority ceiling protocol to clustered scheduling [BW13]. One of the design goals of MrsP is to enable an effective schedulability analysis using the sporadic task model. Each mutex using the MrsP has a ceiling priority for each scheduler instance. The priority of the mutex owner is immediately raised to the ceiling priority of the mutex defined for its home scheduler instance. In case the thread owning the mutex releases the mutex, then the normal priority of the thread is restored. Threads that wait for mutex ownership are not blocked with respect to the scheduler and instead perform a busy wait. The MrsP uses temporary thread migrations to foreign scheduler instances in case of a preemption of the mutex owner. This locking protocol is available since RTEMS 4.11. It was re-implemented in RTEMS 5.1 to overcome some shortcomings of the original implementation [CBHM15].

### 3.4.5 O(m) Independence-Preserving Protocol (OMIP)

The $O(m)$ Independence-Preserving Protocol (OMIP) is a generalization of the priority inheritance protocol to clustered scheduling which avoids the non-preemptive sections present with *priority boosting* [Bra13]. The $m$ denotes the number of processors in the system. Similar to the uniprocessor priority inheritance protocol, the OMIP mutexes do not need any external configuration data, e.g. a ceiling priority. This makes them a good choice for general purpose libraries that need internal locking. The complex part of the implementation is contained in the thread queues and shared with the MrsP support. This locking protocol is available since RTEMS 5.1.

## 3.5 Thread Queues

In case more than one *thread* may wait on a synchronization object, e.g. a semaphore or a message queue, then the waiting threads are added to a data structure called the thread queue. Thread queues are named task wait queues in the Classic API. There are two thread queuing disciplines available which define the order of the threads on a particular thread queue. Threads can wait in FIFO or priority order.

In uniprocessor configurations, the priority queuing discipline just orders the threads according to their current priority and in FIFO order in case of equal priorities. However, in SMP configurations, the situation is a bit more difficult due to the support for clustered scheduling. It makes no sense to compare the priority values of two different scheduler instances. Thus, it is impossible to simply use one plain priority queue for threads of different clusters. Two levels of queues can be used as one way to solve the problem. The top-level queue provides FIFO ordering and contains priority queues. Each priority queue is associated with a scheduler instance and contains only threads of this scheduler instance. Threads are enqueued in the priority queues corresponding to their scheduler instances. To dequeue a thread, the highest priority thread of the first priority queue is selected. Once this is done, the first priority queue is appended to the top-level FIFO queue. This guarantees fairness with respect to the scheduler instances.

Such a two-level queue needs a considerable amount of memory if fast enqueue and dequeue operations are desired. Providing this storage per thread queue would waste a lot of memory in typical applications. Instead, each thread has a queue attached which resides in a dedicated memory space independent of other memory used for the thread (this approach was borrowed from FreeBSD). In case a thread needs to block, there are two options

- the object already has a queue, then the thread enqueues itself to this already present queue and the queue of the thread is added to a list of free queues for this object, or
- otherwise, the queue of the thread is given to the object and the thread enqueues itself to this queue.

In case the thread is dequeued, there are two options

- the thread is the last thread in the queue, then it removes this queue from the object and reclaims it for its own purpose, or
- otherwise, the thread removes one queue from the free list of the object and reclaims it for its own purpose.

Since there are usually more objects than threads, this actually reduces the memory demands. In addition the objects only contain a pointer to the queue structure. This helps to hide implementation details. Inter-cluster priority queues are available since RTEMS 5.1.

A doubly-linked list (chain) is used to implement the FIFO queues yielding a $O(1)$ worst-case time complexity for enqueue and dequeue operations.

A red-black tree is used to implement the priority queues yielding a $O(log(n))$ worst-case time complexity for enqueue and dequeue operations with $n$ being the count of threads already on the queue.

## 3.6 Time

The development of responsive real-time applications requires an understanding of how RTEMS maintains and supports time-related operations. The basic unit of time in RTEMS is known as a *clock tick* or simply *tick*. The tick interval is defined by the application configuration option *CONFIGURE_MICROSECONDS_PER_TICK* (page 583). The tick interval defines the basic resolution of all interval and calendar time operations. Obviously, the directives which use intervals or wall time cannot operate without some external mechanism which provides a periodic clock tick. This clock tick is provided by the clock driver. The tick precision and stability depends on the clock driver and interrupt latency. Most clock drivers provide a timecounter to measure the time with a higher resolution than the tick.

By tracking time in units of ticks, RTEMS is capable of supporting interval timing functions such as task delays, timeouts, timeslicing, the delayed execution of timer service routines, and the rate monotonic scheduling of tasks. An interval is defined as a number of ticks relative to the current time. For example, when a task delays for an interval of ten ticks, it is implied that the task will not execute until ten clock ticks have occurred. All intervals are specified using data type `rtems_interval`.

A characteristic of interval timing is that the actual interval period may be a fraction of a tick less than the interval requested. This occurs because the time at which the delay timer is set up occurs at some time between two clock ticks. Therefore, the first countdown tick occurs in less than the complete time interval for a tick. This can be a problem if the tick resolution is large.

The rate monotonic scheduling algorithm is a hard real-time scheduling methodology. This methodology provides rules which allows one to guarantee that a set of independent periodic tasks will always meet their deadlines even under transient overload conditions. The rate monotonic manager provides directives built upon the Clock Manager's interval timer support routines.

Interval timing is not sufficient for the many applications which require that time be kept in wall time or true calendar form. Consequently, RTEMS maintains the current date and time. This allows selected time operations to be scheduled at an actual calendar date and time. For example, a task could request to delay until midnight on New Year's Eve before lowering the ball at Times Square. The data type `rtems_time_of_day` is used to specify calendar time in RTEMS services. See *Time and Date Data Structures* (page 238).

## 3.7 Timer and Timeouts

Timer and timeout services are a standard component of an operating system. The use cases fall roughly into two categories:

- Timeouts – used to detect if some operations need more time than expected. Since the unexpected happens hopefully rarely, timeout timers are usually removed before they expire. The critical operations are insert and removal. For example, they are important for the performance of a network stack.
- Timers – used to carry out some work in the future. They usually expire and need a high resolution. An example use case is a time driven scheduler, e.g. rate-monotonic or EDF.

In RTEMS versions prior to 5.1 the timer and timeout support was implemented by means of delta chains. This implementation was unfit for SMP systems due to several reasons. The new implementation present since RTEMS 5.1 uses a red-black tree with the expiration time as the key. This leads to $O(log(n))$ worst-case insert and removal operations for $n$ active timer or timeouts. Each processor provides its own timer and timeout service point so that it scales well with the processor count of the system. For each operation it is sufficient to acquire and release a dedicated SMP lock only once. The drawback is that a 64-bit integer type is required internally for the intervals to avoid a potential overflow of the key values.

An alternative to the red-black tree based implementation would be the use of a timer wheel based algorithm [VL87] which is used in Linux and FreeBSD [VC95] for example. A timer wheel based algorithm offers $O(1)$ worst-case time complexity for insert and removal operations. The drawback is that the run-time of the clock tick procedure is unpredictable due to the use of a hash table or cascading.

The red-black tree approach was selected for RTEMS, since it offers a more predictable run-time behaviour. However, this sacrifices the constant insert and removal operations offered by the timer wheel algorithms. See also [GN06]. The implementation can re-use the red-black tree support already used in other areas, e.g. for the thread priority queues. Less code is a good thing for size, testing and verification.

## 3.8 Memory Management

RTEMS memory management facilities can be grouped into two classes: dynamic memory allocation and address translation. Dynamic memory allocation is required by applications whose memory requirements vary through the application's course of execution. Address translation is needed by applications which share memory with another CPU or an intelligent Input/Output processor. The following RTEMS managers provide facilities to manage memory:

- Region
- Partition
- Dual Ported Memory

RTEMS memory management features allow an application to create simple memory pools of fixed size buffers and/or more complex memory pools of variable size segments. The partition manager provides directives to manage and maintain pools of fixed size entities such as resource control blocks. Alternatively, the region manager provides a more general purpose memory allocation scheme that supports variable size blocks of memory which are dynamically obtained and freed by the application. The dual-ported memory manager provides executive support for address translation between internal and external dual-ported RAM address space.

CHAPTER

**FOUR**

# RTEMS DATA TYPES

## 4.1 Introduction

This chapter contains a complete list of the RTEMS primitive data types in alphabetical order. This is intended to be an overview and the user is encouraged to look at the appropriate chapters in the manual for more information about the usage of the various data types.

## 4.2 List of Data Types

The following is a complete list of the RTEMS primitive data types in alphabetical order:

### 4.2.1 BSP_output_char_function_type

Polled character output functions shall have this type.

### 4.2.2 BSP_polling_getchar_function_type

Polled character input functions shall have this type.

### 4.2.3 Timer_Classes

The timer class indicates how the timer was most recently fired.

ENUMERATORS:

**TIMER_DORMANT**
This timer class indicates that the timer was never in use.

**TIMER_INTERVAL**
This timer class indicates that the timer is currently in use as an interval timer which will fire in the context of the clock tick *ISR*.

**TIMER_INTERVAL_ON_TASK**
This timer class indicates that the timer is currently in use as an interval timer which will fire in the context of the Timer Server task.

**TIMER_TIME_OF_DAY**
This timer class indicates that the timer is currently in use as an time of day timer which will fire in the context of the clock tick *ISR*.

**TIMER_TIME_OF_DAY_ON_TASK**
This timer class indicates that the timer is currently in use as an time of day timer which will fire in the context of the Timer Server task.

### 4.2.4 rtems_api_configuration_table

This structure contains a summary of the Classic API configuration.

MEMBERS:

**maximum_tasks**
This member contains the maximum number of Classic API Tasks configured for this application. See *CONFIGURE_MAXIMUM_TASKS* (page 621).

**notepads_enabled**
This member is true, if the Classic API Notepads are enabled, otherwise it is false.

**maximum_timers**
This member contains the maximum number of Classic API Timers configured for this application. See *CONFIGURE_MAXIMUM_TIMERS* (page 622).

**maximum_semaphores**
This member contains the maximum number of Classic API Semaphores configured for this application. See *CONFIGURE_MAXIMUM_SEMAPHORES* (page 620).

**maximum_message_queues**
This member contains the maximum number of Classic API Message Queues configured for this application. See *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615).

**maximum_partitions**
This member contains the maximum number of Classic API Partitions configured for this application. See *CONFIGURE_MAXIMUM_PARTITIONS* (page 616).

**maximum_regions**
This member contains the maximum number of Classic API Regions configured for this application. See *CONFIGURE_MAXIMUM_REGIONS* (page 619).

**maximum_ports**
This member contains the maximum number of Classic API Dual-Ported Memories configured for this application. See *CONFIGURE_MAXIMUM_PORTS* (page 618).

**maximum_periods**
This member contains the maximum number of Classic API Rate Monotonic Periods configured for this application. See *CONFIGURE_MAXIMUM_PERIODS* (page 617).

**maximum_barriers**
This member contains the maximum number of Classic API Barriers configured for this application. See *CONFIGURE_MAXIMUM_BARRIERS* (page 614).

**number_of_initialization_tasks**
This member contains the number of Classic API Initialization Tasks configured for this application. See *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634).

**User_initialization_tasks_table**
This member contains the pointer to Classic API Initialization Tasks Table of this application. See *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634).

DESCRIPTION:

Use *rtems_configuration_get_rtems_api_configuration()* (page 777) to get the configuration table.

### 4.2.5 rtems_asr

This type defines the return type of routines which are used to process asynchronous signals.

NOTES:

This type can be used to document asynchronous signal routines in the source code.

### 4.2.6 rtems_asr_entry

This type defines the prototype of routines which are used to process asynchronous signals.

### 4.2.7 rtems_assert_context

This structure provides the context in which an assertion failed.

MEMBERS:

**file**
This member provides the file name of the source code file containing the failed assertion statement.

**line**
This member provides the line number in the source code file containing the failed assertion statement.

**function**
This member provides the function name containing the failed assertion statement.

**failed_expression**
This member provides the expression of the failed assertion statement.

### 4.2.8 rtems_attribute

This type represents Classic API attributes.

NOTES:

Attributes are primarily used when creating objects.

### 4.2.9 rtems_device_driver

This type shall be used in device driver entry declarations and definitions.

NOTES:

Device driver entries return an `rtems_status_code` status code. This type definition helps to document device driver entries in the source code.

### 4.2.10 rtems_device_driver_entry

Device driver entries shall have this type.

### 4.2.11 rtems_device_major_number

This integer type represents the major number of devices.

NOTES:

The major number of a device is determined by *rtems_io_register_driver()* (page 474) and the application configuration (see *CONFIGURE_MAXIMUM_DRIVERS* (page 611)) .

### 4.2.12 rtems_device_minor_number

This integer type represents the minor number of devices.

NOTES:

The minor number of devices is managed by the device driver.

### 4.2.13 rtems_driver_address_table

This structure contains the device driver entries.

MEMBERS:

**initialization_entry**
This member is the device driver initialization entry. This entry is called by *rtems_io_initialize()* (page 477).

**open_entry**
This member is the device driver open entry. This entry is called by *rtems_io_open()* (page 479).

**close_entry**
This member is the device driver close entry. This entry is called by *rtems_io_close()* (page 480).

**read_entry**
This member is the device driver read entry. This entry is called by *rtems_io_read()* (page 481).

**write_entry**
This member is the device driver write entry. This entry is called by *rtems_io_write()* (page 482).

**control_entry**
This member is the device driver control entry. This entry is called by *rtems_io_control()* (page 483).

DESCRIPTION:

This structure is used to register a device driver via *rtems_io_register_driver()* (page 474).

### 4.2.14 rtems_event_set

This integer type represents a bit field which can hold exactly 32 individual events.

### 4.2.15 rtems_exception_frame

This structure represents an architecture-dependent exception frame.

### 4.2.16 rtems_extensions_table

The extensions table contains a set of extensions which may be registered in the system through the *CONFIGURE_INITIAL_EXTENSIONS* (page 573) application configuration option or the *rtems_extension_create()* (page 550) directive.

### 4.2.17 rtems_fatal_code

This integer type represents system termination codes.

DESCRIPTION:

This integer type is large enough to store a 32-bit integer or a pointer.

NOTES:

The interpretation of a system termination code depends on the system termination source, see *rtems_fatal_source* (page 41).

### 4.2.18 rtems_fatal_extension

Fatal extensions are invoked when the system should terminate.

PARAMETERS:

**`source`**
This parameter is the system termination source. The source indicates the component which caused the system termination request, see *rtems_fatal_source* (page 41). The system termination code may provide additional information related to the system termination request.

**`always_set_to_false`**
This parameter is a value equal to `false`.

**`code`**
This parameter is the system termination code. This value must be interpreted with respect to the source.

NOTES:

The fatal extensions are invoked in *extension forward order* and with maskable interrupts disabled.

The fatal extension should be extremely careful with respect to the RTEMS directives it calls. Depending on the system termination source, the system may be in an undefined and corrupt state.

It is recommended to register fatal extensions through *initial extension sets*, see *CONFIGURE_INITIAL_EXTENSIONS* (page 573).

### 4.2.19 rtems_fatal_source

This enumeration represents system termination sources.

NOTES:

The system termination code may provide additional information depending on the system termination source, see *rtems_fatal_code* (page 40).

### 4.2.20 rtems_id

This type represents RTEMS object identifiers.

### 4.2.21 rtems_initialization_tasks_table

This structure defines the properties of the Classic API user initialization task.

MEMBERS:

**name**
This member defines the task name.

**stack_size**
This member defines the task stack size in bytes.

**initial_priority**
This member defines the initial task priority.

**attribute_set**
This member defines the attribute set of the task.

**entry_point**
This member defines the entry point of the task.

**mode_set**
This member defines the initial modes of the task.

**argument**
This member defines the entry point argument of the task.

### 4.2.22 rtems_interrupt_attributes

This structure provides the attributes of an interrupt vector.

MEMBERS:

**is_maskable**
This member is true, if the interrupt vector is maskable by *rtems_interrupt_local_disable()* (page 168), otherwise it is false. Interrupt vectors which are not maskable by *rtems_interrupt_local_disable()* (page 168) should be used with care since they cannot use most operating system services.

**can_enable**
This member is true, if the interrupt vector can be enabled by *rtems_interrupt_vector_enable()* (page 194), otherwise it is false. When an interrupt vector can be enabled, this means that the enabled state can always be changed from disabled to enabled. For an interrupt vector which can be enabled it follows that it may be enabled.

**maybe_enable**
This member is true, if the interrupt vector may be enabled by *rtems_interrupt_vector_enable()* (page 194), otherwise it is false. When an interrupt vector may be enabled, this means that the enabled state may be changed from disabled to enabled. The requested enabled state change should be checked by *rtems_interrupt_vector_is_enabled()* (page 193). Some interrupt vectors may be optionally available and cannot be enabled on a particular *target*.

**can_disable**
This member is true, if the interrupt vector can be disabled by *rtems_interrupt_vector_disable()* (page 195), otherwise it is false. When an interrupt vector can be disabled, this means that the enabled state can be changed from enabled to disabled. For an interrupt vector which can be disabled it follows that it may be disabled.

**maybe_disable**
This member is true, if the interrupt vector may be disabled by *rtems_interrupt_vector_disable()* (page 195), otherwise it is false. When an interrupt vector may be disabled, this means that the enabled state may be changed from enabled to disabled. The requested enabled state change should be checked by *rtems_interrupt_vector_is_enabled()* (page 193). Some interrupt vectors may be always enabled and cannot be disabled on a particular *target*.

**can_raise**
This member is true, if the interrupt vector can be raised by *rtems_interrupt_raise()* (page 198), otherwise it is false.

**can_raise_on**
This member is true, if the interrupt vector can be raised on a processor by *rtems_interrupt_raise_on()* (page 199), otherwise it is false.

**can_clear**
This member is true, if the interrupt vector can be cleared by *rtems_interrupt_clear()* (page 200), otherwise it is false.

**cleared_by_acknowledge**
This member is true, if the pending status of the interrupt associated with the interrupt vector is cleared by an interrupt acknowledge from the processor, otherwise it is false.

**can_get_affinity**
This member is true, if the affinity set of the interrupt vector can be obtained by *rtems_interrupt_get_affinity()* (page 204), otherwise it is false.

**can_set_affinity**
This member is true, if the affinity set of the interrupt vector can be set by *rtems_interrupt_set_affinity()* (page 205), otherwise it is false.

**can_be_triggered_by_message**
This member is true, if the interrupt associated with the interrupt vector can be triggered by a message. Interrupts may be also triggered by signals, *rtems_interrupt_raise()* (page 198), or *rtems_interrupt_raise_on()* (page 199). Examples for message triggered interrupts are the PCIe MSI/MSI-X and the ARM GICv3 Locality-specific Peripheral Interrupts (LPI).

**trigger_signal**
This member describes the trigger signal of the interrupt associated with the interrupt vector. Interrupts are normally triggered by signals which indicate an interrupt request from a peripheral. Interrupts may be also triggered by messages, *rtems_interrupt_raise()* (page 198), or *rtems_interrupt_raise_on()* (page 199).

**can_get_priority**
This member is true, if the priority of the interrupt vector can be obtained by *rtems_interrupt_get_priority()* (page 201), otherwise it is false.

**can_set_priority**
This member is true, if the priority of the interrupt vector can be set by *rtems_interrupt_set_priority()* (page 202), otherwise it is false.

**maximum_priority**
This member represents the maximum priority value of the interrupt vector. By convention, the minimum priority value is zero. Lower priority values shall be associated with a higher importance. The higher the priority value, the less important is the service of the associated

interrupt vector. Where nested interrupts are supported, interrupts with a lower priority value may preempt other interrupts having a higher priority value.

DESCRIPTION:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to obtain the attributes of an interrupt vector.

### 4.2.23 rtems_interrupt_entry

This structure represents an interrupt entry.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

This structure shall be treated as an opaque data type from the *API* point of view. Members shall not be accessed directly. An entry may be initialized by *RTEMS_INTERRUPT_ENTRY_INITIALIZER()* (page 185) or *rtems_interrupt_entry_initialize()* (page 186). It may be installed for an interrupt vector with *rtems_interrupt_entry_install()* (page 187) and removed from an interrupt vector by *rtems_interrupt_entry_remove()* (page 189).

### 4.2.24 rtems_interrupt_handler

Interrupt handler routines shall have this type.

### 4.2.25 rtems_interrupt_level

This integer type represents interrupt levels.

### 4.2.26 rtems_interrupt_lock

This structure represents an ISR lock.

NOTES:

Lock objects are only needed in some RTEMS build configurations, for example where the SMP support is enabled. The `RTEMS_INTERRUPT_LOCK_NEEDS_OBJECT` constant can be used to determine whether a lock object is needed or not. This may help to reduce the memory demands of an application. All lock operations do not use the lock object parameter if lock objects are not needed.

```
#include <rtems.h>

#if RTEMS_INTERRUPT_LOCK_NEEDS_OBJECT
rtems_interrupt_lock lock = RTEMS_INTERRUPT_LOCK_INITIALIZER( "name" );
#endif

struct s {
#if RTEMS_INTERRUPT_LOCK_NEEDS_OBJECT
```

(continues on next page)

(continued from previous page)

```
  rtems_interrupt_lock lock;
#endif
  int foobar;
};
```

### 4.2.27 rtems_interrupt_lock_context

This structure provides an ISR lock context for acquire and release pairs.

### 4.2.28 rtems_interrupt_per_handler_routine

Visitor routines invoked by *rtems_interrupt_handler_iterate()* (page 208) shall have this type.

### 4.2.29 rtems_interrupt_server_action

This structure represents an interrupt server action.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

This structure shall be treated as an opaque data type from the *API* point of view. Members shall not be accessed directly.

### 4.2.30 rtems_interrupt_server_config

This structure defines an interrupt server configuration.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

See also *rtems_interrupt_server_create()* (page 212).

### 4.2.31 rtems_interrupt_server_control

This structure represents an interrupt server.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

This structure shall be treated as an opaque data type from the *API* point of view. Members shall not be accessed directly. The structure is initialized by *rtems_interrupt_server_create()* (page 212) and maintained by the interrupt server support.

### 4.2.32 rtems_interrupt_server_entry

This structure represents an interrupt server entry.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

This structure shall be treated as an opaque data type from the *API* point of view. Members shall not be accessed directly. An entry is initialized by *rtems_interrupt_server_entry_initialize()* (page 224) and destroyed by *rtems_interrupt_server_entry_destroy()* (page 227). Interrupt server actions can be prepended to the entry by *rtems_interrupt_server_action_prepend()* (page 225). The entry is submitted to be serviced by *rtems_interrupt_server_entry_submit()* (page 228).

### 4.2.33 rtems_interrupt_server_request

This structure represents an interrupt server request.

MEMBERS:

Members of the type shall not be accessed directly by the application.

NOTES:

This structure shall be treated as an opaque data type from the *API* point of view. Members shall not be accessed directly. A request is initialized by *rtems_interrupt_server_request_initialize()* (page 230) and destroyed by *rtems_interrupt_server_request_destroy()* (page 232). The interrupt vector of the request can be set by *rtems_interrupt_server_request_set_vector()* (page 231). The request is submitted to be serviced by *rtems_interrupt_server_request_submit()* (page 233).

### 4.2.34 rtems_interrupt_signal_variant

This enumeration provides interrupt trigger signal variants.

ENUMERATORS:

**RTEMS_INTERRUPT_UNSPECIFIED_SIGNAL**
This interrupt signal variant indicates that the interrupt trigger signal is unspecified.

**RTEMS_INTERRUPT_NO_SIGNAL**
This interrupt signal variant indicates that the interrupt cannot be triggered by a signal.

**RTEMS_INTERRUPT_SIGNAL_LEVEL_LOW**
This interrupt signal variant indicates that the interrupt is triggered by a low level signal.

**RTEMS_INTERRUPT_SIGNAL_LEVEL_HIGH**
This interrupt signal variant indicates that the interrupt is triggered by a high level signal.

**RTEMS_INTERRUPT_SIGNAL_EDGE_FALLING**
This interrupt signal variant indicates that the interrupt is triggered by a falling edge signal.

**RTEMS_INTERRUPT_SIGNAL_EDGE_RAISING**
This interrupt signal variant indicates that the interrupt is triggered by a raising edge signal.

### 4.2.35 rtems_interval

This type represents clock tick intervals.

### 4.2.36 rtems_isr

This type defines the return type of interrupt service routines.

DESCRIPTION:

This type can be used to document interrupt service routines in the source code.

### 4.2.37 rtems_isr_entry

Interrupt service routines installed by *rtems_interrupt_catch()* (page 162) shall have this type.

### 4.2.38 rtems_message_queue_config

This structure defines the configuration of a message queue constructed by *rtems_message_queue_construct()* (page 375).

MEMBERS:

**name**
This member defines the name of the message queue.

**maximum_pending_messages**
This member defines the maximum number of pending messages supported by the message queue.

**maximum_message_size**
This member defines the maximum message size supported by the message queue.

**storage_area**
This member shall point to the message buffer storage area begin. The message buffer storage area for the message queue shall be an array of the type defined by *RTEMS_MESSAGE_QUEUE_BUFFER()* (page 391) with a maximum message size equal to the maximum message size of this configuration.

**storage_size**
This member defines size of the message buffer storage area in bytes.

**storage_free**
This member defines the optional handler to free the message buffer storage area. It is called when the message queue is deleted. It is called from task context under protection of the object allocator lock. It is allowed to call `free()` in this handler. If handler is NULL, then no action will be performed.

**attributes**
This member defines the attributes of the message queue.

### 4.2.39 rtems_mode

This type represents a Classic API task mode set.

### 4.2.40 rtems_mp_packet_classes

This enumeration defines the MPCI packet classes.

### 4.2.41 rtems_mpci_entry

MPCI handler routines shall have this return type.

### 4.2.42 rtems_mpci_get_packet_entry

MPCI get packet routines shall have this type.

### 4.2.43 rtems_mpci_initialization_entry

MPCI initialization routines shall have this type.

### 4.2.44 rtems_mpci_receive_packet_entry

MPCI receive packet routines shall have this type.

### 4.2.45 rtems_mpci_return_packet_entry

MPCI return packet routines shall have this type.

### 4.2.46 rtems_mpci_send_packet_entry

MPCI send packet routines shall have this type.

### 4.2.47 rtems_mpci_table

This type represents the user-provided MPCI control.

### 4.2.48 rtems_multiprocessing_table

This type represents the user-provided MPCI configuration.

### 4.2.49 rtems_name

This type represents Classic API object names.

DESCRIPTION:

It is an unsigned 32-bit integer which can be treated as a numeric value or initialized using *rtems_build_name()* (page 902) to encode four ASCII characters. A value of zero may have a special meaning in some directives.

### 4.2.50 rtems_object_api_class_information

This structure is used to return information to the application about the objects configured for a specific API/Class combination.

MEMBERS:

**minimum_id**
This member contains the minimum valid object identifier for this class.

**maximum_id**
This member contains the maximum valid object identifier for this class.

**maximum**
This member contains the maximum number of active objects configured for this class.

**auto_extend**
This member is true, if this class is configured for automatic object extension, otherwise it is false.

**unallocated**
This member contains the number of currently inactive objects of this class.

### 4.2.51 rtems_option

This type represents a Classic API directive option set.

### 4.2.52 rtems_packet_prefix

This type represents the prefix found at the beginning of each MPCI packet sent between nodes.

### 4.2.53 rtems_rate_monotonic_period_states

This enumeration defines the states in which a period may be.

ENUMERATORS:

**RATE_MONOTONIC_INACTIVE**
This status indicates the period is off the watchdog chain, and has never been initialized.

**RATE_MONOTONIC_ACTIVE**
This status indicates the period is on the watchdog chain, and running. The owner may be executing or blocked waiting on another object.

**RATE_MONOTONIC_EXPIRED**
This status indicates the period is off the watchdog chain, and has expired. The owner may still execute and has taken too much time to complete this iteration of the period.

### 4.2.54 rtems_rate_monotonic_period_statistics

This structure provides the statistics of a period.

MEMBERS:

**count**
This member contains the number of periods executed.

**missed_count**
This member contains the number of periods missed.

**min_cpu_time**
This member contains the least amount of processor time used in a period.

**max_cpu_time**
This member contains the highest amount of processor time used in a period.

**total_cpu_time**
This member contains the total amount of processor time used in a period.

**min_wall_time**
This member contains the least amount of *CLOCK_MONOTONIC* time used in a period.

**max_wall_time**
This member contains the highest amount of *CLOCK_MONOTONIC* time used in a period.

**total_wall_time**
This member contains the total amount of *CLOCK_MONOTONIC* time used in a period.

### 4.2.55 rtems_rate_monotonic_period_status

This structure provides the detailed status of a period.

MEMBERS:

**owner**
This member contains the identifier of the owner task of the period.

**state**
This member contains the state of the period.

**since_last_period**
This member contains the time elapsed since the last successful invocation *rtems_rate_monotonic_period()* (page 317) using *CLOCK_MONOTONIC*. If the period is expired or has not been initiated, then this value has no meaning.

**executed_since_last_period**
This member contains the processor time consumed by the owner task since the last successful invocation *rtems_rate_monotonic_period()* (page 317). If the period is expired or has not been initiated, then this value has no meaning.

**postponed_jobs_count**
This member contains the count of jobs which are not released yet.

### 4.2.56 rtems_regulator_attributes

This structure defines the configuration of a regulator created by *rtems_regulator_create()* (page 832).

MEMBERS:

**deliverer**
This member contains a pointer to an application function invoked by the Delivery thread to output a message to the destination.

**deliverer_context**
This member contains a pointer to an application defined context which is passed to delivery function.

**maximum_message_size**
This member contains the maximum size message to process.

**maximum_messages**
This member contains the maximum number of messages to be able to buffer.

**output_thread_priority**
This member contains the priority of output thread.

**output_thread_stack_size**
This member contains the Stack size of output thread.

**output_thread_period**
This member contains the period (in ticks) of output thread.

**maximum_to_dequeue_per_period**
This member contains the maximum number of messages the output thread should dequeue and deliver per period.

NOTES:

This type is passed as an argument to *rtems_regulator_create()* (page 832).

### 4.2.57 rtems_regulator_deliverer

This type represents the function signature used to specify a delivery function for the RTEMS Regulator.

NOTES:

This type is used in the *rtems_regulator_attributes* (page 50) structure which is passed as an argument to *rtems_regulator_create()* (page 832).

### 4.2.58 rtems_regulator_statistics

This structure defines the statistics maintained by each Regulator instance.

MEMBERS:

**obtained**
This member contains the number of successfully obtained buffers.

**released**
This member contains the number of successfully released buffers.

**delivered**
This member contains the number of successfully delivered buffers.

**period_statistics**
This member contains the Rate Monotonic Period statistics for the Delivery Thread. It is an instance of the *rtems_rate_monotonic_period_statistics* (page 49) structure.

NOTES:

This type is passed as an argument to *rtems_regulator_get_statistics()* (page 840).

### 4.2.59 rtems_signal_set

This integer type represents a bit field which can hold exactly 32 individual signals.

### 4.2.60 rtems_stack_allocate_hook

A thread stack allocator allocate handler shall have this type.

### 4.2.61 rtems_stack_allocate_init_hook

A task stack allocator initialization handler shall have this type.

### 4.2.62 rtems_stack_free_hook

A task stack allocator free handler shall have this type.

### 4.2.63 rtems_status_code

This enumeration provides status codes for directives of the Classic API.

ENUMERATORS:

**RTEMS_SUCCESSFUL**
This status code indicates successful completion of a requested operation.

**RTEMS_TASK_EXITTED**
This status code indicates that a thread exitted.

**RTEMS_MP_NOT_CONFIGURED**
This status code indicates that multiprocessing was not configured.

**RTEMS_INVALID_NAME**
This status code indicates that an object name was invalid.

**RTEMS_INVALID_ID**
This status code indicates that an object identifier was invalid.

**RTEMS_TOO_MANY**
This status code indicates you have attempted to create too many instances of a particular object class.

**RTEMS_TIMEOUT**
This status code indicates that a blocking directive timed out.

**RTEMS_OBJECT_WAS_DELETED**
This status code indicates the object was deleted while the thread was blocked waiting.

**RTEMS_INVALID_SIZE**
This status code indicates that a specified size was invalid.

**RTEMS_INVALID_ADDRESS**
This status code indicates that a specified address was invalid.

**RTEMS_INVALID_NUMBER**
This status code indicates that a specified number was invalid.

**RTEMS_NOT_DEFINED**
This status code indicates that the item has not been initialized.

**RTEMS_RESOURCE_IN_USE**
This status code indicates that the object still had resources in use.

**RTEMS_UNSATISFIED**
This status code indicates that the request was not satisfied.

**RTEMS_INCORRECT_STATE**
This status code indicates that an object was in wrong state for the requested operation.

**RTEMS_ALREADY_SUSPENDED**
This status code indicates that the thread was already suspended.

**RTEMS_ILLEGAL_ON_SELF**
This status code indicates that the operation was illegal on the calling thread.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
This status code indicates that the operation was illegal on a remote object.

**RTEMS_CALLED_FROM_ISR**
This status code indicates that the operation should not be called from this execution environment.

**RTEMS_INVALID_PRIORITY**
This status code indicates that an invalid thread priority was provided.

**RTEMS_INVALID_CLOCK**
This status code indicates that a specified date or time was invalid.

**RTEMS_INVALID_NODE**
This status code indicates that a specified node identifier was invalid.

**RTEMS_NOT_CONFIGURED**
This status code indicates that the directive was not configured.

**RTEMS_NOT_OWNER_OF_RESOURCE**
This status code indicates that the caller was not the owner of the resource.

**RTEMS_NOT_IMPLEMENTED**
This status code indicates the directive or requested portion of the directive is not implemented. This is a hint that you have stumbled across an opportunity to submit code to the RTEMS Project.

**RTEMS_INTERNAL_ERROR**
This status code indicates that an internal RTEMS inconsistency was detected.

**RTEMS_NO_MEMORY**
This status code indicates that the directive attempted to allocate memory but was unable to do so.

**RTEMS_IO_ERROR**
This status code indicates a device driver IO error.

**RTEMS_INTERRUPTED**
This status code is used internally by the implementation to indicate a blocking device driver call has been interrupted and should be reflected to the caller as interrupted.

**RTEMS_PROXY_BLOCKING**
This status code is used internally by the implementation when performing operations on behalf of remote tasks. This is referred to as proxying operations and this status indicates that the operation could not be completed immediately and the proxy is blocking.

### 4.2.64 rtems_task

This type defines the return type of task entry points.

DESCRIPTION:

This type can be used to document task entry points in the source code.

### 4.2.65 rtems_task_argument

This integer type represents task argument values.

NOTES:

The type is an architecture-specific unsigned integer type which is large enough to represent pointer values and 32-bit unsigned integers.

### 4.2.66 rtems_task_begin_extension

Task begin extensions are invoked when a task begins execution.

PARAMETERS:

**`executing`**
This parameter is the *TCB* of the executing thread.

NOTES:

The task begin extensions are invoked in *extension forward order*.

Task begin extensions are invoked with thread dispatching enabled. This allows the use of dynamic memory allocation, creation of POSIX keys, and use of C++ thread-local storage. Blocking synchronization primitives are allowed also.

The task begin extensions are invoked before the global construction.

The task begin extensions may be called as a result of a task restart through *rtems_task_restart()* (page 120).

### 4.2.67 rtems_task_config

This structure defines the configuration of a task constructed by *rtems_task_construct()* (page 113).

MEMBERS:

**name**
This member defines the name of the task.

**initial_priority**
This member defines the initial priority of the task.

**storage_area**
This member shall point to the task storage area begin. The task storage area will contain the task stack, the thread-local storage, and the floating-point context on architectures with a separate floating-point context.

The task storage area begin address and size should be aligned by `RTEMS_TASK_STORAGE_ALIGNMENT`. To avoid memory waste, use `RTEMS_ALIGNED()` and `RTEMS_TASK_STORAGE_ALIGNMENT` to enforce the recommended alignment of a statically allocated task storage area.

**storage_size**
This member defines size of the task storage area in bytes. Use the *RTEMS_TASK_STORAGE_SIZE()* (page 144) macro to determine the recommended task storage area size.

**maximum_thread_local_storage_size**
This member defines the maximum thread-local storage size supported by the task storage area. Use `RTEMS_ALIGN_UP()` and `RTEMS_TASK_STORAGE_ALIGNMENT` to adjust the size to meet the minimum alignment requirement of a thread-local storage area used to construct a task.

If the value is less than the actual thread-local storage size, then the task construction by *rtems_task_construct()* (page 113) fails.

If the is less than the task storage area size, then the task construction by *rtems_task_construct()* (page 113) fails.

The actual thread-local storage size is determined when the application executable is linked. The `rtems-exeinfo` command line tool included in the RTEMS Tools can be used to obtain the thread-local storage size and alignment of an application executable.

The application may configure the maximum thread-local storage size for all threads explicitly through the *CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE* (page 578) configuration option.

**storage_free**
This member defines the optional handler to free the task storage area. It is called on exactly two mutually exclusive occasions. Firstly, when the task construction aborts due to a failed task create extension, or secondly, when the task is deleted. It is called from task context under protection of the object allocator lock. It is allowed to call `free()` in this handler. If handler is NULL, then no action will be performed.

**initial_modes**
This member defines the initial modes of the task.

**attributes**
This member defines the attributes of the task.

### 4.2.68 rtems_task_create_extension

Task create extensions are invoked when a task is created.

PARAMETERS:

**`executing`**
This parameter is the *TCB* of the executing thread. When the idle thread is created, the executing thread is equal to NULL.

**`created`**
This parameter is the *TCB* of the created thread.

RETURN VALUES:

Returns true, if the task create extension was successful, otherwise false.

NOTES:

The task create extensions are invoked in *extension forward order*.

The task create extensions are invoked after a new task has been completely initialized, but before it is started.

While normal tasks are created, the executing thread is the owner of the object allocator mutex. The object allocator mutex allows nesting, so the normal memory allocation routines can be used allocate memory for the created thread.

If the task create extension returns `false`, then the task create operation stops immediately and the entire task create operation will fail. In this case, all task delete extensions are invoked, see *rtems_task_delete_extension* (page 56).

### 4.2.69 rtems_task_delete_extension

Task delete extensions are invoked when a task is deleted.

PARAMETERS:

**`executing`**
This parameter is the *TCB* of the executing thread. If the idle thread is created and one of the initial task create extension fails, then the executing thread is equal to NULL.

**`created`**
This parameter is the *TCB* of the deleted thread. The executing and deleted arguments are never equal.

NOTES:

The task delete extensions are invoked in *extension reverse order*.

The task delete extensions are invoked by task create directives before an attempt to allocate a *TCB* is made.

If a task create extension failed, then a task delete extension may be invoked without a previous invocation of the corresponding task create extension of the extension set.

### 4.2.70 rtems_task_entry

This type defines the *task entry* point of an RTEMS task.

### 4.2.71 rtems_task_exitted_extension

Task exitted extensions are invoked when a task entry returns.

PARAMETERS:

**`executing`**
This parameter is the *TCB* of the executing thread.

NOTES:

The task exitted extensions are invoked in *extension forward order*.

### 4.2.72 rtems_task_priority

This integer type represents task priorities of the Classic API.

### 4.2.73 rtems_task_restart_extension

Task restart extensions are invoked when a task restarts.

PARAMETERS:

**executing**
This parameter is the *TCB* of the executing thread.

**restarted**
This parameter is the *TCB* of the executing thread. Yes, the executing thread.

NOTES:

The task restart extensions are invoked in *extension forward order*.

The task restart extensions are invoked in the context of the restarted thread right before the execution context is reloaded. The thread stack reflects the previous execution context.

Thread restart and delete requests issued by restart extensions lead to recursion.

### 4.2.74 rtems_task_start_extension

Task start extensions are invoked when a task was made ready for the first time.

PARAMETERS:

**executing**
This parameter is the *TCB* of the executing thread.

**started**
This parameter is the *TCB* of the started thread.

NOTES:

The task start extensions are invoked in *extension forward order*.

In SMP configurations, the thread may already run on another processor before the task start extensions are actually invoked. Task switch and task begin extensions may run before or in parallel with the thread start extension in SMP configurations, see *rtems_task_switch_extension* (page 57) and *rtems_task_begin_extension* (page 54).

### 4.2.75 rtems_task_switch_extension

Task switch extensions are invoked when a thread switch from an executing thread to a heir thread takes place.

PARAMETERS:

**executing**
This parameter is the *TCB* of the executing thread. In SMP configurations, this is the previously executing thread also known as the ancestor thread.

**heir**
This parameter is the *TCB* of the heir thread. In SMP configurations, this is the executing thread.

NOTES:

The task switch extensions are invoked in *extension forward order*.

The invocation conditions of the task switch extensions depend on whether RTEMS was built with SMP support enabled or disabled. A user must pay attention to the differences to correctly implement a task switch extension.

Where the system was built with SMP support disabled, the task switch extensions are invoked before the context switch from the currently executing thread to the heir thread. The `executing` is a pointer to the *TCB* of the currently executing thread. The `heir` is a pointer to the TCB of the heir thread. The context switch initiated through the multitasking start is not covered by the task switch extensions.

Where the system was built with SMP support enabled, the task switch extensions are invoked after the context switch to the heir thread. The `executing` is a pointer to the TCB of the previously executing thread. Despite the name, this is not the currently executing thread. The `heir` is a pointer to the TCB of the newly executing thread. This is the currently executing thread. The context switches initiated through the multitasking start are covered by the task switch extensions. The reason for the differences to uniprocessor configurations is that the context switch may update the heir thread of the processor. The task switch extensions are invoked with maskable interrupts disabled and with ownership of a processor-specific SMP lock. Task switch extensions may run in parallel on multiple processors. It is recommended to use thread-local or processor-specific data structures for task switch extensions. A global SMP lock should be avoided for performance reasons, see *rtems_interrupt_lock_initialize()* (page 172).

### 4.2.76 rtems_task_terminate_extension

Task terminate extensions are invoked when a task terminates.

PARAMETERS:

**executing**
This parameter is the *TCB* of the executing thread. This is the terminating thread.

NOTES:

The task terminate extensions are invoked in *extension reverse order*.

The task terminate extensions are invoked in the context of the terminating thread right before the thread dispatch to the heir thread should take place. The thread stack reflects the previous execution context. The POSIX cleanup and key destructors execute in this context.

Thread restart and delete requests issued by terminate extensions lead to recursion.

### 4.2.77 rtems_task_visitor

Visitor routines invoked by *rtems_task_iterate()* (page 143) shall have this type.

### 4.2.78 rtems_tcb

This structure represents the *TCB*.

### 4.2.79 rtems_time_of_day

This type represents Classic API calendar times.

MEMBERS:

**year**
This member contains the year A.D.

**month**
This member contains the month of the year with values from 1 to 12.

**day**
This member contains the day of the month with values from 1 to 31.

**hour**
This member contains the hour of the day with values from 0 to 23.

**minute**
This member contains the minute of the hour with values from 0 to 59.

**second**
This member contains the second of the minute with values from 0 to 59.

**ticks**
This member contains the clock tick of the second with values from 0 to *rtems_clock_get_ticks_per_second()* (page 264) minus one.

### 4.2.80 rtems_timer_information

The structure contains information about a timer.

MEMBERS:

**the_class**
The timer class member indicates how the timer was most recently fired.

**initial**
This member indicates the initial requested interval.

**start_time**
This member indicates the time the timer was initially scheduled. The time is in clock ticks since the clock driver initialization or the last clock tick counter overflow.

**stop_time**
This member indicates the time the timer was scheduled to fire. The time is in clock ticks since the clock driver initialization or the last clock tick counter overflow.

### 4.2.81 rtems_timer_service_routine

This type defines the return type of routines which can be fired by directives of the Timer Manager.

DESCRIPTION:

This type can be used to document timer service routines in the source code.

### 4.2.82 rtems_timer_service_routine_entry

This type defines the prototype of routines which can be fired by directives of the Timer Manager.

### 4.2.83 rtems_vector_number

This integer type represents interrupt vector numbers.

CHAPTER

**FIVE**

# SCHEDULING CONCEPTS

## 5.1 Introduction

The scheduling concepts relate to the allocation of processing time for tasks.

The concept of scheduling in real-time systems dictates the ability to provide an immediate response to specific external events, particularly the necessity of scheduling tasks to run within a specified time limit after the occurrence of an event. For example, software embedded in life-support systems used to monitor hospital patients must take instant action if a change in the patient's status is detected.

The component of RTEMS responsible for providing this capability is appropriately called the scheduler. The scheduler's sole purpose is to allocate the all important resource of processor time to the various tasks competing for attention. The directives provided by the Scheduler Manager are:

- *rtems_scheduler_ident()* (page 72) - Identifies a scheduler by the object name.
- *rtems_scheduler_ident_by_processor()* (page 73) - Identifies a scheduler by the processor index.
- *rtems_scheduler_ident_by_processor_set()* (page 74) - Identifies a scheduler by the processor set.
- *rtems_scheduler_get_maximum_priority()* (page 76) - Gets the maximum task priority of the scheduler.
- *rtems_scheduler_map_priority_to_posix()* (page 77) - Maps a Classic API task priority to the corresponding POSIX thread priority.
- *rtems_scheduler_map_priority_from_posix()* (page 78) - Maps a POSIX thread priority to the corresponding Classic API task priority.
- *rtems_scheduler_get_processor()* (page 79) - Returns the index of the current processor.
- *rtems_scheduler_get_processor_maximum()* (page 80) - Returns the processor maximum supported by the system.
- *rtems_scheduler_get_processor_set()* (page 81) - Gets the set of processors owned by the scheduler.
- *rtems_scheduler_add_processor()* (page 82) - Adds the processor to the set of processors owned by the scheduler.
- *rtems_scheduler_remove_processor()* (page 83) - Removes the processor from the set of processors owned by the scheduler.

## 5.2 Background

### 5.2.1 Scheduling Algorithms

RTEMS provides a plugin framework that allows it to support multiple scheduling algorithms. RTEMS includes multiple scheduling algorithms, and the user can select which of these they wish to use in their application at link-time. In addition, the user can implement their own scheduling algorithm and configure RTEMS to use it.

Supporting multiple scheduling algorithms gives the end user the option to select the algorithm which is most appropriate to their use case. Most real-time operating systems schedule tasks using a priority based algorithm, possibly with preemption control. The classic RTEMS scheduling algorithm which was the only algorithm available in RTEMS 4.10 and earlier, is a fixed-priority scheduling algorithm. This scheduling algorithm is suitable for uniprocessor (e.g., non-SMP) systems and is known as the *Deterministic Priority Scheduler*. Unless the user configures another scheduling algorithm, RTEMS will use this on uniprocessor systems.

### 5.2.2 Priority Scheduling

When using priority based scheduling, RTEMS allocates the processor using a priority-based, preemptive algorithm augmented to provide round-robin characteristics within individual priority groups. The goal of this algorithm is to guarantee that the task which is executing on the processor at any point in time is the one with the highest priority among all tasks in the ready state.

When a task is added to the ready chain, it is placed behind all other tasks of the same priority. This rule provides a round-robin within a priority group scheduling characteristic. This means that in a group of equal priority tasks, tasks will execute in the order they become ready or FIFO order. Even though there are ways to manipulate and adjust task priorities, the most important rule to remember is:

> **Note**
>
> Priority based scheduling algorithms will always select the highest priority task that is ready to run when allocating the processor to a task.

Priority scheduling is the most commonly used scheduling algorithm. It should be used by applications in which multiple tasks contend for CPU time or other resources, and there is a need to ensure certain tasks are given priority over other tasks.

There are a few common methods of accomplishing the mechanics of this algorithm. These ways involve a list or chain of tasks in the ready state.

- The least efficient method is to randomly place tasks in the ready chain forcing the scheduler to scan the entire chain to determine which task receives the processor.
- A more efficient method is to schedule the task by placing it in the proper place on the ready chain based on the designated scheduling criteria at the time it enters the ready state. Thus, when the processor is free, the first task on the ready chain is allocated the processor.
- Another mechanism is to maintain a list of FIFOs per priority. When a task is readied, it is placed on the rear of the FIFO for its priority. This method is often used with a bitmap

to assist in locating which FIFOs have ready tasks on them. This data structure has $O(1)$ insert, extract and find highest ready run-time complexities.

- A red-black tree may be used for the ready queue with the priority as the key. This data structure has $O(log(n))$ insert, extract and find highest ready run-time complexities while $n$ is the count of tasks in the ready queue.

RTEMS currently includes multiple priority based scheduling algorithms as well as other algorithms that incorporate deadline. Each algorithm is discussed in the following sections.

### 5.2.3 Scheduling Modification Mechanisms

RTEMS provides four mechanisms which allow the user to alter the task scheduling decisions:

- user-selectable task priority level
- task preemption control
- task timeslicing control
- manual round-robin selection

Each of these methods provides a powerful capability to customize sets of tasks to satisfy the unique and particular requirements encountered in custom real-time applications. Although each mechanism operates independently, there is a precedence relationship which governs the effects of scheduling modifications. The evaluation order for scheduling characteristics is always priority, preemption mode, and timeslicing. When reading the descriptions of timeslicing and manual round-robin it is important to keep in mind that preemption (if enabled) of a task by higher priority tasks will occur as required, overriding the other factors presented in the description.

#### 5.2.3.1 Task Priority and Scheduling

The most significant task scheduling modification mechanism is the ability for the user to assign a priority level to each individual task when it is created and to alter a task's priority at run-time, see *Task Priority* (page 100).

#### 5.2.3.2 Preemption

Another way the user can alter the basic scheduling algorithm is by manipulating the preemption mode flag (`RTEMS_PREEMPT_MASK`) of individual tasks. If preemption is disabled for a task (`RTEMS_NO_PREEMPT`), then the task will not relinquish control of the processor until it terminates, blocks, or re-enables preemption. Even tasks which become ready to run and possess higher priority levels will not be allowed to execute. Note that the preemption setting has no effect on the manner in which a task is scheduled. It only applies once a task has control of the processor.

#### 5.2.3.3 Timeslicing

Timeslicing or round-robin scheduling is an additional method which can be used to alter the basic scheduling algorithm. Like preemption, timeslicing is specified on a task by task basis using the timeslicing mode flag (`RTEMS_TIMESLICE_MASK`). If timeslicing is enabled for a task (`RTEMS_TIMESLICE`), then RTEMS will limit the amount of time the task can execute before the processor is allocated to another task. Each tick of the real-time clock reduces the currently running task's timeslice. When the execution time equals the timeslice, RTEMS will dispatch another task of the same priority to execute. If there are no other tasks of the same priority

ready to execute, then the current task is allocated an additional timeslice and continues to run. Remember that a higher priority task will preempt the task (unless preemption is disabled) as soon as it is ready to run, even if the task has not used up its entire timeslice.

#### 5.2.3.4 Manual Round-Robin

The final mechanism for altering the RTEMS scheduling algorithm is called manual round-robin. Manual round-robin is invoked by using the `rtems_task_wake_after` directive with a `ticks` parameter of `RTEMS_YIELD_PROCESSOR`. This allows a task to give up the processor and be immediately returned to the ready chain at the end of its priority group. If no other tasks of the same priority are ready to run, then the task does not lose control of the processor.

### 5.2.4 Dispatching Tasks

The dispatcher is the RTEMS component responsible for allocating the processor to a ready task. In order to allocate the processor to one task, it must be deallocated or retrieved from the task currently using it. This involves a concept called a context switch. To perform a context switch, the dispatcher saves the context of the current task and restores the context of the task which has been allocated to the processor. Saving and restoring a task's context is the storing/loading of all the essential information about a task to enable it to continue execution without any effects of the interruption. For example, the contents of a task's register set must be the same when it is given the processor as they were when it was taken away. All of the information that must be saved or restored for a context switch is located either in the TCB or on the task's stacks.

Tasks that utilize a numeric coprocessor and are created with the `RTEMS_FLOATING_POINT` attribute require additional operations during a context switch. These additional operations are necessary to save and restore the floating point context of `RTEMS_FLOATING_POINT` tasks. To avoid unnecessary save and restore operations, the state of the numeric coprocessor is only saved when a `RTEMS_FLOATING_POINT` task is dispatched and that task was not the last task to utilize the coprocessor.

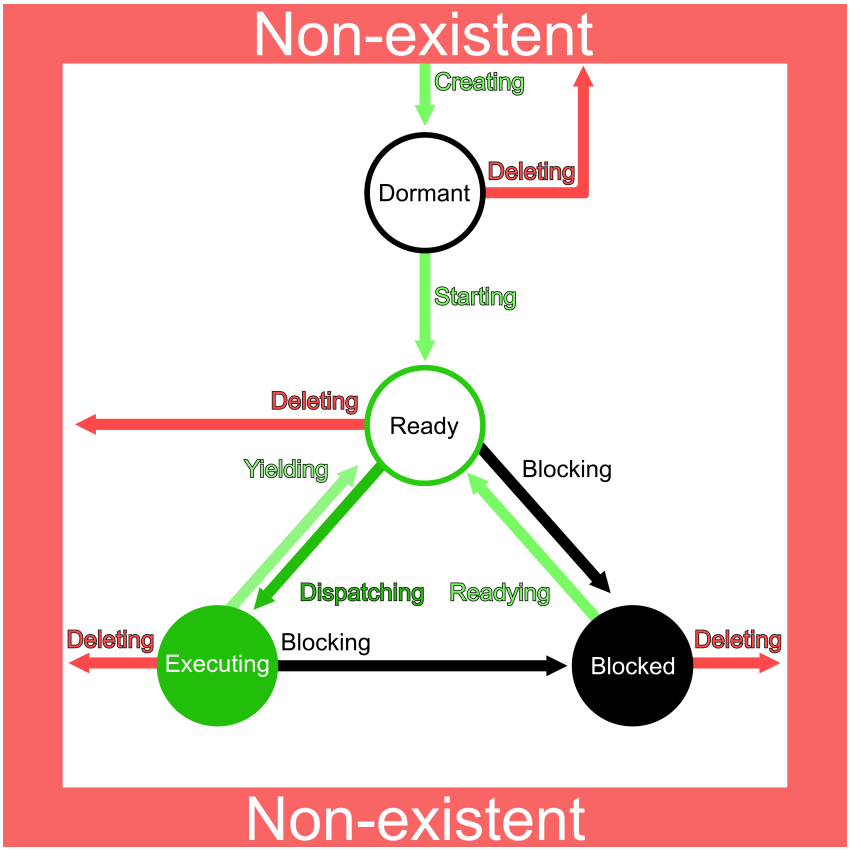
### 5.2.5 Task State Transitions

Tasks in an RTEMS system must always be in one of the five allowable task states. These states are: executing, ready, blocked, dormant, and non-existent.

A task occupies the non-existent state before a `rtems_task_create` has been issued on its behalf. A task enters the non-existent state from any other state in the system when it is deleted with the `rtems_task_delete` directive. While a task occupies this state it does not have a TCB or a task ID assigned to it; therefore, no other tasks in the system may reference this task.

When a task is created via the `rtems_task_create` directive, it enters the dormant state. This state is not entered through any other means. Although the task exists in the system, it cannot actively compete for system resources. It will remain in the dormant state until it is started via the `rtems_task_start` directive, at which time it enters the ready state. The task is now permitted to be scheduled for the processor and to compete for other system resources.

A task occupies the blocked state whenever it is unable to be scheduled to run. A running task may block itself or be blocked by other tasks in the system. The running task blocks itself through voluntary operations that cause the task to wait. The only way a task can block a task other than itself is with the `rtems_task_suspend` directive. A task enters the blocked state due to any of the following conditions:




- A task issues a `rtems_task_suspend` directive which blocks either itself or another task in the system.
- The running task issues a `rtems_barrier_wait` directive.
- The running task issues a `rtems_message_queue_receive` directive with the wait option, and the message queue is empty.
- The running task issues a `rtems_event_receive` directive with the wait option, and the currently pending events do not satisfy the request.
- The running task issues a `rtems_semaphore_obtain` directive with the wait option and the requested semaphore is unavailable.
- The running task issues a `rtems_task_wake_after` directive which blocks the task for the given count of ticks. If the count of ticks specified is zero, the task yields the processor and remains in the ready state.
- The running task issues a `rtems_task_wake_when` directive which blocks the task until the requested date and time arrives.
- The running task issues a `rtems_rate_monotonic_period` directive and must wait for the specified rate monotonic period to conclude.
- The running task issues a `rtems_region_get_segment` directive with the wait option and there is not an available segment large enough to satisfy the task's request.

A blocked task may also be suspended. Therefore, both the suspension and the blocking condition must be removed before the task becomes ready to run again.

A task occupies the ready state when it is able to be scheduled to run, but currently does not

have control of the processor. Tasks of the same or higher priority will yield the processor by either becoming blocked, completing their timeslice, or being deleted. All tasks with the same priority will execute in FIFO order. A task enters the ready state due to any of the following conditions:

- A running task issues a `rtems_task_resume` directive for a task that is suspended and the task is not blocked waiting on any resource.
- A running task issues a `rtems_message_queue_send`, `rtems_message_queue_broadcast`, or a `rtems_message_queue_urgent` directive which posts a message to the queue on which the blocked task is waiting.
- A running task issues an `rtems_event_send` directive which sends an event condition to a task that is blocked waiting on that event condition.
- A running task issues a `rtems_semaphore_release` directive which releases the semaphore on which the blocked task is waiting.
- The requested count of ticks has elapsed for a task which was blocked by a call to the `rtems_task_wake_after` directive.
- A timeout period expires for a task which blocked by a call to the `rtems_task_wake_when` directive.
- A running task issues a `rtems_region_return_segment` directive which releases a segment to the region on which the blocked task is waiting and a resulting segment is large enough to satisfy the task's request.
- A rate monotonic period expires for a task which blocked by a call to the `rtems_rate_monotonic_period` directive.
- A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout specified.
- A running task issues a directive which deletes a message queue, a semaphore, or a region on which the blocked task is waiting.
- A running task issues a `rtems_task_restart` directive for the blocked task.
- The running task, with its preemption mode enabled, may be made ready by issuing any of the directives that may unblock a task with a higher priority. This directive may be issued from the running task itself or from an ISR. A ready task occupies the executing state when it has control of the CPU. A task enters the executing state due to any of the following conditions:
- The task is the highest priority ready task in the system.
- The running task blocks and the task is next in the scheduling queue. The task may be of equal priority as in round-robin scheduling or the task may possess the highest priority of the remaining ready tasks.
- The running task may reenable its preemption mode and a task exists in the ready queue that has a higher priority than the running task.
- The running task lowers its own priority and another task is of higher priority as a result.
- The running task raises the priority of a task above its own and the running task is in preemption mode.

## 5.3 Uniprocessor Schedulers

All uniprocessor schedulers included in RTEMS are priority based. The processor is allocated to the highest priority task allowed to run.

### 5.3.1 Deterministic Priority Scheduler

This is the scheduler implementation which has always been in RTEMS. After the 4.10 release series, it was factored into a pluggable scheduler selection. It schedules tasks using a priority based algorithm which takes into account preemption. It is implemented using an array of FIFOs with a FIFO per priority. It maintains a bitmap which is used to track which priorities have ready tasks.

This algorithm is deterministic (e.g., predictable and fixed) in execution time. This comes at the cost of using slightly over three (3) kilobytes of RAM on a system configured to support 256 priority levels.

This scheduler is only aware of a single core.

### 5.3.2 Simple Priority Scheduler

This scheduler implementation has the same behaviour as the Deterministic Priority Scheduler but uses only one linked list to manage all ready tasks. When a task is readied, a linear search of that linked list is performed to determine where to insert the newly readied task.

This algorithm uses much less RAM than the Deterministic Priority Scheduler but is $O(n)$ where $n$ is the number of ready tasks. In a small system with a small number of tasks, this will not be a performance issue. Reducing RAM consumption is often critical in small systems that are incapable of supporting a large number of tasks.

This scheduler is only aware of a single core.

### 5.3.3 Earliest Deadline First Scheduler

This is an alternative scheduler in RTEMS for single-core applications. The primary EDF advantage is high total CPU utilization (theoretically up to 100%). It assumes that tasks have priorities equal to deadlines.

This EDF is initially preemptive, however, individual tasks may be declared not-preemptive. Deadlines are declared using only Rate Monotonic manager whose goal is to handle periodic behavior. Period is always equal to the deadline. All ready tasks reside in a single ready queue implemented using a red-black tree.

This implementation of EDF schedules two different types of task priority types while each task may switch between the two types within its execution. If a task does have a deadline declared using the Rate Monotonic manager, the task is deadline-driven and its priority is equal to deadline. On the contrary, if a task does not have any deadline or the deadline is cancelled using the Rate Monotonic manager, the task is considered a background task with priority equal to that assigned upon initialization in the same manner as for priority scheduler. Each background task is of lower importance than each deadline-driven one and is scheduled when no deadline-driven task and no higher priority background task is ready to run.

Every deadline-driven scheduling algorithm requires means for tasks to claim a deadline. The Rate Monotonic Manager is responsible for handling periodic execution. In RTEMS periods are equal to deadlines, thus if a task announces a period, it has to be finished until the end of this

period. The call of `rtems_rate_monotonic_period` passes the scheduler the length of an oncoming deadline. Moreover, the `rtems_rate_monotonic_cancel` and `rtems_rate_monotonic_delete` calls clear the deadlines assigned to the task.

### 5.3.4 Constant Bandwidth Server Scheduling (CBS)

This is an alternative scheduler in RTEMS for single-core applications. The CBS is a budget aware extension of EDF scheduler. The main goal of this scheduler is to ensure temporal isolation of tasks meaning that a task's execution in terms of meeting deadlines must not be influenced by other tasks as if they were run on multiple independent processors.

Each task can be assigned a server (current implementation supports only one task per server). The server is characterized by period (deadline) and computation time (budget). The ratio budget/period yields bandwidth, which is the fraction of CPU to be reserved by the scheduler for each subsequent period.

The CBS is equipped with a set of rules applied to tasks attached to servers ensuring that deadline miss because of another task cannot occur. In case a task breaks one of the rules, its priority is pulled to background until the end of its period and then restored again. The rules are:

- Task cannot exceed its registered budget,
- Task cannot be unblocked when a ratio between remaining budget and remaining deadline is higher than declared bandwidth.

The CBS provides an extensive API. Unlike EDF, the `rtems_rate_monotonic_period` does not declare a deadline because it is carried out using CBS API. This call only announces next period.

## 5.4 SMP Schedulers

All SMP schedulers included in RTEMS are priority based. The processors managed by a scheduler instance are allocated to the highest priority tasks allowed to run.

### 5.4.1 Earliest Deadline First SMP Scheduler

This is a job-level fixed-priority scheduler using the Earliest Deadline First (EDF) method. By convention, the maximum priority level is $min(INT_MAX, 2^{62}-1)$ for background tasks. Tasks without an active deadline are background tasks. In case deadlines are not used, then the EDF scheduler behaves exactly like a fixed-priority scheduler. The tasks with an active deadline have a higher priority than the background tasks. This scheduler supports task processor affinities of one-to-one and one-to-all, e.g., a task can execute on exactly one processor or all processors managed by the scheduler instance. The processor affinity set of a task must contain all online processors to select the one-to-all affinity. This is to avoid pathological cases if processors are added/removed to/from the scheduler instance at run-time. In case the processor affinity set contains not all online processors, then a one-to-one affinity will be used selecting the processor with the largest index within the set of processors currently owned by the scheduler instance. This scheduler algorithm supports *thread pinning* (page 868). The ready queues use a red-black tree with the task priority as the key.

This scheduler algorithm is the default scheduler in SMP configurations if more than one processor is configured (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)).

### 5.4.2 Deterministic Priority SMP Scheduler

A fixed-priority scheduler which uses a table of chains with one chain per priority level for the ready tasks. The maximum priority level is configurable. By default, the maximum priority level is 255 (256 priority levels), see *CONFIGURE_MAXIMUM_PRIORITY* (page 712).

### 5.4.3 Simple Priority SMP Scheduler

A fixed-priority scheduler which uses a sorted chain for the ready tasks. By convention, the maximum priority level is 255. The implementation limit is actually $2^{63} - 1$.

### 5.4.4 Arbitrary Processor Affinity Priority SMP Scheduler

A fixed-priority scheduler which uses a table of chains with one chain per priority level for the ready tasks. The maximum priority level is configurable. By default, the maximum priority level is 255 (256 priority levels), see *CONFIGURE_MAXIMUM_PRIORITY* (page 712). This scheduler supports arbitrary task processor affinities. The worst-case run-time complexity of some scheduler operations exceeds $O(n)$ while $n$ is the count of ready tasks.

## 5.5 Directives

This section details the directives of the Scheduler Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 5.5.1 rtems_scheduler_ident()

Identifies a scheduler by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the scheduler name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the scheduler will be stored in this object.

DESCRIPTION:

This directive obtains a scheduler identifier associated with the scheduler name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
There was no scheduler associated with the name.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

NOTES:

The scheduler name is determined by the scheduler configuration.

The scheduler identifier is used with other scheduler related directives to access the scheduler.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.2 rtems_scheduler_ident_by_processor()

Identifies a scheduler by the processor index.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_ident_by_processor(
  uint32_t  cpu_index,
  rtems_id *id
);
```

PARAMETERS:

**cpu_index**
This parameter is the processor index to identify the scheduler.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the scheduler will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The processor index was invalid.

**RTEMS_INCORRECT_STATE**
The processor index was valid, however, the corresponding processor was not owned by a scheduler.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.3 rtems_scheduler_ident_by_processor_set()

Identifies a scheduler by the processor set.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_ident_by_processor_set(
  size_t           cpusetsize,
  const cpu_set_t *cpuset,
  rtems_id        *id
);
```

PARAMETERS:

**cpusetsize**
This parameter is the size of the processor set referenced by `cpuset` in bytes. The size shall be positive.

**cpuset**
This parameter is the pointer to a `cpu_set_t`. The referenced processor set will be used to identify the scheduler.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the scheduler will be stored in this object.

DESCRIPTION:

The scheduler is selected according to the highest numbered online processor in the specified processor set.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `cpuset` parameter was NULL.

**RTEMS_INVALID_SIZE**
The processor set size was invalid.

**RTEMS_INVALID_NAME**
The processor set contained no online processor.

**RTEMS_INCORRECT_STATE**
The processor set was valid, however, the highest numbered online processor in the processor set was not owned by a scheduler.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.4 rtems_scheduler_get_maximum_priority()

Gets the maximum task priority of the scheduler.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_get_maximum_priority(
  rtems_id             scheduler_id,
  rtems_task_priority *priority
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**priority**
This parameter is the pointer to an *rtems_task_priority* (page 57) object. When the directive the maximum priority of the scheduler will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_ADDRESS**
The `priority` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.5 rtems_scheduler_map_priority_to_posix()

Maps a Classic API task priority to the corresponding POSIX thread priority.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_map_priority_to_posix(
  rtems_id            scheduler_id,
  rtems_task_priority priority,
  int                *posix_priority
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**priority**
This parameter is the Classic API task priority to map.

**posix_priority**
This parameter is the pointer to an `int` object. When the directive call is successful, the POSIX thread priority value corresponding to the specified Classic API task priority value will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `posix_priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_PRIORITY**
The Classic API task priority was invalid.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.6 rtems_scheduler_map_priority_from_posix()

Maps a POSIX thread priority to the corresponding Classic API task priority.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_map_priority_from_posix(
  rtems_id             scheduler_id,
  int                  posix_priority,
  rtems_task_priority *priority
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**posix_priority**
This parameter is the POSIX thread priority to map.

**priority**
This parameter is the pointer to an *rtems_task_priority* (page 57) object. When the directive call is successful, the Classic API task priority value corresponding to the specified POSIX thread priority value will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_PRIORITY**
The POSIX thread priority was invalid.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.7 rtems_scheduler_get_processor()

Returns the index of the current processor.

CALLING SEQUENCE:

```
uint32_t rtems_scheduler_get_processor( void );
```

DESCRIPTION:

Where the system was built with SMP support disabled, this directive evaluates to a compile time constant of zero.

Where the system was built with SMP support enabled, this directive returns the index of the current processor. The set of processor indices is the range of integers starting with zero up to *rtems_scheduler_get_processor_maximum()* (page 80) minus one.

RETURN VALUES:

Returns the index of the current processor.

NOTES:

Outside of sections with disabled thread dispatching the current processor index may change after every instruction since the thread may migrate from one processor to another. Sections with disabled interrupts are sections with thread dispatching disabled.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.8 rtems_scheduler_get_processor_maximum()

Returns the processor maximum supported by the system.

CALLING SEQUENCE:

```
uint32_t rtems_scheduler_get_processor_maximum( void );
```

DESCRIPTION:

Where the system was built with SMP support disabled, this directive evaluates to a compile time constant of one.

Where the system was built with SMP support enabled, this directive returns the minimum of the processors (physically or virtually) available at the *target* and the configured processor maximum (see *CONFIGURE_MAXIMUM_PROCESSORS* (page 577)). Not all processors in the range from processor index zero to the last processor index (which is the processor maximum minus one) may be configured to be used by a scheduler or may be online (online processors have a scheduler assigned).

RETURN VALUES:

Returns the processor maximum supported by the system.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.9 rtems_scheduler_get_processor_set()

Gets the set of processors owned by the scheduler.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_get_processor_set(
  rtems_id   scheduler_id,
  size_t     cpusetsize,
  cpu_set_t *cpuset
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**cpusetsize**
This parameter is the size of the processor set referenced by `cpuset` in bytes.

**cpuset**
This parameter is the pointer to a `cpu_set_t` object. When the directive call is successful, the processor set of the scheduler will be stored in this object. A set bit in the processor set means that the corresponding processor is owned by the scheduler, otherwise the bit is cleared.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `cpuset` parameter was NULL.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_SIZE**
The provided processor set was too small for the set of processors owned by the scheduler.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 5.5.10 rtems_scheduler_add_processor()

Adds the processor to the set of processors owned by the scheduler.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_add_processor(
  rtems_id scheduler_id,
  uint32_t cpu_index
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**cpu_index**
This parameter is the index of the processor to add.

DESCRIPTION:

This directive adds the processor specified by the `cpu_index` to the scheduler specified by `scheduler_id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_NOT_CONFIGURED**
The processor was not configured to be used by the application.

**RTEMS_INCORRECT_STATE**
The processor was configured to be used by the application, however, it was not online.

**RTEMS_RESOURCE_IN_USE**
The processor was already assigned to a scheduler.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 5.5.11 rtems_scheduler_remove_processor()

Removes the processor from the set of processors owned by the scheduler.

CALLING SEQUENCE:

```
rtems_status_code rtems_scheduler_remove_processor(
  rtems_id scheduler_id,
  uint32_t cpu_index
);
```

PARAMETERS:

**scheduler_id**
This parameter is the scheduler identifier.

**cpu_index**
This parameter is the index of the processor to remove.

DESCRIPTION:

This directive removes the processor specified by the `cpu_index` from the scheduler specified by `scheduler_id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_NUMBER**
The processor was not owned by the scheduler.

**RTEMS_RESOURCE_IN_USE**
The processor was required by at least one non-idle task that used the scheduler as its *home scheduler*.

**RTEMS_RESOURCE_IN_USE**
The processor was the last processor owned by the scheduler and there was at least one task that used the scheduler as a *helping scheduler*.

NOTES:

Removing a processor from a scheduler is a complex operation that involves all tasks of the system.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

CHAPTER

**SIX**

# INITIALIZATION MANAGER

## 6.1 Introduction

The Initialization Manager is responsible for initializing the system.

The system initialization includes the initialization of the Board Support Package, RTEMS, device drivers, the root filesystem, and the application. The *Fatal Error Manager* (page 515) is responsible for the system shutdown. The directives provided by the Initialization Manager are:

- *rtems_initialize_executive()* (page 95) - Initializes the system and starts multitasking.

## 6.2 Background

### 6.2.1 Initialization Tasks

Initialization task(s) are the mechanism by which RTEMS transfers initial control to the user's application. Initialization tasks differ from other application tasks in that they are defined in the User Initialization Tasks Table and automatically created and started by RTEMS as part of its initialization sequence. Since the initialization tasks are scheduled using the same algorithm as all other RTEMS tasks, they must be configured at a priority and mode which will ensure that they will complete execution before other application tasks execute. Although there is no upper limit on the number of initialization tasks, an application is required to define at least one.

A typical initialization task will create and start the static set of application tasks. It may also create any other objects used by the application. Initialization tasks which only perform initialization should delete themselves upon completion to free resources for other tasks. Initialization tasks may transform themselves into a "normal" application task. This transformation typically involves changing priority and execution mode. RTEMS does not automatically delete the initialization tasks.

### 6.2.2 The Idle Task

The Idle Task is the lowest priority task in a system and executes only when no other task is ready to execute. The default implementation of this task consists of an infinite loop. RTEMS allows the Idle Task body to be replaced by a CPU specific implementation, a BSP specific implementation or an application specific implementation.

The Idle Task is preemptible and *WILL* be preempted when any other task is made ready to execute. This characteristic is critical to the overall behavior of any application.

### 6.2.3 Initialization Manager Failure

System initialization errors are fatal. See *Internal Error Codes* (page 519).

## 6.3 Operations

### 6.3.1 Initializing RTEMS

The Initialization Manager `rtems_initialize_executive()` directives is called by the `boot_card()` routine which is invoked by the Board Support Package once a basic C run-time environment is set up. This consists of

- a valid and accessible text section, read-only data, read-write data and zero-initialized data,
- an initialization stack large enough to initialize the rest of the Board Support Package, RTEMS and the device drivers,
- all registers and components mandated by Application Binary Interface, and
- disabled interrupts.

The `rtems_initialize_executive()` directive uses a system initialization *linker set* (page 999) to initialize only those parts of the overall RTEMS feature set that is necessary for a particular application. Each RTEMS feature used the application may optionally register an initialization handler. The system initialization API is available via `#included <rtems/sysinit.h>`.

A list of all initialization steps follows. Some steps are optional depending on the requested feature set of the application. The initialization steps are execute in the order presented here.

**RTEMS_SYSINIT_RECORD**
Initialization of the event recording is the first initialization step. This allows to record the further system initialization. This step is optional and depends on the *CONFIGURE_RECORD_PER_PROCESSOR_ITEMS* (page 654) configuration option.

**RTEMS_SYSINIT_BSP_EARLY**
The Board Support Package may perform an early platform initialization in this step. This step is optional.

**RTEMS_SYSINIT_MEMORY**
The Board Support Package should initialize everything so that calls to `_Memory_Get()` can be made after this step. This step is optional.

**RTEMS_SYSINIT_DIRTY_MEMORY**
The free memory is dirtied in this step. This step is optional and depends on the `BSP_DIRTY_MEMORY` BSP option.

**RTEMS_SYSINIT_ISR_STACK**
The stack checker initializes the ISR stacks in this step. This step is optional and depends on the *CONFIGURE_STACK_CHECKER_ENABLED* (page 585) configuration option.

**RTEMS_SYSINIT_PER_CPU_DATA**
The per-CPU data is initialized in this step. This step is mandatory.

**RTEMS_SYSINIT_SBRK**
The Board Support Package may initialize the `sbrk()` support in this step. This step is optional.

**RTEMS_SYSINIT_WORKSPACE**
The workspace is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_MALLOC**
The C program heap is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_BSP_START**
The Board Support Package should perform a general platform initialization in this step (e.g. interrupt controller initialization). This step is mandatory.

**RTEMS_SYSINIT_CPU_COUNTER**
Initialization of the CPU counter hardware and support functions. The CPU counter is initialized early to allow its use in the tracing and profiling of the system initialization sequence. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_INITIAL_EXTENSIONS**
Registers the initial extensions. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_MP_EARLY**
In MPCI configurations, an early MPCI initialization is performed in this step. This step is mandatory in MPCI configurations.

**RTEMS_SYSINIT_DATA_STRUCTURES**
This directive is called when the Board Support Package has completed its basic initialization and allows RTEMS to initialize the application environment based upon the information in the Configuration Table, User Initialization Tasks Table, Device Driver Table, User Extension Table, Multiprocessor Configuration Table, and the Multiprocessor Communications Interface (MPCI) Table.

**RTEMS_SYSINIT_MP**
In MPCI configurations, a general MPCI initialization is performed in this step. This step is mandatory in MPCI configurations.

**RTEMS_SYSINIT_USER_EXTENSIONS**
Initialization of the User Extensions object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_TASKS**
Initialization of the Classic Tasks object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_TASKS_MP**
In MPCI configurations, the Classic Tasks MPCI support is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_TIMER**
Initialization of the Classic Timer object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_SIGNAL**
Initialization of the Classic Signal support. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_SIGNAL_MP**
In MPCI configurations, the Classic Signal MPCI support is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_EVENT**
Initialization of the Classic Event support. This step is optional and depends on the application

configuration. This step is only used on MPCI configurations.

**RTEMS_SYSINIT_CLASSIC_EVENT_MP**
In MPCI configurations, the Classic Event MPCI support is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_MESSAGE_QUEUE**
Initialization of the Classic Message Queue object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_SEMAPHORE**
Initialization of the Classic Semaphore object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_SEMAPHORE_MP**
In MPCI configurations, the Classic Semaphore MPCI support is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_PARTITION**
Initialization of the Classic Partition object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_PARTITION_MP**
In MPCI configurations, the Classic Partition MPCI support is initialized in this step. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_REGION**
Initialization of the Classic Region object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_DUAL_PORTED_MEMORY**
Initialization of the Classic Dual-Ported Memory object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_RATE_MONOTONIC**
Initialization of the Classic Rate-Monotonic object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_CLASSIC_BARRIER**
Initialization of the Classic Barrier object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_SIGNALS**
Initialization of the POSIX Signals support. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_THREADS**
Initialization of the POSIX Threads object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_MESSAGE_QUEUE**
Initialization of the POSIX Message Queue object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_SEMAPHORE**
Initialization of the POSIX Semaphore object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_TIMER**
Initialization of the POSIX Timer object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_SHM**
Initialization of the POSIX Shared Memory object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_KEYS**
Initialization of the POSIX Keys object class. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_CLEANUP**
Initialization of the POSIX Cleanup support. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_IDLE_THREADS**
Initialization of idle threads. This step is mandatory.

**RTEMS_SYSINIT_LIBIO**
Initialization of IO library. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_ROOT_FILESYSTEM**
Initialization of the root filesystem. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_DRVMGR**
Driver manager initialization. This step is optional and depends on the application configuration. Only available if the driver manager is enabled.

**RTEMS_SYSINIT_MP_SERVER**
In MPCI configurations, the MPCI server is initialized in this step. This step is mandatory in MPCI configurations.

**RTEMS_SYSINIT_BSP_PRE_DRIVERS**
Initialization step performed right before device drivers are initialized. This step is mandatory.

**RTEMS_SYSINIT_DRVMGR_LEVEL_1**
Driver manager level 1 initialization. This step is optional and depends on the application configuration. Only available if the driver manager is enabled.

**RTEMS_SYSINIT_DEVICE_DRIVERS**
This step initializes all statically configured device drivers and performs all RTEMS initialization which requires device drivers to be initialized. This step is mandatory. In a multiprocessor configuration, this service will initialize the Multiprocessor Communications Interface (MPCI) and synchronize with the other nodes in the system.

**RTEMS_SYSINIT_DRVMGR_LEVEL_2**
Driver manager level 2 initialization. This step is optional and depends on the application configuration. Only available if the driver manager is enabled.

**RTEMS_SYSINIT_DRVMGR_LEVEL_3**
Driver manager level 3 initialization. This step is optional and depends on the application configuration. Only available if the driver manager is enabled.

**RTEMS_SYSINIT_DRVMGR_LEVEL_4**
Driver manager level 4 initialization. This step is optional and depends on the application configuration. Only available if the driver manager is enabled.

**RTEMS_SYSINIT_MP_FINALIZE**
Finalize MPCI initialization. This step is mandatory on MPCI configurations.

**RTEMS_SYSINIT_CLASSIC_USER_TASKS**
Creates and starts the Classic initialization tasks. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_POSIX_USER_THREADS**
Creates POSIX initialization threads. This step is optional and depends on the application configuration.

**RTEMS_SYSINIT_STD_FILE_DESCRIPTORS**
Open the standard input, output and error file descriptors. This step is optional and depends on the application configuration.

The final action of the `rtems_initialize_executive()` directive is to start multitasking and switch to the highest priority ready thread. RTEMS does not return to the initialization context and the initialization stack may be re-used for interrupt processing.

Many of RTEMS actions during initialization are based upon the contents of the Configuration Table. For more information regarding the format and contents of this table, please refer to the chapter *Configuring a System* (page 555).

### 6.3.2 Global Construction

The *global construction* is carried out by the Classic API initialization task. If no Classic API initialization task exists, then it is carried out by the POSIX API initialization thread. If no initialization task or thread exists, then no global construction is performed. The Classic API task or POSIX API thread which carries out global construction is called the main thread. For configuration options related to initialization tasks, see *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634), *CONFIGURE_POSIX_INIT_THREAD_TABLE* (page 648), and *CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION* (page 707).

Global construction runs before the *task entry* of the main thread. The configuration of the main thread must take the global construction into account. In particular, the main thread stack size, priority, attributes and initial modes must be set accordingly. Thread-local objects and POSIX key values created during global construction are accessible by the main thread. If other initialization tasks are configured, and one of them has a higher priority than the main thread and the main thread is preemptible, this task executes before the global construction. In case the main thread blocks during global construction, then other tasks may run. In SMP configurations, other initialization tasks may run in parallel with global construction. Tasks created during global construction may preempt the main thread or run in parallel in SMP configurations. All RTEMS services allowed in task context are allowed during global construction.

Global constructors are C++ global object constructors or functions with the constructor attribute. For example, the following test program

```
#include <stdio.h>
#include <assert.h>

class A {
  public:
    A()
    {
      puts( "A:A()" );
```

(continues on next page)

(continued from previous page)

```
    }
};

static A a;

static thread_local int i;

static thread_local int j;

static __attribute__(( __constructor__ )) void b( void )
{
  i = 1;
  puts( "b()" );
}

static __attribute__(( __constructor__( 1000 ) )) void c( void )
{
  puts( "c()" );
}

int main( void )
{
  assert( i == 1 );
  assert( j == 0 );
  return 0;
}
```

should output:

```
c()
b()
A:A()
```

## 6.4 Directives

This section details the directives of the Initialization Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 6.4.1 rtems_initialize_executive()

Initializes the system and starts multitasking.

CALLING SEQUENCE:

```
void rtems_initialize_executive( void );
```

DESCRIPTION:

Iterates through the system initialization linker set and invokes the registered handlers. The final step is to start multitasking.

NOTES:

Errors in the initialization sequence are usually fatal and lead to a system termination.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive should be called by `boot_card()` only.
- The directive will not return to the caller.

CHAPTER

**SEVEN**

# TASK MANAGER

## 7.1 Introduction

The Task Manager provides a comprehensive set of directives to create, delete, and administer tasks. The directives provided by the Task Manager are:

- *rtems_task_create()* (page 109) - Creates a task.
- *rtems_task_construct()* (page 113) - Constructs a task from the specified task configuration.
- *rtems_task_ident()* (page 115) - Identifies a task by the object name.
- *rtems_task_self()* (page 117) - Gets the task identifier of the calling task.
- *rtems_task_start()* (page 118) - Starts the task.
- *rtems_task_restart()* (page 120) - Restarts the task.
- *rtems_task_delete()* (page 122) - Deletes the task.
- *rtems_task_exit()* (page 124) - Deletes the calling task.
- *rtems_task_suspend()* (page 125) - Suspends the task.
- *rtems_task_resume()* (page 126) - Resumes the task.
- *rtems_task_is_suspended()* (page 127) - Checks if the task is suspended.
- *rtems_task_set_priority()* (page 128) - Sets the real priority or gets the current priority of the task.
- *rtems_task_get_priority()* (page 130) - Gets the current priority of the task with respect to the scheduler.
- *rtems_task_mode()* (page 132) - Gets and optionally sets the mode of the calling task.
- *rtems_task_wake_after()* (page 134) - Wakes up after a count of *clock ticks* have occurred or yields the processor.
- *rtems_task_wake_when()* (page 135) - Wakes up when specified.
- *rtems_task_get_scheduler()* (page 136) - Gets the home scheduler of the task.
- *rtems_task_set_scheduler()* (page 137) - Sets the home scheduler for the task.
- *rtems_task_get_affinity()* (page 139) - Gets the processor affinity of the task.
- *rtems_task_set_affinity()* (page 141) - Sets the processor affinity of the task.
- *rtems_task_iterate()* (page 143) - Iterates over all tasks and invokes the visitor routine for each task.
- *RTEMS_TASK_STORAGE_SIZE()* (page 144) - Gets the recommended task storage area size for the size and task attributes.

## 7.2 Background

### 7.2.1 Task Definition

Many definitions of a task have been proposed in computer literature. Unfortunately, none of these definitions encompasses all facets of the concept in a manner which is operating system independent. Several of the more common definitions are provided to enable each user to select a definition which best matches their own experience and understanding of the task concept:

- a "dispatchable" unit.
- an entity to which the processor is allocated.
- an atomic unit of a real-time, multiprocessor system.
- single threads of execution which concurrently compete for resources.
- a sequence of closely related computations which can execute concurrently with other computational sequences.

From RTEMS' perspective, a task is the smallest thread of execution which can compete on its own for system resources. A task is manifested by the existence of a task control block (TCB).

### 7.2.2 Task Control Block

The Task Control Block (TCB) is an RTEMS defined data structure which contains all the information that is pertinent to the execution of a task. During system initialization, RTEMS reserves a TCB for each task configured. A TCB is allocated upon creation of the task and is returned to the TCB free list upon deletion of the task.

The TCB's elements are modified as a result of system calls made by the application in response to external and internal stimuli. TCBs are the only RTEMS internal data structure that can be accessed by an application via user extension routines. The TCB contains a task's name, ID, current priority, current and starting states, execution mode, TCB user extension pointer, scheduling control structures, as well as data required by a blocked task.

A task's context is stored in the TCB when a task switch occurs. When the task regains control of the processor, its context is restored from the TCB. When a task is restarted, the initial state of the task is restored from the starting context area in the task's TCB.

### 7.2.3 Task Memory

The system uses two separate memory areas to manage a task. One memory area is the *Task Control Block* (page 99). The other memory area is allocated from the stack space or provided by the user and contains

- the task stack,
- the thread-local storage (*TLS*), and
- an optional architecture-specific floating-point context.

The size of the thread-local storage is determined at link time. A user-provided task stack must take the size of the thread-local storage into account.

On architectures with a dedicated floating-point context, the application configuration assumes that every task is a floating-point task, but whether or not a task is actually floating-point is determined at runtime during task creation (see *Floating Point Considerations* (page 103)). In highly memory constrained systems this potential overestimate of the task stack space can

be mitigated through the *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584) configuration option and aligned task stack sizes for the tasks. A user-provided task stack must take the potential floating-point context into account.

### 7.2.4 Task Name

By default, the task name is defined by the task object name given to rtems_task_create(). The task name can be obtained with the pthread_getname_np() function. Optionally, a new task name may be set with the pthread_setname_np() function. The maximum size of a task name is defined by the application configuration option *CONFIGURE_MAXIMUM_THREAD_NAME_SIZE* (page 579).

### 7.2.5 Task States

A task may exist in one of the following five states:

- *executing* - Currently scheduled to the CPU
- *ready* - May be scheduled to the CPU
- *blocked* - Unable to be scheduled to the CPU
- *dormant* - Created task that is not started
- *non-existent* - Uncreated or deleted task

An active task may occupy the executing, ready, blocked or dormant state, otherwise the task is considered non-existent. One or more tasks may be active in the system simultaneously. Multiple tasks communicate, synchronize, and compete for system resources with each other via system calls. The multiple tasks appear to execute in parallel, but actually each is dispatched to the CPU for periods of time determined by the RTEMS scheduling algorithm. The scheduling of a task is based on its current state and priority.

### 7.2.6 Task Priority

A task's *priority* determines its importance in relation to the other tasks executing on the processor set owned by a *scheduler*. Normally, RTEMS supports 256 levels of priority ranging from 0 to 255. The priority level 0 represents a special priority reserved for the operating system. The data type `rtems_task_priority` is used to store task priorities. The maximum priority level depends on the configured scheduler, see *CONFIGURE_MAXIMUM_PRIORITY* (page 712), *Clustered Scheduler Configuration* (page 726), and *Scheduling Concepts* (page 61).

Tasks of numerically smaller priority values are more important tasks than tasks of numerically larger priority values. For example, a task at priority level 5 is of higher privilege than a task at priority level 10. There is no limit to the number of tasks assigned to the same priority.

Each task has a priority associated with it at all times. The initial value of this priority is assigned at task creation time. The priority of a task may be changed at any subsequent time.

Priorities are used by the scheduler to determine which ready task will be allowed to execute. In general, the higher the logical priority of a task, the more likely it is to receive processor execution time.

### 7.2.7 Task Mode

A task's execution mode is a combination of the following four components:

- preemption
- ASR processing
- timeslicing
- interrupt level

It is used to modify RTEMS' scheduling process and to alter the execution environment of the task. The data type `rtems_task_mode` is used to manage the task execution mode.

The preemption component allows a task to determine when control of the processor is relinquished. If preemption is disabled (`RTEMS_NO_PREEMPT`), the task will retain control of the processor as long as it is in the executing state - even if a higher priority task is made ready. If preemption is enabled (`RTEMS_PREEMPT`) and a higher priority task is made ready, then the processor will be taken away from the current task immediately and given to the higher priority task.

The timeslicing component is used by the RTEMS scheduler to determine how the processor is allocated to tasks of equal priority. If timeslicing is enabled (`RTEMS_TIMESLICE`), then RTEMS will limit the amount of time the task can execute before the processor is allocated to another ready task of equal priority. The length of the timeslice is application dependent and specified in the Configuration Table. If timeslicing is disabled (`RTEMS_NO_TIMESLICE`), then the task will be allowed to execute until a task of higher priority is made ready. If `RTEMS_NO_PREEMPT` is selected, then the timeslicing component is ignored by the scheduler.

The asynchronous signal processing component is used to determine when received signals are to be processed by the task. If signal processing is enabled (`RTEMS_ASR`), then signals sent to the task will be processed the next time the task executes. If signal processing is disabled (`RTEMS_NO_ASR`), then all signals received by the task will remain posted until signal processing is enabled. This component affects only tasks which have established a routine to process asynchronous signals.

The interrupt level component is used to determine which interrupts will be enabled when the task is executing. `RTEMS_INTERRUPT_LEVEL(n)` specifies that the task will execute at interrupt level n.

| | |
|---|---|
| `RTEMS_PREEMPT` | enable preemption (default) |
| `RTEMS_NO_PREEMPT` | disable preemption |
| `RTEMS_NO_TIMESLICE` | disable timeslicing (default) |
| `RTEMS_TIMESLICE` | enable timeslicing |
| `RTEMS_ASR` | enable ASR processing (default) |
| `RTEMS_NO_ASR` | disable ASR processing |
| `RTEMS_INTERRUPT_LEVEL(0)` | enable all interrupts (default) |
| `RTEMS_INTERRUPT_LEVEL(n)` | execute at interrupt level n |

The set of default modes may be selected by specifying the `RTEMS_DEFAULT_MODES` constant.

### 7.2.8 Task Life States

Independent of the task state with respect to the scheduler, the task life is determined by several orthogonal states:

- *protected* or *unprotected*
- *deferred life changes* or *no deferred life changes*
- *restarting* or *not restarting*
- *terminating* or *not terminating*
- *detached* or *not detached*

While the task life is *protected*, asynchronous task restart and termination requests are blocked. A task may still restart or terminate itself. All tasks are created with an unprotected task life. The task life protection is used by the system to prevent system resources being affected by asynchronous task restart and termination requests. The task life protection can be enabled (`PTHREAD_CANCEL_DISABLE`) or disabled (`PTHREAD_CANCEL_ENABLE`) for the calling task through the `pthread_setcancelstate()` directive.

While *deferred life changes* are enabled, asynchronous task restart and termination requests are delayed until the task performs a life change itself or calls `pthread_testcancel()`. Cancellation points are not implemented in RTEMS. Deferred task life changes can be enabled (`PTHREAD_CANCEL_DEFERRED`) or disabled (`PTHREAD_CANCEL_ASYNCHRONOUS`) for the calling task through the `pthread_setcanceltype()` directive. Classic API tasks are created with deferred life changes disabled. POSIX threads are created with deferred life changes enabled.

A task is made *restarting* by issuing a task restart request through the *rtems_task_restart()* (page 120) directive.

A task is made *terminating* by issuing a task termination request through the *rtems_task_exit()* (page 124), *rtems_task_delete()* (page 122), `pthread_exit()`, and `pthread_cancel()` directives.

When a *detached* task terminates, the termination procedure completes without the need for another task to join with the terminated task. Classic API tasks are created as not detached. The detached state of created POSIX threads is determined by the thread attributes. They are created as not detached by default. The calling task is made detached through the `pthread_detach()` directive. The *rtems_task_exit()* (page 124) directive and self deletion though *rtems_task_delete()* (page 122) directive make the calling task detached. In contrast, the `pthread_exit()` directive does not change the detached state of the calling task.

### 7.2.9 Accessing Task Arguments

All RTEMS tasks are invoked with a single argument which is specified when they are started or restarted. The argument is commonly used to communicate startup information to the task. The simplest manner in which to define a task which accesses it argument is:

```
rtems_task user_task(
    rtems_task_argument argument
);
```

Application tasks requiring more information may view this single argument as an index into an array of parameter blocks.

### 7.2.10 Floating Point Considerations

Please consult the *RTEMS CPU Architecture Supplement* if this section is relevant on your architecture. On some architectures the floating-point context is contained in the normal task context and this section does not apply.

Creating a task with the RTEMS_FLOATING_POINT attribute flag results in additional memory being allocated for the task to store the state of the numeric coprocessor during task switches. This additional memory is **not** allocated for RTEMS_NO_FLOATING_POINT tasks. Saving and restoring the context of a RTEMS_FLOATING_POINT task takes longer than that of a RTEMS_NO_FLOATING_POINT task because of the relatively large amount of time required for the numeric coprocessor to save or restore its computational state.

Since RTEMS was designed specifically for embedded military applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. In uniprocessor configurations, the state of the numeric coprocessor is only saved when a RTEMS_FLOATING_POINT task is dispatched and that task was not the last task to utilize the coprocessor. In a uniprocessor system with only one RTEMS_FLOATING_POINT task, the state of the numeric coprocessor will never be saved or restored.

Although the overhead imposed by RTEMS_FLOATING_POINT tasks is minimal, some applications may wish to completely avoid the overhead associated with RTEMS_FLOATING_POINT tasks and still utilize a numeric coprocessor. By preventing a task from being preempted while performing a sequence of floating point operations, a RTEMS_NO_FLOATING_POINT task can utilize the numeric coprocessor without incurring the overhead of a RTEMS_FLOATING_POINT context switch. This approach also avoids the allocation of a floating point context area. However, if this approach is taken by the application designer, **no** tasks should be created as RTEMS_FLOATING_POINT tasks. Otherwise, the floating point context will not be correctly maintained because RTEMS assumes that the state of the numeric coprocessor will not be altered by RTEMS_NO_FLOATING_POINT tasks. Some architectures with a dedicated floating-point context raise a processor exception if a task with RTEMS_NO_FLOATING_POINT issues a floating-point instruction, so this approach may not work at all.

If the supported processor type does not have hardware floating capabilities or a standard numeric coprocessor, RTEMS will not provide built-in support for hardware floating point on that processor. In this case, all tasks are considered RTEMS_NO_FLOATING_POINT whether created as RTEMS_FLOATING_POINT or RTEMS_NO_FLOATING_POINT tasks. A floating point emulation software library must be utilized for floating point operations.

On some processors, it is possible to disable the floating point unit dynamically. If this capability is supported by the target processor, then RTEMS will utilize this capability to enable the floating point unit only for tasks which are created with the RTEMS_FLOATING_POINT attribute. The consequence of a RTEMS_NO_FLOATING_POINT task attempting to access the floating point unit is CPU dependent but will generally result in an exception condition.

### 7.2.11 Building a Task Attribute Set

In general, an attribute set is built by a bitwise OR of the desired components. The set of valid task attribute components is listed below:

| | |
|---|---|
| RTEMS_NO_FLOATING_POINT | does not use coprocessor (default) |
| RTEMS_FLOATING_POINT | uses numeric coprocessor |
| RTEMS_LOCAL | local task (default) |
| RTEMS_GLOBAL | global task |

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. A component listed as a default is not required to appear in the component list, although it is a good programming practice to specify default components. If all defaults are desired, then RTEMS_DEFAULT_ATTRIBUTES should be used.

This example demonstrates the attribute_set parameter needed to create a local task which utilizes the numeric coprocessor. The attribute_set parameter could be RTEMS_FLOATING_POINT or RTEMS_LOCAL | RTEMS_FLOATING_POINT. The attribute_set parameter can be set to RTEMS_FLOATING_POINT because RTEMS_LOCAL is the default for all created tasks. If the task were global and used the numeric coprocessor, then the attribute_set parameter would be RTEMS_GLOBAL | RTEMS_FLOATING_POINT.

### 7.2.12 Building a Mode and Mask

In general, a mode and its corresponding mask is built by a bitwise OR of the desired components. The set of valid mode constants and each mode's corresponding mask constant is listed below:

| | |
|---|---|
| RTEMS_PREEMPT | is masked by RTEMS_PREEMPT_MASK and enables preemption |
| RTEMS_NO_PREEMPT | is masked by RTEMS_PREEMPT_MASK and disables preemption |
| RTEMS_NO_TIMESLICE | is masked by RTEMS_TIMESLICE_MASK and disables timeslicing |
| RTEMS_TIMESLICE | is masked by RTEMS_TIMESLICE_MASK and enables timeslicing |
| RTEMS_ASR | is masked by RTEMS_ASR_MASK and enables ASR processing |
| RTEMS_NO_ASR | is masked by RTEMS_ASR_MASK and disables ASR processing |
| RTEMS_INTERRUPT_LEVEL(0) | is masked by RTEMS_INTERRUPT_MASK and enables all interrupts |
| RTEMS_INTERRUPT_LEVEL(n) | is masked by RTEMS_INTERRUPT_MASK and sets interrupts level n |

Mode values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each mode appears exactly once in the component list. A mode component listed as a default is not required to appear in the mode component list, although it is a good programming practice to specify default components. If all defaults are desired, the mode RTEMS_DEFAULT_MODES and the mask RTEMS_ALL_MODE_MASKS should be used.

The following example demonstrates the mode and mask parameters used with the rtems_task_mode directive to place a task at interrupt level 3 and make it non-preemptible. The mode should be set to RTEMS_INTERRUPT_LEVEL(3) | RTEMS_NO_PREEMPT to indicate the desired preemption mode and interrupt level, while the mask parameter should be set to RTEMS_INTERRUPT_MASK | RTEMS_NO_PREEMPT_MASK to indicate that the calling task's interrupt level and preemption mode are being altered.

## 7.3 Operations

### 7.3.1 Creating Tasks

The `rtems_task_create` directive creates a task by allocating a task control block, assigning the task a user-specified name, allocating it a stack and floating point context area, setting a user-specified initial priority, setting a user-specified initial mode, and assigning it a task ID. Newly created tasks are initially placed in the dormant state. All RTEMS tasks execute in the most privileged mode of the processor.

### 7.3.2 Obtaining Task IDs

When a task is created, RTEMS generates a unique task ID and assigns it to the created task until it is deleted. The task ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_task_create` directive, the task ID is stored in a user provided location. Second, the task ID may be obtained later using the `rtems_task_ident` directive. The task ID is used by other directives to manipulate this task.

### 7.3.3 Starting and Restarting Tasks

The `rtems_task_start` directive is used to place a dormant task in the ready state. This enables the task to compete, based on its current priority, for the processor and other system resources. Any actions, such as suspension or change of priority, performed on a task prior to starting it are nullified when the task is started.

With the `rtems_task_start` directive the user specifies the task's starting address and argument. The argument is used to communicate some startup information to the task. As part of this directive, RTEMS initializes the task's stack based upon the task's initial execution mode and start address. The starting argument is passed to the task in accordance with the target processor's calling convention.

The `rtems_task_restart` directive restarts a task at its initial starting address with its original priority and execution mode, but with a possibly different argument. The new argument may be used to distinguish between the original invocation of the task and subsequent invocations. The task's stack and control block are modified to reflect their original creation values. Although references to resources that have been requested are cleared, resources allocated by the task are NOT automatically returned to RTEMS. A task cannot be restarted unless it has previously been started (i.e. dormant tasks cannot be restarted). All restarted tasks are placed in the ready state.

### 7.3.4 Suspending and Resuming Tasks

The `rtems_task_suspend` directive is used to place either the caller or another task into a suspended state. The task remains suspended until a `rtems_task_resume` directive is issued. This implies that a task may be suspended as well as blocked waiting either to acquire a resource or for the expiration of a timer.

The `rtems_task_resume` directive is used to remove another task from the suspended state. If the task is not also blocked, resuming it will place it in the ready state, allowing it to once again compete for the processor and resources. If the task was blocked as well as suspended, this directive clears the suspension and leaves the task in the blocked state.

Suspending a task which is already suspended or resuming a task which is not suspended is considered an error. The `rtems_task_is_suspended` can be used to determine if a task is currently suspended.

### 7.3.5 Delaying the Currently Executing Task

The `rtems_task_wake_after` directive creates a sleep timer which allows a task to go to sleep for a specified count of clock ticks. The task is blocked until the count of clock ticks has elapsed, at which time the task is unblocked. A task calling the `rtems_task_wake_after` directive with a delay of `RTEMS_YIELD_PROCESSOR` ticks will yield the processor to any other ready task of equal or greater priority and remain ready to execute.

The `rtems_task_wake_when` directive creates a sleep timer which allows a task to go to sleep until a specified date and time. The calling task is blocked until the specified date and time has occurred, at which time the task is unblocked.

### 7.3.6 Changing Task Priority

The `rtems_task_set_priority` directive is used to obtain or change the current priority of either the calling task or another task. If the new priority requested is `RTEMS_CURRENT_PRIORITY` or the task's actual priority, then the current priority will be returned and the task's priority will remain unchanged. If the task's priority is altered, then the task will be scheduled according to its new priority.

The `rtems_task_restart` directive resets the priority of a task to its original value.

### 7.3.7 Changing Task Mode

The `rtems_task_mode` directive is used to obtain or change the current execution mode of the calling task. A task's execution mode is used to enable preemption, timeslicing, ASR processing, and to set the task's interrupt level.

The `rtems_task_restart` directive resets the mode of a task to its original value.

### 7.3.8 Task Deletion

RTEMS provides the `rtems_task_delete` directive to allow a task to delete itself or any other task. This directive removes all RTEMS references to the task, frees the task's control block, removes it from resource wait queues, and deallocates its stack as well as the optional floating point context. The task's name and ID become inactive at this time, and any subsequent references to either of them is invalid. In fact, RTEMS may reuse the task ID for another task which is created later in the application. A specialization of `rtems_task_delete` is `rtems_task_exit` which deletes the calling task.

Unexpired delay timers (i.e. those used by `rtems_task_wake_after` and `rtems_task_wake_when`) and timeout timers associated with the task are automatically deleted, however, other resources dynamically allocated by the task are NOT automatically returned to RTEMS. Therefore, before a task is deleted, all of its dynamically allocated resources should be deallocated by the user. This may be accomplished by instructing the task to delete itself rather than directly deleting the task. Other tasks may instruct a task to delete itself by sending a "delete self" message, event, or signal, or by restarting the task with special arguments which instruct the task to delete itself.

### 7.3.9 Setting Affinity to a Single Processor

On some embedded applications targeting SMP systems, it may be beneficial to lock individual tasks to specific processors. In this way, one can designate a processor for I/O tasks, another for computation, etc.. The following illustrates the code sequence necessary to assign a task an affinity for processor with index `processor_index`.

```
#include <rtems.h>
#include <assert.h>

void pin_to_processor(rtems_id task_id, int processor_index)
{
    rtems_status_code sc;
    cpu_set_t         cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(processor_index, &cpuset);
    sc = rtems_task_set_affinity(task_id, sizeof(cpuset), &cpuset);
    assert(sc == RTEMS_SUCCESSFUL);
}
```

It is important to note that the `cpuset` is not validated until the `rtems_task_set_affinity` call is made. At that point, it is validated against the current system configuration.

### 7.3.10 Transition Advice for Removed Notepads

Task notepads and the associated directives *TASK_GET_NOTE - Get task notepad entry* (page 148) and *TASK_SET_NOTE - Set task notepad entry* (page 149) were removed in RTEMS 5.1. These were never thread-safe to access and subject to conflicting use of the notepad index by libraries which were designed independently.

It is recommended that applications be modified to use services which are thread safe and not subject to issues with multiple applications conflicting over the key (e.g. notepad index) selection. For most applications, POSIX Keys should be used. These are available in all RTEMS build configurations. It is also possible that thread-local storage (TLS) is an option for some use cases.

### 7.3.11 Transition Advice for Removed Task Variables

Task notepads and the associated directives *TASK_VARIABLE_ADD - Associate per task variable* (page 150), *TASK_VARIABLE_GET - Obtain value of a per task variable* (page 151) and *TASK_VARIABLE_DELETE - Remove per task variable* (page 152) were removed in RTEMS 5.1. Task variables must be replaced by POSIX Keys or thread-local storage (TLS). POSIX Keys are available in all configurations and support value destructors. For the TLS support consult the *RTEMS CPU Architecture Supplement*.

## 7.4 Directives

This section details the directives of the Task Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 7.4.1 rtems_task_create()

Creates a task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_create(
  rtems_name          name,
  rtems_task_priority initial_priority,
  size_t              stack_size,
  rtems_mode          initial_modes,
  rtems_attribute     attribute_set,
  rtems_id           *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the task.

**initial_priority**
This parameter is the initial task priority.

**stack_size**
This parameter is the task stack size in bytes.

**initial_modes**
This parameter is the initial mode set of the task.

**attribute_set**
This parameter is the attribute set of the task.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created task will be stored in this object.

DESCRIPTION:

This directive creates a task which resides on the local node. The task has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the task with other task related directives.

The **initial priority** of the task is specified in `initial_priority`. The *home scheduler* of the created task is the home scheduler of the calling task at some time point during the task creation. The initial task priority specified in `initial_priority` shall be valid for this scheduler.

The **stack size** of the task is specified in `stack_size`. If the requested stack size is less than the configured minimum stack size, then RTEMS will use the configured minimum as the stack size for this task. The configured minimum stack size is defined by the *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584) application configuration option. In addition to being able to specify the task stack size as a integer, there are two constants which may be specified:

- The `RTEMS_MINIMUM_STACK_SIZE` constant can be specified to use the **recommended minimum stack size** for the target processor. This value is selected by the RTEMS maintainers conservatively to minimize the risk of blown stacks for most user applications. Using this

constant when specifying the task stack size, indicates that the stack size will be at least RTEMS_MINIMUM_STACK_SIZE bytes in size. If the user configured minimum stack size is larger than the recommended minimum, then it will be used.

- The RTEMS_CONFIGURED_MINIMUM_STACK_SIZE constant can be specified to use the minimum stack size that was configured by the application. If not explicitly configured by the application, the default configured minimum stack size is the target processor dependent value RTEMS_MINIMUM_STACK_SIZE. Since this uses the configured minimum stack size value, you may get a stack size that is smaller or larger than the recommended minimum. This can be used to provide large stacks for all tasks on complex applications or small stacks on applications that are trying to conserve memory.

The **initial mode set** specified in initial_modes is built through a *bitwise or* of the mode constants described below. Not all combinations of modes are allowed. Some modes are mutually exclusive. If mutually exclusive modes are combined, the behaviour is undefined. Default task modes can be selected by using the RTEMS_DEFAULT_MODES constant. The task mode set defines

- the preemption mode of the task: RTEMS_PREEMPT (default) or RTEMS_NO_PREEMPT,
- the timeslicing mode of the task: RTEMS_TIMESLICE or RTEMS_NO_TIMESLICE (default),
- the *ASR* processing mode of the task: RTEMS_ASR (default) or RTEMS_NO_ASR,
- the interrupt level of the task: RTEMS_INTERRUPT_LEVEL() with a default of RTEMS_INTERRUPT_LEVEL( 0 ) which is associated with enabled interrupts.

The **initial preemption mode** of the task is enabled or disabled.

- An **enabled preemption** is the default and can be emphasized through the use of the RTEMS_PREEMPT mode constant.
- A **disabled preemption** is set by the RTEMS_NO_PREEMPT mode constant.

The **initial timeslicing mode** of the task is enabled or disabled.

- A **disabled timeslicing** is the default and can be emphasized through the use of the RTEMS_NO_TIMESLICE mode constant.
- An **enabled timeslicing** is set by the RTEMS_TIMESLICE mode constant.

The **initial ASR processing mode** of the task is enabled or disabled.

- An **enabled ASR processing** is the default and can be emphasized through the use of the RTEMS_ASR mode constant.
- A **disabled ASR processing** is set by the RTEMS_NO_ASR mode constant.

The **initial interrupt level mode** of the task is defined by RTEMS_INTERRUPT_LEVEL().

- Task execution with **interrupts enabled** the default and can be emphasized through the use of the RTEMS_INTERRUPT_LEVEL() mode macro with a value of zero (0) for the parameter. An interrupt level of zero is associated with enabled interrupts on all target processors.
- Task execution at a **non-zero interrupt level** can be specified by the RTEMS_INTERRUPT_LEVEL() mode macro with a non-zero value for the parameter. The interrupt level portion of the task mode supports a maximum of 256 interrupt levels. These levels are mapped onto the interrupt levels actually supported by the target processor in a processor dependent fashion.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant. The attribute set defines

- the scope of the task: `RTEMS_LOCAL` (default) or `RTEMS_GLOBAL` and
- the floating-point unit use of the task: `RTEMS_FLOATING_POINT` or `RTEMS_NO_FLOATING_POINT` (default).

The task has a local or global **scope** in a multiprocessing network (this attribute does not refer to SMP systems). The scope is selected by the mutually exclusive `RTEMS_LOCAL` and `RTEMS_GLOBAL` attributes.

- A **local scope** is the default and can be emphasized through the use of the `RTEMS_LOCAL` attribute. A local task can be only used by the node which created it.
- A **global scope** is established if the `RTEMS_GLOBAL` attribute is set. Setting the global attribute in a single node system has no effect.the

The **use of the floating-point unit** is selected by the mutually exclusive `RTEMS_FLOATING_POINT` and `RTEMS_NO_FLOATING_POINT` attributes. On some target processors, the use of the floating-point unit can be enabled or disabled for each task. Other target processors may have no hardware floating-point unit or enable the use of the floating-point unit for all tasks. Consult the *RTEMS CPU Architecture Supplement* for the details.

- A **disabled floating-point unit** is the default and can be emphasized through use of the `RTEMS_NO_FLOATING_POINT` attribute. For performance reasons, it is recommended that tasks not using the floating-point unit should specify this attribute.
- An **enabled floating-point unit** is selected by the `RTEMS_FLOATING_POINT` attribute.

RETURN VALUES:

**`RTEMS_SUCCESSFUL`**
The requested operation was successful.

**`RTEMS_INVALID_NAME`**
The `name` parameter was invalid.

**`RTEMS_INVALID_ADDRESS`**
The `id` parameter was NULL.

**`RTEMS_INVALID_PRIORITY`**
The `initial_priority` was invalid.

**`RTEMS_TOO_MANY`**
There was no inactive object available to create a task. The number of tasks available to the application is configured through the *CONFIGURE_MAXIMUM_TASKS* (page 621) application configuration option.

**`RTEMS_TOO_MANY`**
In multiprocessing configurations, there was no inactive global object available to create a global task. The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

**RTEMS_UNSATISFIED**
There was not enough memory to allocate the task storage area. The task storage area contains the task stack, the thread-local storage, and the floating point context.

**RTEMS_UNSATISFIED**
One of the task create extensions failed to create the task.

**RTEMS_UNSATISFIED**
In SMP configurations, the non-preemption mode was not supported.

**RTEMS_UNSATISFIED**
In SMP configurations, the interrupt level mode was not supported.

NOTES:

The task processor affinity is initialized to the set of online processors.

When created, a task is placed in the dormant state and can only be made ready to execute using the directive *rtems_task_start()* (page 118).

Application developers should consider the stack usage of the device drivers when calculating the stack size required for tasks which utilize the driver. The task stack size shall account for an target processor dependent interrupt stack frame which may be placed on the stack of the interrupted task while servicing an interrupt. The stack checker may be used to monitor the stack usage, see *CONFIGURE_STACK_CHECKER_ENABLED* (page 585).

For control and maintenance of the task, RTEMS allocates a *TCB* from the local TCB free pool and initializes it.

The TCB for a global task is allocated on the local node. Task should not be made global unless remote tasks must interact with the task. This is to avoid the system overhead incurred by the creation of a global task. When a global task is created, the task's name and identifier must be transmitted to every node in the system for insertion in the local copy of the global object table.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of tasks available to the application is configured through the *CONFIGURE_MAXIMUM_TASKS* (page 621) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 7.4.2 rtems_task_construct()

Constructs a task from the specified task configuration.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_construct(
  const rtems_task_config *config,
  rtems_id                *id
);
```

PARAMETERS:

**config**
This parameter is the pointer to an *rtems_task_config* (page 54) object. It configures the task.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the constructed task will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `config` parameter was NULL.

**RTEMS_INVALID_NAME**
The task name was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_PRIORITY**
The initial task priority was invalid.

**RTEMS_INVALID_SIZE**
The thread-local storage size is greater than the maximum thread-local storage size specified in the task configuration. The thread-local storage size is determined by the thread-local variables used by the application and *CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE* (page 578).

**RTEMS_INVALID_SIZE**
The task storage area was too small to provide a task stack of the configured minimum size, see *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584). The task storage area contains the task stack, the thread-local storage, and the floating-point context on architectures with a separate floating-point context.

**RTEMS_TOO_MANY**
There was no inactive task object available to construct a task.

**RTEMS_TOO_MANY**
In multiprocessing configurations, there was no inactive global object available to construct a global task.

**RTEMS_UNSATISFIED**
One of the task create extensions failed during the task construction.

**RTEMS_UNSATISFIED**
In SMP configurations, the non-preemption mode was not supported.

**RTEMS_UNSATISFIED**
In SMP configurations, the interrupt level mode was not supported.

NOTES:

In contrast to tasks created by *rtems_task_create()* (page 109), the tasks constructed by this directive use a user-provided task storage area. The task storage area contains the task stack, the thread-local storage, and the floating-point context on architectures with a separate floating-point context.

This directive is intended for applications which do not want to use the RTEMS Workspace and instead statically allocate all operating system resources. It is not recommended to use *rtems_task_create()* (page 109) and *rtems_task_construct()* (page 113) together in an application. It is also not recommended to use *rtems_task_construct()* (page 113) for drivers or general purpose libraries. The reason for these recommendations is that the task configuration needs settings which can be only given with a through knowledge of the application resources.

An application based solely on static allocation can avoid any runtime memory allocators. This can simplify the application architecture as well as any analysis that may be required.

The stack space estimate done by `<rtems/confdefs.h>` assumes that all tasks are created by *rtems_task_create()* (page 109). The estimate can be adjusted to take user-provided task storage areas into account through the *CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE* (page 624) application configuration option.

The *CONFIGURE_MAXIMUM_TASKS* (page 621) should include tasks constructed by *rtems_task_construct()* (page 113).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of tasks available to the application is configured through the *CONFIGURE_MAXIMUM_TASKS* (page 621) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 7.4.3 rtems_task_ident()

Identifies a task by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_ident(
  rtems_name name,
  uint32_t   node,
  rtems_id  *id
);
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**node**
This parameter is the node or node set to search for a matching object.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a task identifier associated with the task name specified in `name`.

A task may obtain its own identifier by specifying `RTEMS_WHO_AM_I` for the name.

The node to search is specified in `node`. It shall be

- a valid node number,
- the constant `RTEMS_SEARCH_ALL_NODES` to search in all nodes,
- the constant `RTEMS_SEARCH_LOCAL_NODE` to search in the local node only, or
- the constant `RTEMS_SEARCH_OTHER_NODES` to search in all nodes except the local node.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the specified nodes.

**RTEMS_INVALID_NODE**
In multiprocessing configurations, the specified node was invalid.

NOTES:

If the task name is not unique, then the task identifier will match the first task with that name in the search order. However, this task identifier is not guaranteed to correspond to the desired task.

The objects are searched from lowest to the highest index. If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched from lowest to the highest node number.

If node is a valid node number which does not represent the local node, then only the tasks exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

The task identifier is used with other task related directives to access the task.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 7.4.4 rtems_task_self()

Gets the task identifier of the calling task.

CALLING SEQUENCE:

```
rtems_id rtems_task_self( void );
```

DESCRIPTION:

This directive returns the task identifier of the calling task.

RETURN VALUES:

Returns the task identifier of the calling task.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 7.4.5 rtems_task_start()

Starts the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_start(
  rtems_id            id,
  rtems_task_entry    entry_point,
  rtems_task_argument argument
);
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**entry_point**
This parameter is the task entry point.

**argument**
This parameter is the task entry point argument.

DESCRIPTION:

This directive readies the task, specified by `id`, for execution based on the priority and execution mode specified when the task was created. The *task entry* point of the task is given in `entry_point`. The task's entry point argument is contained in `argument`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `entry_point` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INCORRECT_STATE**
The task was not in the dormant state.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

NOTES:

The type of the entry point argument is an unsigned integer type. However, the integer type has the property that any valid pointer to `void` can be converted to this type and then converted back to a pointer to `void`. The result will compare equal to the original pointer. The type can represent at least 32 bits. Some applications use the entry point argument as an index into a parameter table to get task-specific parameters.

Any actions performed on a dormant task such as suspension or change of priority are nullified when the task is initiated via the *rtems_task_start()* (page 118) directive.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.

### 7.4.6 rtems_task_restart()

Restarts the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_restart(
  rtems_id            id,
  rtems_task_argument argument
);
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**argument**
This parameter is the task entry point argument.

DESCRIPTION:

This directive resets the task specified by `id` to begin execution at its original entry point. The task's priority and execution mode are set to the original creation values. If the task is currently blocked, RTEMS automatically makes the task ready. A task can be restarted from any state, except the dormant state. The task's entry point argument is contained in `argument`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INCORRECT_STATE**
The task never started.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

NOTES:

The type of the entry point argument is an unsigned integer type. However, the integer type has the property that any valid pointer to `void` can be converted to this type and then converted back to a pointer to `void`. The result will compare equal to the original pointer. The type can represent at least 32 bits. Some applications use the entry point argument as an index into a parameter table to get task-specific parameters.

A new entry point argument may be used to distinguish between the initial *rtems_task_start()* (page 118) of the task and any ensuing calls to *rtems_task_restart()* (page 120) of the task. This can be beneficial in deleting a task. Instead of deleting a task using the *rtems_task_delete()* (page 122) directive, a task can delete another task by restarting that task, and allowing that task to release resources back to RTEMS and then delete itself.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may change the priority of a task. This may cause the calling task to be preempted.
- The directive may unblock a task. This may cause the calling task to be preempted.

### 7.4.7 rtems_task_delete()

Deletes the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

DESCRIPTION:

This directive deletes the task, either the calling task or another task, as specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_INCORRECT_STATE**
The task termination procedure was started, however, waiting for the terminating task would have resulted in a deadlock.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

NOTES:

The task deletion is done in several steps. Firstly, the task is marked as terminating. While the task life of the terminating task is protected, it executes normally until it disables the task life protection or it deletes itself. A terminating task will eventually stop its normal execution and start its termination procedure. The procedure executes in the context of the terminating task. The task termination procedure involves the destruction of POSIX key values and running the task termination user extensions. Once complete the execution of the task is stopped and task-specific resources are reclaimed by the system, such as the stack memory, any allocated delay or timeout timers, the *TCB*, and, if the task is `RTEMS_FLOATING_POINT`, its floating point context area. RTEMS explicitly does not reclaim the following resources: region segments, partition buffers, semaphores, timers, or rate monotonic periods.

A task is responsible for releasing its resources back to RTEMS before deletion. To insure proper deallocation of resources, a task should not be deleted unless it is unable to execute or does not hold any RTEMS resources. If a task holds RTEMS resources, the task should be allowed to deallocate its resources before deletion. A task can be directed to release its resources and delete itself by restarting it with a special argument or by sending it a message, an event, or a signal.

Deletion of the calling task (RTEMS_SELF) will force RTEMS to select another task to execute.

When a task deletes another task, the calling task waits until the task termination procedure of the task being deleted has completed. The terminating task inherits the *eligible priorities* of the calling task.

When a global task is deleted, the task identifier must be transmitted to every node in the system for deletion from the local copy of the global object table.

The task must reside on the local node, even if the task was created with the RTEMS_GLOBAL attribute.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 7.4.8 rtems_task_exit()

Deletes the calling task.

CALLING SEQUENCE:

```
void rtems_task_exit( void );
```

DESCRIPTION:

This directive deletes the calling task.

NOTES:

The directive is an optimized variant of the following code sequences, see also *rtems_task_delete()* (page 122):

```
#include <pthread.h>
#include <rtems.h>

void classic_delete_self( void )
{
  (void) rtems_task_delete( RTEMS_SELF );
}

void posix_delete_self( void )
{
  (void) pthread_detach( pthread_self() );
  (void) pthread_exit( NULL);
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive will not return to the caller.
- While thread dispatching is disabled, if the directive performs a thread dispatch, then the fatal error with the fatal source *INTERNAL_ERROR_CORE* (page 518) and the fatal code *INTERNAL_ERROR_BAD_THREAD_DISPATCH_DISABLE_LEVEL* (page 519) will occur.

### 7.4.9 rtems_task_suspend()

Suspends the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_suspend( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

DESCRIPTION:

This directive suspends the task specified by `id` from further execution by placing it in the suspended state. This state is additive to any other blocked state that the task may already be in. The task will not execute again until another task issues the *rtems_task_resume()* (page 126) directive for this task and any blocked state has been removed. The *rtems_task_restart()* (page 120) directive will also remove the suspended state.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_ALREADY_SUSPENDED**
The task was already suspended.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

NOTES:

The requesting task can suspend itself for example by specifying `RTEMS_SELF` as `id`. In this case, the task will be suspended and a successful return code will be returned when the task is resumed.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 7.4.10 rtems_task_resume()

Resumes the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_resume( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the task identifier.

DESCRIPTION:

This directive removes the task specified by `id` from the suspended state. If the task is in the ready state after the suspension is removed, then it will be scheduled to run. If the task is still in a blocked state after the suspension is removed, then it will remain in that blocked state.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INCORRECT_STATE**
The task was not suspended.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 7.4.11 rtems_task_is_suspended()

Checks if the task is suspended.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_is_suspended( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

DESCRIPTION:

This directive returns a status code indicating whether or not the task specified by `id` is currently suspended.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The task was **not** suspended.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_ALREADY_SUSPENDED**
The task was suspended.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 7.4.12 rtems_task_set_priority()

Sets the real priority or gets the current priority of the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_priority(
  rtems_id             id,
  rtems_task_priority  new_priority,
  rtems_task_priority *old_priority
);
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**new_priority**
This parameter is the new real priority or `RTEMS_CURRENT_PRIORITY` to get the current priority.

**old_priority**
This parameter is the pointer to an *rtems_task_priority* (page 57) object. When the directive call is successful, the current or previous priority of the task with respect to its *home scheduler* will be stored in this object.

DESCRIPTION:

This directive manipulates the priority of the task specified by `id`. When `new_priority` is not equal to `RTEMS_CURRENT_PRIORITY`, the specified task's previous priority is returned in `old_priority`. When `new_priority` is `RTEMS_CURRENT_PRIORITY`, the specified task's current priority is returned in `old_priority`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `old_priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INVALID_PRIORITY**
The task priority specified in `new_priority` was invalid with respect to the *home scheduler* of the task.

NOTES:

Valid priorities range from one to a maximum value which depends on the configured scheduler. The lower the priority value the higher is the importance of the task.

If the task is currently holding any binary semaphores which use a locking protocol, then the task's priority cannot be lowered immediately. If the task's priority were lowered immediately, then this could violate properties of the locking protocol and may result in priority inversion.

The requested lowering of the task's priority will occur when the task has released all binary semaphores which make the task more important. The task's priority can be increased regardless of the task's use of binary semaphores with locking protocols.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may change the priority of a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 7.4.13 rtems_task_get_priority()

Gets the current priority of the task with respect to the scheduler.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_get_priority(
  rtems_id             task_id,
  rtems_id             scheduler_id,
  rtems_task_priority *priority
);
```

PARAMETERS:

**task_id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**scheduler_id**
This parameter is the scheduler identifier.

**priority**
This parameter is the pointer to an *rtems_task_priority* (page 57) object. When the directive call is successful, the current priority of the task with respect to the specified scheduler will be stored in this object.

DESCRIPTION:

This directive returns the current priority in `priority` of the task specified by `task_id` with respect to the scheduler specified by `scheduler_id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `task_id`.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_NOT_DEFINED**
The task had no priority with respect to the scheduler.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

NOTES:

The current priority reflects temporary priority adjustments due to locking protocols, the rate-monotonic period objects on some schedulers such as EDF, and the POSIX sporadic server.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 7.4.14 rtems_task_mode()

Gets and optionally sets the mode of the calling task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_mode(
  rtems_mode  mode_set,
  rtems_mode  mask,
  rtems_mode *previous_mode_set
);
```

PARAMETERS:

**mode_set**
This parameter is the mode set to apply to the calling task. When `mask` is set to `RTEMS_CURRENT_MODE`, the value of this parameter is ignored. Only modes requested by `mask` are applied to the calling task.

**mask**
This parameter is the mode mask which specifies which modes in `mode_set` are applied to the calling task. When the value is `RTEMS_CURRENT_MODE`, the mode of the calling task is not changed.

**previous_mode_set**
This parameter is the pointer to an `rtems_mode` object. When the directive call is successful, the mode of the task before any mode changes done by the directive call will be stored in this object.

DESCRIPTION:

This directive queries and optionally manipulates the execution mode of the calling task. A task's execution mode enables and disables preemption, timeslicing, asynchronous signal processing, as well as specifying the interrupt level. To modify an execution mode, the mode class(es) to be changed must be specified in the `mask` parameter and the desired mode(s) must be specified in the `mode_set` parameter.

A task can obtain its current execution mode, without modifying it, by calling this directive with a `mask` value of `RTEMS_CURRENT_MODE`.

The **mode set** specified in `mode_set` is built through a *bitwise or* of the mode constants described below. Not all combinations of modes are allowed. Some modes are mutually exclusive. If mutually exclusive modes are combined, the behaviour is undefined. Default task modes can be selected by using the `RTEMS_DEFAULT_MODES` constant. The task mode set defines

- the preemption mode of the task: `RTEMS_PREEMPT` (default) or `RTEMS_NO_PREEMPT`,
- the timeslicing mode of the task: `RTEMS_TIMESLICE` or `RTEMS_NO_TIMESLICE` (default),
- the *ASR* processing mode of the task: `RTEMS_ASR` (default) or `RTEMS_NO_ASR`,
- the interrupt level of the task: `RTEMS_INTERRUPT_LEVEL()` with a default of `RTEMS_INTERRUPT_LEVEL( 0 )` which is associated with enabled interrupts.

The **mode mask** specified in `mask` is built through a *bitwise or* of the mode mask constants described below.

When the RTEMS_PREEMPT_MASK is set in mask, the **preemption mode** of the calling task is

- enabled by using the RTEMS_PREEMPT mode constant in mode_set and
- disabled by using the RTEMS_NO_PREEMPT mode constant in mode_set.

When the RTEMS_TIMESLICE_MASK is set in mask, the **timeslicing mode** of the calling task is

- enabled by using the RTEMS_TIMESLICE mode constant in mode_set and
- disabled by using the RTEMS_NO_TIMESLICE mode constant in mode_set.

Enabling timeslicing has no effect if preemption is disabled. For a task to be timesliced, that task must have both preemption and timeslicing enabled.

When the RTEMS_ASR_MASK is set in mask, the **ASR processing mode** of the calling task is

- enabled by using the RTEMS_ASR mode constant in mode_set and
- disabled by using the RTEMS_NO_ASR mode constant in mode_set.

When the RTEMS_INTERRUPT_MASK is set in mask, **interrupts** of the calling task are

- enabled by using the RTEMS_INTERRUPT_LEVEL() mode macro with a value of zero (0) in mode_set and
- disabled up to the specified level by using the RTEMS_INTERRUPT_LEVEL() mode macro with a positive value in mode_set.

An interrupt level of zero is associated with enabled interrupts on all target processors. The interrupt level portion of the task mode supports a maximum of 256 interrupt levels. These levels are mapped onto the interrupt levels actually supported by the target processor in a processor dependent fashion.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_NOT_IMPLEMENTED**
The RTEMS_NO_PREEMPT was set in mode_set and setting the preemption mode was requested by RTEMS_PREEMPT_MASK in mask and the system configuration had no implementation for this mode.

**RTEMS_NOT_IMPLEMENTED**
The RTEMS_INTERRUPT_LEVEL() was set to a positive level in mode_set and setting the interrupt level was requested by RTEMS_INTERRUPT_MASK in mask and the system configuration had no implementation for this mode.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- When the directive enables preemption for the calling task, another task may preempt the calling task.
- While thread dispatching is disabled, if the directive performs a thread dispatch, then the fatal error with the fatal source *INTERNAL_ERROR_CORE* (page 518) and the fatal code *INTERNAL_ERROR_BAD_THREAD_DISPATCH_DISABLE_LEVEL* (page 519) will occur.

### 7.4.15 rtems_task_wake_after()

Wakes up after a count of *clock ticks* have occurred or yields the processor.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_after( rtems_interval ticks );
```

PARAMETERS:

**ticks**
This parameter is the count of *clock ticks* to delay the task or `RTEMS_YIELD_PROCESSOR` to yield the processor.

DESCRIPTION:

This directive blocks the calling task for the specified `ticks` count of clock ticks if the value is not equal to `RTEMS_YIELD_PROCESSOR`. When the requested count of ticks have occurred, the task is made ready. The clock tick directives automatically update the delay period. The calling task may give up the processor and remain in the ready state by specifying a value of `RTEMS_YIELD_PROCESSOR` in `ticks`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

NOTES:

Setting the system date and time with the *rtems_clock_set()* (page 243) directive and similar directives which set *CLOCK_REALTIME* have no effect on a *rtems_task_wake_after()* (page 134) blocked task. The delay until first clock tick will never be a whole clock tick interval since this directive will never execute exactly on a clock tick. Applications requiring use of a clock (*CLOCK_REALTIME* or *CLOCK_MONOTONIC*) instead of clock ticks should make use of clock_nanosleep().

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive requires a *Clock Driver*.
- While thread dispatching is disabled, if the directive performs a thread dispatch, then the fatal error with the fatal source *INTERNAL_ERROR_CORE* (page 518) and the fatal code *INTERNAL_ERROR_BAD_THREAD_DISPATCH_DISABLE_LEVEL* (page 519) will occur.

### 7.4.16 rtems_task_wake_when()

Wakes up when specified.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_when( const rtems_time_of_day *time_buffer );
```

PARAMETERS:

**time_buffer**
This parameter is the date and time to wake up.

DESCRIPTION:

This directive blocks a task until the date and time specified in `time_buffer`. At the requested date and time, the calling task will be unblocked and made ready to execute.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_NOT_DEFINED**
The system date and time was not set.

**RTEMS_INVALID_ADDRESS**
The `time_buffer` parameter was NULL.

**RTEMS_INVALID_CLOCK**
The time of day was invalid.

NOTES:

The ticks portion of `time_buffer` structure is ignored. The timing granularity of this directive is a second.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive requires a *Clock Driver*.
- While thread dispatching is disabled, if the directive performs a thread dispatch, then the fatal error with the fatal source *INTERNAL_ERROR_CORE* (page 518) and the fatal code *INTERNAL_ERROR_BAD_THREAD_DISPATCH_DISABLE_LEVEL* (page 519) will occur.

### 7.4.17 rtems_task_get_scheduler()

Gets the home scheduler of the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_get_scheduler(
  rtems_id  task_id,
  rtems_id *scheduler_id
);
```

PARAMETERS:

**task_id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**scheduler_id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the *home scheduler* of the task will be stored in this object.

DESCRIPTION:

This directive returns the identifier of the *home scheduler* of the task specified by `task_id` in `scheduler_id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `scheduler_id` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `task_id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 7.4.18 rtems_task_set_scheduler()

Sets the home scheduler for the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_scheduler(
  rtems_id            task_id,
  rtems_id            scheduler_id,
  rtems_task_priority priority
);
```

PARAMETERS:

**task_id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**scheduler_id**
This parameter is the scheduler identifier of the new *home scheduler* for the task specified by `task_id`.

**priority**
This parameter is the new real priority for the task with respect to the scheduler specified by `scheduler_id`.

DESCRIPTION:

This directive sets the *home scheduler* to the scheduler specified by `scheduler_id` for the task specified by `task_id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_PRIORITY**
The *task priority* specified by `priority` was invalid with respect to the scheduler specified by `scheduler_id`.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `task_id`.

**RTEMS_RESOURCE_IN_USE**
The task specified by `task_id` was enqueued on a *wait queue*.

**RTEMS_RESOURCE_IN_USE**
The task specified by `task_id` had a *current priority* which consisted of more than the *real priority*.

**RTEMS_RESOURCE_IN_USE**
The task specified by `task_id` had a *helping scheduler*.

**RTEMS_RESOURCE_IN_USE**
The task specified by `task_id` was pinned.

**RTEMS_UNSATISFIED**
The scheduler specified by `scheduler_id` owned no processor.

**RTEMS_UNSATISFIED**
The scheduler specified by `scheduler_id` did not support the affinity set of the task specified by `task_id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may change the priority of a task. This may cause the calling task to be preempted.

### 7.4.19 rtems_task_get_affinity()

Gets the processor affinity of the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_get_affinity(
  rtems_id   id,
  size_t     cpusetsize,
  cpu_set_t *cpuset
);
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**cpusetsize**
This parameter is the size of the processor set referenced by `cpuset` in bytes.

**cpuset**
This parameter is the pointer to a `cpu_set_t` object. When the directive call is successful, the processor affinity set of the task will be stored in this object. A set bit in the processor set means that the corresponding processor is in the processor affinity set of the task, otherwise the bit is cleared.

DESCRIPTION:

This directive returns the processor affinity of the task in `cpuset` of the task specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `cpuset` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INVALID_SIZE**
The size specified by `cpusetsize` of the processor set was too small for the processor affinity set of the task.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.

- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 7.4.20 rtems_task_set_affinity()

Sets the processor affinity of the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_affinity(
  rtems_id         id,
  size_t           cpusetsize,
  const cpu_set_t *cpuset
);
```

PARAMETERS:

**id**
This parameter is the task identifier. The constant `RTEMS_SELF` may be used to specify the calling task.

**cpusetsize**
This parameter is the size of the processor set referenced by `cpuset` in bytes.

**cpuset**
This parameter is the pointer to a `cpu_set_t` object. The processor set defines the new processor affinity set of the task. A set bit in the processor set means that the corresponding processor shall be in the processor affinity set of the task, otherwise the bit shall be cleared.

DESCRIPTION:

This directive sets the processor affinity of the task specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `cpuset` parameter was NULL.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_INVALID_NUMBER**
The referenced processor set was not a valid new processor affinity set for the task.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The task resided on a remote node.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

- The directive may change the processor affinity of a task. This may cause the calling task to be preempted.

### 7.4.21 rtems_task_iterate()

Iterates over all tasks and invokes the visitor routine for each task.

CALLING SEQUENCE:

```
void rtems_task_iterate( rtems_task_visitor visitor, void *arg );
```

PARAMETERS:

**visitor**
This parameter is the visitor routine invoked for each task.

**arg**
This parameter is the argument passed to each visitor routine invocation during the iteration.

DESCRIPTION:

This directive iterates over all tasks in the system. This operation covers all tasks of all APIs. The user should be careful in accessing the contents of the *TCB*. The visitor argument `arg` is passed to all invocations of `visitor` in addition to the TCB. The iteration stops immediately in case the visitor routine returns true.

NOTES:

The visitor routine is invoked while owning the objects allocator lock. It is allowed to perform blocking operations in the visitor routine, however, care must be taken so that no deadlocks via the object allocator lock can occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 7.4.22 RTEMS_TASK_STORAGE_SIZE()

Gets the recommended task storage area size for the size and task attributes.

CALLING SEQUENCE:

```
size_t RTEMS_TASK_STORAGE_SIZE( size_t size, rtems_attribute attributes );
```

PARAMETERS:

**size**
This parameter is the size dedicated to the task stack and thread-local storage in bytes.

**attributes**
This parameter is the attribute set of the task using the storage area.

RETURN VALUES:

Returns the recommended task storage area size calculated from the input parameters.

## 7.5 Deprecated Directives

### 7.5.1 ITERATE_OVER_ALL_THREADS - Iterate Over Tasks

**Warning**

This directive is deprecated. Its use is unsafe. Use rtems_task_iterate instead.

**CALLING SEQUENCE:**

```
typedef void (*rtems_per_thread_routine)(Thread_Control *the_thread);
void rtems_iterate_over_all_threads(
    rtems_per_thread_routine routine
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This directive iterates over all of the existant threads in the system and invokes `routine` on each of them. The user should be careful in accessing the contents of `the_thread`.

This routine is intended for use in diagnostic utilities and is not intented for routine use in an operational system.

**NOTES:**

There is **no protection** while this routine is called. The thread control block may be in an inconsistent state or may change due to interrupts or activity on other processors.

## 7.6 Removed Directives

### 7.6.1 TASK_GET_NOTE - Get task notepad entry

**Warning**

This directive was removed in RTEMS 5.1.

**CALLING SEQUENCE:**

```
rtems_status_code rtems_task_get_note(
  rtems_id  id,
  uint32_t  notepad,
  uint32_t *note
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | note value obtained successfully |
| RTEMS_INVALID_ADDRESS | note parameter is NULL |
| RTEMS_INVALID_ID | invalid task id |
| RTEMS_INVALID_NUMBER | invalid notepad location |

**DESCRIPTION:**

This directive returns the note contained in the notepad location of the task specified by id.

**NOTES:**

This directive will not cause the running task to be preempted.

If id is set to RTEMS_SELF, the calling task accesses its own notepad.

The sixteen notepad locations can be accessed using the constants RTEMS_NOTEPAD_0 through RTEMS_NOTEPAD_15.

Getting a note of a global task which does not reside on the local node will generate a request to the remote node to obtain the notepad entry of the specified task.

### 7.6.2 TASK_SET_NOTE - Set task notepad entry

**Warning**

This directive was removed in RTEMS 5.1.

**CALLING SEQUENCE:**

```
rtems_status_code rtems_task_set_note(
  rtems_id  id,
  uint32_t  notepad,
  uint32_t  note
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | note set successfully |
| RTEMS_INVALID_ID | invalid task id |
| RTEMS_INVALID_NUMBER | invalid notepad location |

**DESCRIPTION:**

This directive sets the notepad entry for the task specified by id to the value note.

**NOTES:**

If `id` is set to `RTEMS_SELF`, the calling task accesses its own notepad.

This directive will not cause the running task to be preempted.

The sixteen notepad locations can be accessed using the constants `RTEMS_NOTEPAD_0` through `RTEMS_NOTEPAD_15`.

Setting a note of a global task which does not reside on the local node will generate a request to the remote node to set the notepad entry of the specified task.

### 7.6.3 TASK_VARIABLE_ADD - Associate per task variable

**Warning**

This directive was removed in RTEMS 5.1.

**CALLING SEQUENCE:**

```
rtems_status_code rtems_task_variable_add(
    rtems_id  tid,
    void    **task_variable,
    void    (*dtor)(void *)
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | per task variable added successfully |
| RTEMS_INVALID_ADDRESS | task_variable is NULL |
| RTEMS_INVALID_ID | invalid task id |
| RTEMS_NO_MEMORY | invalid task id |
| RTEMS_ILLEGAL_ON_REMOTE_OBJECT | not supported on remote tasks |

**DESCRIPTION:**

This directive adds the memory location specified by the ptr argument to the context of the given task. The variable will then be private to the task. The task can access and modify the variable, but the modifications will not appear to other tasks, and other tasks' modifications to that variable will not affect the value seen by the task. This is accomplished by saving and restoring the variable's value each time a task switch occurs to or from the calling task. If the dtor argument is non-NULL it specifies the address of a 'destructor' function which will be called when the task is deleted. The argument passed to the destructor function is the task's value of the variable.

**NOTES:**

Task variables increase the context switch time to and from the tasks that own them so it is desirable to minimize the number of task variables. One efficient method is to have a single task variable that is a pointer to a dynamically allocated structure containing the task's private 'global' data. In this case the destructor function could be 'free'.

Per-task variables are disabled in SMP configurations and this service is not available.

### 7.6.4 TASK_VARIABLE_GET - Obtain value of a per task variable

**Warning**

This directive was removed in RTEMS 5.1.

**CALLING SEQUENCE:**

```
rtems_status_code rtems_task_variable_get(
    rtems_id  tid,
    void    **task_variable,
    void    **task_variable_value
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | per task variable obtained successfully |
| RTEMS_INVALID_ADDRESS | task_variable is NULL |
| RTEMS_INVALID_ADDRESS | task_variable_value is NULL |
| RTEMS_INVALID_ADDRESS | task_variable is not found |
| RTEMS_NO_MEMORY | invalid task id |
| RTEMS_ILLEGAL_ON_REMOTE_OBJECT | not supported on remote tasks |

**DESCRIPTION:**

This directive looks up the private value of a task variable for a specified task and stores that value in the location pointed to by the result argument. The specified task is usually not the calling task, which can get its private value by directly accessing the variable.

**NOTES:**

If you change memory which task_variable_value points to, remember to declare that memory as volatile, so that the compiler will optimize it correctly. In this case both the pointer task_variable_value and data referenced by task_variable_value should be considered volatile.

Per-task variables are disabled in SMP configurations and this service is not available.

### 7.6.5 TASK_VARIABLE_DELETE - Remove per task variable

**Warning**

This directive was removed in RTEMS 5.1.

**CALLING SEQUENCE:**

```
rtems_status_code rtems_task_variable_delete(
    rtems_id  id,
    void    **task_variable
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | per task variable deleted successfully |
| RTEMS_INVALID_ID | invalid task id |
| RTEMS_NO_MEMORY | invalid task id |
| RTEMS_INVALID_ADDRESS | task_variable is NULL |
| RTEMS_ILLEGAL_ON_REMOTE_OBJECT | not supported on remote tasks |

**DESCRIPTION:**

This directive removes the given location from a task's context.

**NOTES:**

Per-task variables are disabled in SMP configurations and this service is not available.

CHAPTER

**EIGHT**

# INTERRUPT MANAGER

## 8.1 Introduction

Any real-time executive must provide a mechanism for quick response to externally generated interrupts to satisfy the critical time constraints of the application. The Interrupt Manager provides this mechanism for RTEMS. This manager permits quick interrupt response times by providing the critical ability to alter task execution which allows a task to be preempted upon exit from an ISR. The directives provided by the Interrupt Manager are:

- *rtems_interrupt_catch()* (page 162) - Establishes an interrupt service routine.
- *rtems_interrupt_disable()* (page 164) - Disables the maskable interrupts on the current processor.
- *rtems_interrupt_enable()* (page 166) - Restores the previous interrupt level on the current processor.
- *rtems_interrupt_flash()* (page 167) - Flashes interrupts on the current processor.
- *rtems_interrupt_local_disable()* (page 168) - Disables the maskable interrupts on the current processor.
- *rtems_interrupt_local_enable()* (page 170) - Restores the previous interrupt level on the current processor.
- *rtems_interrupt_is_in_progress()* (page 171) - Checks if an ISR is in progress on the current processor.
- *rtems_interrupt_lock_initialize()* (page 172) - Initializes the ISR lock.
- *rtems_interrupt_lock_destroy()* (page 173) - Destroys the ISR lock.
- *rtems_interrupt_lock_acquire()* (page 174) - Acquires the ISR lock.
- *rtems_interrupt_lock_release()* (page 176) - Releases the ISR lock.
- *rtems_interrupt_lock_acquire_isr()* (page 177) - Acquires the ISR lock from within an ISR.
- *rtems_interrupt_lock_release_isr()* (page 178) - Releases the ISR lock from within an ISR.
- *rtems_interrupt_lock_interrupt_disable()* (page 179) - Disables maskable interrupts on the current processor.
- *RTEMS_INTERRUPT_LOCK_DECLARE()* (page 180) - Declares an ISR lock object.
- *RTEMS_INTERRUPT_LOCK_DEFINE()* (page 181) - Defines an ISR lock object.
- *RTEMS_INTERRUPT_LOCK_INITIALIZER()* (page 182) - Statically initializes an ISR lock object.
- *RTEMS_INTERRUPT_LOCK_MEMBER()* (page 183) - Defines an ISR lock member.
- *RTEMS_INTERRUPT_LOCK_REFERENCE()* (page 184) - Defines an ISR lock object reference.
- *RTEMS_INTERRUPT_ENTRY_INITIALIZER()* (page 185) - Statically initializes an interrupt entry object.
- *rtems_interrupt_entry_initialize()* (page 186) - Initializes the interrupt entry.
- *rtems_interrupt_entry_install()* (page 187) - Installs the interrupt entry at the interrupt vector.

- *rtems_interrupt_entry_remove()* (page 189) - Removes the interrupt entry from the interrupt vector.
- *rtems_interrupt_handler_install()* (page 190) - Installs the interrupt handler routine and argument at the interrupt vector.
- *rtems_interrupt_handler_remove()* (page 192) - Removes the interrupt handler routine and argument from the interrupt vector.
- *rtems_interrupt_vector_is_enabled()* (page 193) - Checks if the interrupt vector is enabled.
- *rtems_interrupt_vector_enable()* (page 194) - Enables the interrupt vector.
- *rtems_interrupt_vector_disable()* (page 195) - Disables the interrupt vector.
- *rtems_interrupt_is_pending()* (page 196) - Checks if the interrupt is pending.
- *rtems_interrupt_raise()* (page 198) - Raises the interrupt vector.
- *rtems_interrupt_raise_on()* (page 199) - Raises the interrupt vector on the processor.
- *rtems_interrupt_clear()* (page 200) - Clears the interrupt vector.
- *rtems_interrupt_get_priority()* (page 201) - Gets the priority of the interrupt vector.
- *rtems_interrupt_set_priority()* (page 202) - Sets the priority of the interrupt vector.
- *rtems_interrupt_get_affinity()* (page 204) - Gets the processor affinity set of the interrupt vector.
- *rtems_interrupt_set_affinity()* (page 205) - Sets the processor affinity set of the interrupt vector.
- *rtems_interrupt_get_attributes()* (page 207) - Gets the attributes of the interrupt vector.
- *rtems_interrupt_handler_iterate()* (page 208) - Iterates over all interrupt handler installed at the interrupt vector.
- *rtems_interrupt_server_initialize()* (page 210) - Initializes the interrupt server tasks.
- *rtems_interrupt_server_create()* (page 212) - Creates an interrupt server.
- *rtems_interrupt_server_handler_install()* (page 213) - Installs the interrupt handler routine and argument at the interrupt vector on the interrupt server.
- *rtems_interrupt_server_handler_remove()* (page 215) - Removes the interrupt handler routine and argument from the interrupt vector and the interrupt server.
- *rtems_interrupt_server_set_affinity()* (page 216) - Sets the processor affinity of the interrupt server.
- *rtems_interrupt_server_delete()* (page 218) - Deletes the interrupt server.
- *rtems_interrupt_server_suspend()* (page 219) - Suspends the interrupt server.
- *rtems_interrupt_server_resume()* (page 220) - Resumes the interrupt server.
- *rtems_interrupt_server_move()* (page 221) - Moves the interrupt handlers installed at the interrupt vector and the source interrupt server to the destination interrupt server.
- *rtems_interrupt_server_handler_iterate()* (page 222) - Iterates over all interrupt handler installed at the interrupt vector and interrupt server.
- *rtems_interrupt_server_entry_initialize()* (page 224) - Initializes the interrupt server entry.

- *rtems_interrupt_server_action_prepend()* (page 225) - Prepends the interrupt server action to the list of actions of the interrupt server entry.
- *rtems_interrupt_server_entry_destroy()* (page 227) - Destroys the interrupt server entry.
- *rtems_interrupt_server_entry_submit()* (page 228) - Submits the interrupt server entry to be serviced by the interrupt server.
- *rtems_interrupt_server_entry_move()* (page 229) - Moves the interrupt server entry to the interrupt server.
- *rtems_interrupt_server_request_initialize()* (page 230) - Initializes the interrupt server request.
- *rtems_interrupt_server_request_set_vector()* (page 231) - Sets the interrupt vector in the interrupt server request.
- *rtems_interrupt_server_request_destroy()* (page 232) - Destroys the interrupt server request.
- *rtems_interrupt_server_request_submit()* (page 233) - Submits the interrupt server request to be serviced by the interrupt server.

## 8.2 Background

### 8.2.1 Processing an Interrupt

The interrupt manager allows the application to connect a function to a hardware interrupt vector. When an interrupt occurs, the processor will automatically vector to RTEMS. RTEMS saves and restores all registers which are not preserved by the normal C calling convention for the target processor and invokes the user's ISR. The user's ISR is responsible for processing the interrupt, clearing the interrupt if necessary, and device specific manipulation.

The `rtems_interrupt_catch` directive connects a procedure to an interrupt vector. The vector number is managed using the `rtems_vector_number` data type.

The interrupt service routine is assumed to abide by these conventions and have a prototype similar to the following:

```
rtems_isr user_isr(
  rtems_vector_number vector
);
```

The vector number argument is provided by RTEMS to allow the application to identify the interrupt source. This could be used to allow a single routine to service interrupts from multiple instances of the same device. For example, a single routine could service interrupts from multiple serial ports and use the vector number to identify which port requires servicing.

To minimize the masking of lower or equal priority level interrupts, the ISR should perform the minimum actions required to service the interrupt. Other non-essential actions should be handled by application tasks. Once the user's ISR has completed, it returns control to the RTEMS interrupt manager which will perform task dispatching and restore the registers saved before the ISR was invoked.

The RTEMS interrupt manager guarantees that proper task scheduling and dispatching are performed at the conclusion of an ISR. A system call made by the ISR may have readied a task of higher priority than the interrupted task. Therefore, when the ISR completes, the postponed dispatch processing must be performed. No dispatch processing is performed as part of directives which have been invoked by an ISR.

Applications must adhere to the following rule if proper task scheduling and dispatching is to be performed:

> **Note**
>
> The interrupt manager must be used for all ISRs which may be interrupted by the highest priority ISR which invokes an RTEMS directive.

Consider a processor which allows a numerically low interrupt level to interrupt a numerically greater interrupt level. In this example, if an RTEMS directive is used in a level 4 ISR, then all ISRs which execute at levels 0 through 4 must use the interrupt manager.

Interrupts are nested whenever an interrupt occurs during the execution of another ISR. RTEMS supports efficient interrupt nesting by allowing the nested ISRs to terminate without performing any dispatch processing. Only when the outermost ISR terminates will the postponed dispatching occur.

### 8.2.2 RTEMS Interrupt Levels

Many processors support multiple interrupt levels or priorities. The exact number of interrupt levels is processor dependent. RTEMS internally supports 256 interrupt levels which are mapped to the processor's interrupt levels. For specific information on the mapping between RTEMS and the target processor's interrupt levels, refer to the Interrupt Processing chapter of the Applications Supplement document for a specific target processor.

### 8.2.3 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables all maskable interrupts before the execution of the section and restores them to the previous level upon completion of the section. RTEMS has been optimized to ensure that interrupts are disabled for a minimum length of time. The maximum length of time interrupts are disabled by RTEMS is processor dependent and is detailed in the Timing Specification chapter of the Applications Supplement document for a specific target processor.

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

## 8.3 Operations

### 8.3.1 Establishing an ISR

The `rtems_interrupt_catch` directive establishes an ISR for the system. The address of the ISR and its associated CPU vector number are specified to this directive. This directive installs the RTEMS interrupt wrapper in the processor's Interrupt Vector Table and the address of the user's ISR in the RTEMS' Vector Table. This directive returns the previous contents of the specified vector in the RTEMS' Vector Table.

### 8.3.2 Directives Allowed from an ISR

Using the interrupt manager ensures that RTEMS knows when a directive is being called from an ISR. The ISR may then use system calls to synchronize itself with an application task. The synchronization may involve messages, events or signals being passed by the ISR to the desired task. Directives invoked by an ISR must operate only on objects which reside on the local node. The following is a list of RTEMS system calls that may be made from an ISR:

- Task Management Although it is acceptable to operate on the RTEMS_SELF task (e.g. the currently executing task), while in an ISR, this will refer to the interrupted task. Most of the time, it is an application implementation error to use RTEMS_SELF from an ISR.
  - rtems_task_suspend
  - rtems_task_resume
- Interrupt Management
  - rtems_interrupt_enable
  - rtems_interrupt_disable
  - rtems_interrupt_flash
  - rtems_interrupt_lock_acquire
  - rtems_interrupt_lock_release
  - rtems_interrupt_lock_acquire_isr
  - rtems_interrupt_lock_release_isr
  - rtems_interrupt_is_in_progress
  - rtems_interrupt_catch
- Clock Management
  - rtems_clock_set
  - rtems_clock_get_tod
  - rtems_clock_get_tod_timeval
  - rtems_clock_get_seconds_since_epoch
  - rtems_clock_get_ticks_per_second
  - rtems_clock_get_ticks_since_boot
  - rtems_clock_get_uptime
- Timer Management

  - rtems_timer_cancel
  - rtems_timer_reset
  - rtems_timer_fire_after
  - rtems_timer_fire_when
  - rtems_timer_server_fire_after
  - rtems_timer_server_fire_when
- Event Management
  - rtems_event_send
  - rtems_event_system_send
  - rtems_event_transient_send
- Semaphore Management
  - rtems_semaphore_release
- Message Management
  - rtems_message_queue_broadcast
  - rtems_message_queue_send
  - rtems_message_queue_urgent
- Signal Management
  - rtems_signal_send
- Dual-Ported Memory Management
  - rtems_port_external_to_internal
  - rtems_port_internal_to_external
- IO Management The following services are safe to call from an ISR if and only if the device driver service invoked is also safe. The IO Manager itself is safe but the invoked driver entry point may or may not be.
  - rtems_io_initialize
  - rtems_io_open
  - rtems_io_close
  - rtems_io_read
  - rtems_io_write
  - rtems_io_control
- Fatal Error Management
  - rtems_fatal
  - rtems_fatal_error_occurred
- Multiprocessing
  - rtems_multiprocessing_announce

## 8.4 Directives

This section details the directives of the Interrupt Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 8.4.1 rtems_interrupt_catch()

Establishes an interrupt service routine.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_catch(
  rtems_isr_entry     new_isr_handler,
  rtems_vector_number vector,
  rtems_isr_entry    *old_isr_handler
);
```

PARAMETERS:

**new_isr_handler**
This parameter is the new interrupt service routine.

**vector**
This parameter is the interrupt vector number.

**old_isr_handler**
This parameter is the pointer to an *rtems_isr_entry* (page 47) object. When the directive call is successful, the previous interrupt service routine established for this interrupt vector will be stored in this object.

DESCRIPTION:

This directive establishes an interrupt service routine (ISR) for the interrupt specified by the `vector` number. The `new_isr_handler` parameter specifies the entry point of the ISR. The entry point of the previous ISR for the specified vector is returned in `old_isr_handler`.

To release an interrupt vector, pass the old handler's address obtained when the vector was first capture.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The interrupt vector number was illegal.

**RTEMS_INVALID_ADDRESS**
The `new_isr_handler` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `old_isr_handler` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

- The directive will not cause the calling task to be preempted.
- The directive is only available where the *target architecture* support enabled simple vectored interrupts.

### 8.4.2 rtems_interrupt_disable()

Disables the maskable interrupts on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_disable( rtems_interrupt_level isr_cookie );
```

PARAMETERS:

**isr_cookie**
This parameter is a variable of type *rtems_interrupt_level* (page 44) which will be used to save the previous interrupt level.

DESCRIPTION:

This directive disables all maskable interrupts on the current processor and returns the previous interrupt level in `isr_cookie`.

NOTES:

A later invocation of the *rtems_interrupt_enable()* (page 166) directive should be used to restore the previous interrupt level.

This directive is implemented as a macro which sets the `isr_cookie` parameter.

```
#include <rtems.h>

void local_critical_section( void )
{
  rtems_interrupt_level level;

  // Please note that the rtems_interrupt_disable() is a macro.  The
  // previous interrupt level (before the maskable interrupts are
  // disabled) is returned here in the level macro parameter.  This
  // would be wrong:
  //
  // rtems_interrupt_disable( &level );
  rtems_interrupt_disable( level );

  // Here is the critical section: maskable interrupts are disabled

  {
    rtems_interrupt_level nested_level;

    rtems_interrupt_disable( nested_level );

    // Here is a nested critical section

    rtems_interrupt_enable( nested_level );
  }

```

(continues on next page)

(continued from previous page)

```
  // Maskable interrupts are still disabled

  rtems_interrupt_enable( level );
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- Where the system was built with SMP support enabled, the directive is not available. Its use will result in compiler warnings and linker errors. The *rtems_interrupt_local_disable()* (page 168) and *rtems_interrupt_local_enable()* (page 170) directives are available in all build configurations.

### 8.4.3 rtems_interrupt_enable()

Restores the previous interrupt level on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_enable( rtems_interrupt_level isr_cookie );
```

PARAMETERS:

**isr_cookie**
This parameter is the previous interrupt level to restore. The value must be obtained by a previous call to *rtems_interrupt_disable()* (page 164) or *rtems_interrupt_flash()* (page 167).

DESCRIPTION:

This directive restores the interrupt level specified by `isr_cookie` on the current processor.

NOTES:

The `isr_cookie` parameter value must be obtained by a previous call to *rtems_interrupt_disable()* (page 164) or *rtems_interrupt_flash()* (page 167). Using an otherwise obtained value is undefined behaviour.

This directive is unsuitable to enable particular interrupt sources, for example in an interrupt controller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- While at least one maskable interrupt is pending, when the directive enables maskable interrupts, the pending interrupts are immediately serviced. The interrupt service routines may unblock higher priority tasks which may preempt the calling task.
- Where the system was built with SMP support enabled, the directive is not available. Its use will result in compiler warnings and linker errors. The *rtems_interrupt_local_disable()* (page 168) and *rtems_interrupt_local_enable()* (page 170) directives are available in all build configurations.

### 8.4.4 rtems_interrupt_flash()

Flashes interrupts on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_flash( rtems_interrupt_level isr_cookie );
```

PARAMETERS:

**isr_cookie**
This parameter is the previous interrupt level.

DESCRIPTION:

This directive is functionally equivalent to a calling *rtems_interrupt_enable()* (page 166) immediately followed by a *rtems_interrupt_disable()* (page 164). On some architectures it is possible to provide an optimized implementation for this sequence.

NOTES:

The `isr_cookie` parameter value must be obtained by a previous call to *rtems_interrupt_disable()* (page 164) or *rtems_interrupt_flash()* (page 167). Using an otherwise obtained value is undefined behaviour.

Historically, the interrupt flash directive was heavily used in the operating system implementation. However, this is no longer the case. The interrupt flash directive is provided for backward compatibility reasons.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- Where the system was built with SMP support enabled, the directive is not available. Its use will result in compiler warnings and linker errors. The *rtems_interrupt_local_disable()* (page 168) and *rtems_interrupt_local_enable()* (page 170) directives are available in all build configurations.

### 8.4.5 rtems_interrupt_local_disable()

Disables the maskable interrupts on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_local_disable( rtems_interrupt_level isr_cookie );
```

PARAMETERS:

**isr_cookie**
This parameter is a variable of type *rtems_interrupt_level* (page 44) which will be used to save the previous interrupt level.

DESCRIPTION:

This directive disables all maskable interrupts on the current processor and returns the previous interrupt level in `isr_cookie`.

NOTES:

A later invocation of the *rtems_interrupt_local_enable()* (page 170) directive should be used to restore the previous interrupt level.

This directive is implemented as a macro which sets the `isr_cookie` parameter.

Where the system was built with SMP support enabled, this will not ensure system wide mutual exclusion. Use interrupt locks instead, see *rtems_interrupt_lock_acquire()* (page 174). Interrupt disabled critical sections may be used to access processor-specific data structures or disable thread dispatching.

```
#include <rtems.h>

void local_critical_section( void )
{
  rtems_interrupt_level level;

  // Please note that the rtems_interrupt_local_disable() is a macro.
  // The previous interrupt level (before the maskable interrupts are
  // disabled) is returned here in the level macro parameter.  This would
  // be wrong:
  //
  // rtems_interrupt_local_disable( &level );
  rtems_interrupt_local_disable( level );

  // Here is the critical section: maskable interrupts are disabled

  {
    rtems_interrupt_level nested_level;

    rtems_interrupt_local_disable( nested_level );

    // Here is a nested critical section
```

(continues on next page)

(continued from previous page)

```

    rtems_interrupt_local_enable( nested_level );
  }

  // Maskable interrupts are still disabled

  rtems_interrupt_local_enable( level );
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.6 rtems_interrupt_local_enable()

Restores the previous interrupt level on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_local_enable( rtems_interrupt_level isr_cookie );
```

PARAMETERS:

**isr_cookie**
This parameter is the previous interrupt level to restore. The value must be obtained by a previous call to *rtems_interrupt_local_disable()* (page 168).

DESCRIPTION:

This directive restores the interrupt level specified by `isr_cookie` on the current processor.

NOTES:

The `isr_cookie` parameter value must be obtained by a previous call to *rtems_interrupt_local_disable()* (page 168). Using an otherwise obtained value is undefined behaviour.

This directive is unsuitable to enable particular interrupt sources, for example in an interrupt controller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- While at least one maskable interrupt is pending, when the directive enables maskable interrupts, the pending interrupts are immediately serviced. The interrupt service routines may unblock higher priority tasks which may preempt the calling task.

### 8.4.7 rtems_interrupt_is_in_progress()

Checks if an ISR is in progress on the current processor.

CALLING SEQUENCE:

```
bool rtems_interrupt_is_in_progress( void );
```

DESCRIPTION:

This directive returns `true`, if the current processor is currently servicing an interrupt, and `false` otherwise. A return value of `true` indicates that the caller is an interrupt service routine, **not** a task. The directives available to an interrupt service routine are restricted.

RETURN VALUES:

Returns true, if the current processor is currently servicing an interrupt, otherwise false.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.8 rtems_interrupt_lock_initialize()

Initializes the ISR lock.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_initialize(
  rtems_interrupt_lock *lock,
  const char           *name
);
```

PARAMETERS:

**lock**
This parameter is the ISR lock to initialize.

**name**
This parameter is the ISR lock name. It shall be a string. The name is only used where the system was built with profiling support enabled.

NOTES:

ISR locks may also be statically defined by *RTEMS_INTERRUPT_LOCK_DEFINE()* (page 181) or statically initialized by *RTEMS_INTERRUPT_LOCK_INITIALIZER()* (page 182).

### 8.4.9 rtems_interrupt_lock_destroy()

Destroys the ISR lock.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_destroy( rtems_interrupt_lock *lock );
```

PARAMETERS:

**lock**
This parameter is the ISR lock to destroy.

NOTES:

The lock must have been dynamically initialized by *rtems_interrupt_lock_initialize()* (page 172), statically defined by *RTEMS_INTERRUPT_LOCK_DEFINE()* (page 181), or statically initialized by *RTEMS_INTERRUPT_LOCK_INITIALIZER()* (page 182).

Concurrent lock use during the destruction or concurrent destruction leads to unpredictable results.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.10 rtems_interrupt_lock_acquire()

Acquires the ISR lock.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_acquire(
  rtems_interrupt_lock         *lock,
  rtems_interrupt_lock_context *lock_context
);
```

PARAMETERS:

**lock**
This parameter is the ISR lock to acquire.

**lock_context**
This parameter is the ISR lock context. This lock context shall be used to release the lock by calling *rtems_interrupt_lock_release()* (page 176).

DESCRIPTION:

This directive acquires the ISR lock specified by `lock` using the lock context provided by `lock_context`. Maskable interrupts will be disabled on the current processor.

NOTES:

A caller-specific lock context shall be provided for each acquire/release pair, for example an automatic variable.

Where the system was built with SMP support enabled, this directive acquires an SMP lock. An attempt to recursively acquire the lock may result in an infinite loop with maskable interrupts disabled.

This directive establishes a non-preemptive critical section with system wide mutual exclusion on the local node in all RTEMS build configurations.

```
#include <rtems.h>

void critical_section( rtems_interrupt_lock *lock )
{
  rtems_interrupt_lock_context lock_context;

  rtems_interrupt_lock_acquire( lock, &lock_context );

  // Here is the critical section.  Maskable interrupts are disabled.
  // Where the system was built with SMP support enabled, this section
  // is protected by an SMP lock.

  rtems_interrupt_lock_release( lock, &lock_context );
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.11 rtems_interrupt_lock_release()

Releases the ISR lock.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_release( rtems_interrupt_lock_context *lock );
```

PARAMETERS:

**lock**
This parameter is the ISR lock to release.

**lock_context**
This parameter is the ISR lock context. This lock context shall have been used to acquire the lock by calling *rtems_interrupt_lock_acquire()* (page 174).

DESCRIPTION:

This directive releases the ISR lock specified by `lock` using the lock context provided by `lock_context`. The previous interrupt level will be restored on the current processor.

NOTES:

The lock context shall be the one used to acquire the lock, otherwise the result is unpredictable.

Where the system was built with SMP support enabled, this directive releases an SMP lock.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- While at least one maskable interrupt is pending, when the directive enables maskable interrupts, the pending interrupts are immediately serviced. The interrupt service routines may unblock higher priority tasks which may preempt the calling task.

### 8.4.12 rtems_interrupt_lock_acquire_isr()

Acquires the ISR lock from within an ISR.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_acquire_isr(
  rtems_interrupt_lock         *lock,
  rtems_interrupt_lock_context *lock_context
);
```

PARAMETERS:

**lock**
This parameter is the ISR lock to acquire within an ISR.

**lock_context**
This parameter is the ISR lock context. This lock context shall be used to release the lock by calling *rtems_interrupt_lock_release_isr()* (page 178).

DESCRIPTION:

This directive acquires the ISR lock specified by `lock` using the lock context provided by `lock_context`. The interrupt level will remain unchanged.

NOTES:

A caller-specific lock context shall be provided for each acquire/release pair, for example an automatic variable.

Where the system was built with SMP support enabled, this directive acquires an SMP lock. An attempt to recursively acquire the lock may result in an infinite loop.

This directive is intended for device drivers and should be called from the corresponding interrupt service routine.

In case the corresponding interrupt service routine can be interrupted by higher priority interrupts and these interrupts enter the critical section protected by this lock, then the result is unpredictable. This directive may be used under specific circumstances as an optimization. In doubt, use *rtems_interrupt_lock_acquire()* (page 174) and *rtems_interrupt_lock_release()* (page 176).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.13 rtems_interrupt_lock_release_isr()

Releases the ISR lock from within an ISR.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_release_isr(
  rtems_interrupt_lock         *lock,
  rtems_interrupt_lock_context *lock_context
);
```

PARAMETERS:

**lock**
This parameter is the ISR lock to release within an ISR.

**lock_context**
This parameter is the ISR lock context. This lock context shall have been used to acquire the lock by calling *rtems_interrupt_lock_acquire_isr()* (page 177).

DESCRIPTION:

This directive releases the ISR lock specified by `lock` using the lock context provided by `lock_context`. The interrupt level will remain unchanged.

NOTES:

The lock context shall be the one used to acquire the lock, otherwise the result is unpredictable.

Where the system was built with SMP support enabled, this directive releases an SMP lock.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.14 rtems_interrupt_lock_interrupt_disable()

Disables maskable interrupts on the current processor.

CALLING SEQUENCE:

```
void rtems_interrupt_lock_interrupt_disable(
  rtems_interrupt_lock_context *lock_context
);
```

PARAMETERS:

**lock_context**
This parameter is the ISR lock context for an acquire and release pair.

DESCRIPTION:

This directive disables maskable interrupts on the current processor and stores the previous interrupt level in `lock_context`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.15 RTEMS_INTERRUPT_LOCK_DECLARE()

Declares an ISR lock object.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_LOCK_DECLARE( specifier, designator );
```

PARAMETERS:

**specifier**
This parameter is the storage-class specifier for the ISR lock to declare, for example `extern` or `static`.

**designator**
This parameter is the ISR lock object designator.

NOTES:

Do not add a “;” after this macro.

### 8.4.16 RTEMS_INTERRUPT_LOCK_DEFINE()

Defines an ISR lock object.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_LOCK_DEFINE( specifier, designator, const char *name );
```

PARAMETERS:

**specifier**
This parameter is the storage-class specifier for the ISR lock to declare, for example `extern` or `static`.

**designator**
This parameter is the ISR lock object designator.

**name**
This parameter is the ISR lock name. It shall be a string. The name is only used where the system was built with profiling support enabled.

NOTES:

Do not add a ";" after this macro.

ISR locks may also be dynamically initialized by *rtems_interrupt_lock_initialize()* (page 172) or statically by *RTEMS_INTERRUPT_LOCK_INITIALIZER()* (page 182).

### 8.4.17 RTEMS_INTERRUPT_LOCK_INITIALIZER()

Statically initializes an ISR lock object.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_LOCK_INITIALIZER( const char *name );
```

PARAMETERS:

**name**
This parameter is the ISR lock name. It shall be a string. The name is only used where the system was built with profiling support enabled.

NOTES:

ISR locks may also be dynamically initialized by *rtems_interrupt_lock_initialize()* (page 172) or statically defined by *RTEMS_INTERRUPT_LOCK_DEFINE()* (page 181).

### 8.4.18 RTEMS_INTERRUPT_LOCK_MEMBER()

Defines an ISR lock member.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_LOCK_MEMBER( designator );
```

PARAMETERS:

**designator**
This parameter is the ISR lock member designator.

NOTES:

Do not add a ";" after this macro.

### 8.4.19 RTEMS_INTERRUPT_LOCK_REFERENCE()

Defines an ISR lock object reference.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_LOCK_REFERENCE( designator, rtems_interrupt_lock *target );
```

PARAMETERS:

**designator**
This parameter is the ISR lock reference designator.

**target**
This parameter is the target object to reference.

NOTES:

Do not add a “;” after this macro.

### 8.4.20 RTEMS_INTERRUPT_ENTRY_INITIALIZER()

Statically initializes an interrupt entry object.

CALLING SEQUENCE:

```
RTEMS_INTERRUPT_ENTRY_INITIALIZER(
  rtems_interrupt_handler routine,
  void                    *arg,
  const char              *info
);
```

PARAMETERS:

**routine**
This parameter is the interrupt handler routine for the entry.

**arg**
This parameter is the interrupt handler argument for the entry.

**info**
This parameter is the descriptive information for the entry.

NOTES:

Alternatively, *rtems_interrupt_entry_initialize()* (page 186) may be used to dynamically initialize an interrupt entry.

### 8.4.21 rtems_interrupt_entry_initialize()

Initializes the interrupt entry.

CALLING SEQUENCE:

```
void rtems_interrupt_entry_initialize(
  rtems_interrupt_entry  *entry,
  rtems_interrupt_handler routine,
  void                   *arg,
  const char             *info
);
```

PARAMETERS:

**entry**
This parameter is the interrupt entry to initialize.

**routine**
This parameter is the interrupt handler routine for the entry.

**arg**
This parameter is the interrupt handler argument for the entry.

**info**
This parameter is the descriptive information for the entry.

NOTES:

Alternatively, *RTEMS_INTERRUPT_ENTRY_INITIALIZER()* (page 185) may be used to statically initialize an interrupt entry.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 8.4.22 rtems_interrupt_entry_install()

Installs the interrupt entry at the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_entry_install(
  rtems_vector_number    vector,
  rtems_option           options,
  rtems_interrupt_entry *entry
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**options**
This parameter is the interrupt entry install option set.

**entry**
This parameter is the interrupt entry to install.

DESCRIPTION:

One of the following mutually exclusive options

- `RTEMS_INTERRUPT_UNIQUE`, and
- `RTEMS_INTERRUPT_SHARED`

shall be set in the `options` parameter.

The handler routine of the entry specified by `entry` will be called with the handler argument of the entry when dispatched. The order in which shared interrupt handlers are dispatched for one vector is defined by the installation order. The first installed handler is dispatched first.

If the option `RTEMS_INTERRUPT_UNIQUE` is set, then it will be ensured that the handler will be the only one for the interrupt vector.

If the option `RTEMS_INTERRUPT_SHARED` is set, then multiple handlers may be installed for the interrupt vector.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `entry` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The service was not initialized.

**RTEMS_INVALID_ADDRESS**
The handler routine of the entry was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_INVALID_NUMBER**
An option specified by `options` was not applicable.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_UNIQUE` option was set in `entry` and the interrupt vector was already occupied by a handler.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_SHARED` option was set in `entry` and the interrupt vector was already occupied by a unique handler.

**RTEMS_TOO_MANY**
The handler routine of the entry specified by `entry` was already installed for the interrupt vector specified by `vector` with an argument equal to the handler argument of the entry.

NOTES:

When the directive call was successful, the ownership of the interrupt entry has been transferred from the caller to the interrupt service. An installed interrupt entry may be removed from the interrupt service by calling *rtems_interrupt_entry_remove()* (page 189).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The interrupt entry shall have been initialized by *rtems_interrupt_entry_initialize()* (page 186) or *RTEMS_INTERRUPT_ENTRY_INITIALIZER()* (page 185).

### 8.4.23 rtems_interrupt_entry_remove()

Removes the interrupt entry from the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_entry_remove(
  rtems_vector_number    vector,
  rtems_interrupt_entry *entry
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**entry**
This parameter is the interrupt entry to remove.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The service was not initialized.

**RTEMS_INVALID_ADDRESS**
The `entry` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_UNSATISFIED**
The entry specified by `entry` was not installed at the interrupt vector specified by `vector`.

NOTES:

When the directive call was successful, the ownership of the interrupt entry has been transferred from the interrupt service to the caller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The interrupt entry shall have been installed by *rtems_interrupt_entry_install()* (page 187).

### 8.4.24 rtems_interrupt_handler_install()

Installs the interrupt handler routine and argument at the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_handler_install(
  rtems_vector_number     vector,
  const char             *info,
  rtems_option            options,
  rtems_interrupt_handler routine,
  void                   *arg
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**info**
This parameter is the descriptive information of the interrupt handler to install.

**options**
This parameter is the interrupt handler install option set.

**routine**
This parameter is the interrupt handler routine to install.

**arg**
This parameter is the interrupt handler argument to install.

DESCRIPTION:

One of the following mutually exclusive options

- `RTEMS_INTERRUPT_UNIQUE`,
- `RTEMS_INTERRUPT_SHARED`, and
- `RTEMS_INTERRUPT_REPLACE`

shall be set in the `options` parameter.

The handler routine will be called with the argument specified by `arg` when dispatched. The order in which shared interrupt handlers are dispatched for one vector is defined by the installation order. The first installed handler is dispatched first.

If the option `RTEMS_INTERRUPT_UNIQUE` is set, then it will be ensured that the handler will be the only one for the interrupt vector.

If the option `RTEMS_INTERRUPT_SHARED` is set, then multiple handler may be installed for the interrupt vector.

If the option `RTEMS_INTERRUPT_REPLACE` is set, then the handler specified by `routine` will replace the first handler with the same argument for the interrupt vector if it exists, otherwise an error status will be returned. A second handler with the same argument for the interrupt vector will remain unchanged. The new handler will inherit the unique or shared options from the replaced handler.

An informative description may be provided in `info`. It may be used for system debugging and diagnostic tools. The referenced string has to be persistent as long as the handler is installed.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The service was not initialized.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_NO_MEMORY**
There was not enough memory available to allocate data structures to install the handler.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_UNIQUE` option was set in `options` and the interrupt vector was already occupied by a handler.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_SHARED` option was set in `options` and the interrupt vector was already occupied by a unique handler.

**RTEMS_TOO_MANY**
The handler specified by `routine` was already installed for the interrupt vector specified by `vector` with an argument equal to the argument specified by `arg`.

**RTEMS_UNSATISFIED**
The `RTEMS_INTERRUPT_REPLACE` option was set in `options` and no handler to replace was installed.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.25 rtems_interrupt_handler_remove()

Removes the interrupt handler routine and argument from the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_handler_remove(
  rtems_vector_number     vector,
  rtems_interrupt_handler routine,
  void                   *arg
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**routine**
This parameter is the interrupt handler routine to remove.

**arg**
This parameter is the interrupt handler argument to remove.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The service was not initialized.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_UNSATISFIED**
There was no handler routine and argument pair installed specified by `routine` and `arg`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.26 rtems_interrupt_vector_is_enabled()

Checks if the interrupt vector is enabled.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_vector_is_enabled(
  rtems_vector_number vector,
  bool               *enabled
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**enabled**
This parameter is the pointer to a `bool` object. When the directive call is successful, the enabled status of the interrupt associated with the interrupt vector specified by `vector` will be stored in this object. When the interrupt was enabled for the processor executing the directive call at some time point during the call, the object value will be set to `true`, otherwise to `false`.

DESCRIPTION:

The directive checks if the interrupt associated with the interrupt vector specified by `vector` was enabled for the processor executing the directive call at some time point during the call.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `enabled` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

NOTES:

Interrupt vectors may be enabled by *rtems_interrupt_vector_enable()* (page 194) and disabled by *rtems_interrupt_vector_disable()* (page 195).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.27 rtems_interrupt_vector_enable()

Enables the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_vector_enable( rtems_vector_number vector );
```

PARAMETERS:

**vector**
This parameter is the number of the interrupt vector to enable.

DESCRIPTION:

The directive enables the interrupt vector specified by `vector`. This allows that interrupt service requests are issued to the target processors of the interrupt vector. Interrupt service requests for an interrupt vector may be raised by *rtems_interrupt_raise()* (page 198), *rtems_interrupt_raise_on()* (page 199), external signals, or messages.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
The request to enable the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if an interrupt vector can be enabled. Interrupt vectors may be disabled by *rtems_interrupt_vector_disable()* (page 195).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.28 rtems_interrupt_vector_disable()

Disables the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_vector_disable( rtems_vector_number vector );
```

PARAMETERS:

**vector**
This parameter is the number of the interrupt vector to disable.

DESCRIPTION:

The directive disables the interrupt vector specified by `vector`. This prevents that an interrupt service request is issued to the target processors of the interrupt vector.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
The request to disable the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if an interrupt vector can be disabled. Interrupt vectors may be enabled by *rtems_interrupt_vector_enable()* (page 194). There may be targets on which some interrupt vectors cannot be disabled, for example a hardware watchdog interrupt or software generated interrupts.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.29 rtems_interrupt_is_pending()

Checks if the interrupt is pending.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_is_pending(
  rtems_vector_number vector,
  bool                *pending
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**pending**
This parameter is the pointer to a `bool` object. When the directive call is successful, the pending status of the interrupt associated with the interrupt vector specified by `vector` will be stored in this object. When the interrupt was pending for the processor executing the directive call at some time point during the call, the object value will be set to `true`, otherwise to `false`.

DESCRIPTION:

The directive checks if the interrupt associated with the interrupt vector specified by `vector` was pending for the processor executing the directive call at some time point during the call.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `pending` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
The request to get the pending status has not been satisfied.

NOTES:

Interrupts may be made pending by calling the *rtems_interrupt_raise()* (page 198) or *rtems_interrupt_raise_on()* (page 199) directives or due to external signals or messages. The pending state may be cleared by *rtems_interrupt_clear()* (page 200).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

- The directive will not cause the calling task to be preempted.

### 8.4.30 rtems_interrupt_raise()

Raises the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_raise( rtems_vector_number vector );
```

PARAMETERS:

**vector**
This parameter is the number of the interrupt vector to raise.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
The request to raise the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if an interrupt vector can be raised.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.31 rtems_interrupt_raise_on()

Raises the interrupt vector on the processor.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_raise_on(
  rtems_vector_number vector,
  uint32_t            cpu_index
);
```

PARAMETERS:

**vector**
This parameter is the number of the interrupt vector to raise.

**cpu_index**
This parameter is the index of the target processor of the interrupt vector to raise.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_NOT_CONFIGURED**
The processor specified by `cpu_index` was not configured to be used by the application.

**RTEMS_INCORRECT_STATE**
The processor specified by `cpu_index` was configured to be used by the application, however, it was not online.

**RTEMS_UNSATISFIED**
The request to raise the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if an interrupt vector can be raised on a processor.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.32 rtems_interrupt_clear()

Clears the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_clear( rtems_vector_number vector );
```

PARAMETERS:

**vector**
This parameter is the number of the interrupt vector to clear.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
The request to raise the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if an interrupt vector can be cleared.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.33 rtems_interrupt_get_priority()

Gets the priority of the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_get_priority(
  rtems_vector_number vector,
  uint32_t           *priority
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**priority**
This parameter is the pointer to an uint32_t object. When the directive call is successful, the priority of the interrupt vector will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
There is no priority associated with the interrupt vector.

NOTES:

The *rtems_interrupt_set_priority()* (page 202) directive may be used to set the priority associated with an interrupt vector.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.34 rtems_interrupt_set_priority()

Sets the priority of the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_set_priority(
  rtems_vector_number vector,
  uint32_t            priority
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**priority**
This parameter is the new priority for the interrupt vector.

DESCRIPTION:

This directive sets the priority of the interrupt specified by `vector` to the priority specified by `priority`.

For processor-specific interrupts, the priority of the interrupt specific to a processor executing the directive call will be set.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_INVALID_PRIORITY**
The priority specified by `priority` was not a valid new priority for the interrupt vector.

**RTEMS_UNSATISFIED**
The request to set the priority of the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_priority()* (page 201) directive may be used to get the priority associated with an interrupt vector.

The interrupt prioritization support depends on the interrupt controller of the *target*. It is strongly recommended to read the relevant hardware documentation. What happens when the priority of a pending or active interrupt is changed, depends on the interrupt controller. In general, you should set the interrupt priority of an interrupt vector before a handler is installed. On some interrupt controllers, setting the priority to the maximum value (lowest importance) effectively disables the interrupt.

On some architectures, a range of interrupt priority values may be not disabled by the interrupt disable directives such as *rtems_interrupt_disable()* (page 164) and

*rtems_interrupt_local_disable()* (page 168). These interrupts are called non-maskable interrupts. Handlers of non-maskable interrupts shall not use operating system services. In addition, non-maskable interrupts may be not installable through *rtems_interrupt_entry_install()* (page 187) or *rtems_interrupt_handler_install()* (page 190), and may require architecture-specific prologue and epilogue code.

The interrupt priority settings affect the maximum nesting depth while servicing interrupts. The interrupt stack size calculation needs to take this into account, see also *CONFIGURE_INTERRUPT_STACK_SIZE* (page 574).

For the ARM Generic Interrupt Controller (GIC), an 8-bit priority value is supported. The granularity of the priority levels depends on the interrupt controller configuration. Some low-order bits of a priority value may be read-as-zero (RAZ) and writes are ignored (WI). Where group 0 (FIQ) and group 1 (IRQ) interrupts are used, it is recommended to use the lower half of the supported priority value range for the group 0 interrupts and the upper half for group 1 interrupts. This ensures that group 1 interrupts cannot preempt group 0 interrupts.

For the Armv7-M Nested Vector Interrupt Controller (NVIC), an 8-bit priority value is supported. The granularity of the priority levels depends on the interrupt controller configuration. Some lower bits of a priority value may be read-as-zero (RAZ) and writes are ignored (WI). Interrupts with a priority value less than 128 are not disabled by the RTEMS interrupt disable directives. Handlers of such interrupts shall not use operating system services.

For the RISC-V Platform-Level Interrupt Controller (PLIC), all priority values from 0 up to and including the 0xffffffff are supported since the priority for the PLIC is defined by a write-any-read-legal (WARL) register. Please note that for this directive in contrast to the PLIC, a higher priority value is associated with a lower importance. The maximum priority value (mapped to the value 0 for the PLIC) is reserved to mean “never interrupt” and effectively disables the interrupt.

For the QorIQ Multicore Programmable Interrupt Controller (MPIC), a 4-bit priority value is supported. Please note that for this directive in contrast to the MPIC, a higher priority value is associated with a lower importance. The maximum priority value of 15 (mapped to the value 0 for the MPIC) inhibits signalling of this interrupt.

Consult the *RTEMS CPU Architecture Supplement* and the *BSP* documentation in the *RTEMS User Manual* for further information.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.35 rtems_interrupt_get_affinity()

Gets the processor affinity set of the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_get_affinity(
  rtems_vector_number vector,
  size_t              affinity_size,
  cpu_set_t          *affinity
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**affinity_size**
This parameter is the size of the processor set referenced by `affinity` in bytes.

**affinity**
This parameter is the pointer to a `cpu_set_t` object. When the directive call is successful, the processor affinity set of the interrupt vector will be stored in this object. A set bit in the processor set means that the corresponding processor is in the processor affinity set of the interrupt vector, otherwise the bit is cleared.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `affinity` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_INVALID_SIZE**
The size specified by `affinity_size` of the processor set was too small for the processor affinity set of the interrupt vector.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.36 rtems_interrupt_set_affinity()

Sets the processor affinity set of the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_set_affinity(
  rtems_vector_number vector,
  size_t              affinity_size,
  const cpu_set_t    *affinity
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**affinity_size**
This parameter is the size of the processor set referenced by `affinity` in bytes.

**affinity**
This parameter is the pointer to a `cpu_set_t` object. The processor set defines the new processor affinity set of the interrupt vector. A set bit in the processor set means that the corresponding processor shall be in the processor affinity set of the interrupt vector, otherwise the bit shall be cleared.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `affinity` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_INVALID_NUMBER**
The referenced processor set was not a valid new processor affinity set for the interrupt vector.

**RTEMS_UNSATISFIED**
The request to set the processor affinity of the interrupt vector has not been satisfied.

NOTES:

The *rtems_interrupt_get_attributes()* (page 207) directive may be used to check if the processor affinity of an interrupt vector can be set.

Only online processors of the affinity set specified by `affinity_size` and `affinity` are considered by the directive. Other processors of the set are ignored. If the set contains no online processor, then the set is invalid and an error status is returned.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.37 rtems_interrupt_get_attributes()

Gets the attributes of the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_get_attributes(
  rtems_vector_number         vector,
  rtems_interrupt_attributes *attributes
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**attributes**
This parameter is the pointer to an *rtems_interrupt_attributes* (page 42) object. When the directive call is successful, the attributes of the interrupt vector will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `attributes` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 8.4.38 rtems_interrupt_handler_iterate()

Iterates over all interrupt handler installed at the interrupt vector.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_handler_iterate(
  rtems_vector_number                 vector,
  rtems_interrupt_per_handler_routine routine,
  void                                *arg
);
```

PARAMETERS:

**vector**
This parameter is the interrupt vector number.

**routine**
This parameter is the visitor routine.

**arg**
This parameter is the visitor argument.

DESCRIPTION:

For each installed handler at the interrupt vector the visitor function specified by `routine` will be called with the argument specified by `arg` and the handler information, options, routine and argument.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The service was not initialized.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

NOTES:

The directive is intended for system information and diagnostics.

Never install or remove an interrupt handler within the visitor function. This may result in a deadlock.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.39 rtems_interrupt_server_initialize()

Initializes the interrupt server tasks.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_initialize(
  rtems_task_priority priority,
  size_t              stack_size,
  rtems_mode          modes,
  rtems_attribute     attributes,
  uint32_t           *server_count
);
```

PARAMETERS:

**priority**
This parameter is the initial *task priority* of the created interrupt servers.

**stack_size**
This parameter is the task stack size of the created interrupt servers.

**modes**
This parameter is the initial mode set of the created interrupt servers.

**attributes**
This parameter is the attribute set of the created interrupt servers.

**server_count**
This parameter is the pointer to an uint32_t object or NULL. When the pointer is not equal to NULL, the count of successfully created interrupt servers is stored in this object regardless of the return status.

DESCRIPTION:

The directive tries to create an interrupt server task for each online processor in the system. The tasks will have the initial priority specified by `priority`, the stack size specified by `stack_size`, the initial mode set specified by `modes`, and the attribute set specified by `attributes`. The count of successfully created server tasks will be returned in `server_count` if the pointer is not equal to NULL.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The interrupt servers were already initialized.

The directive uses *rtems_task_create()* (page 109). If this directive fails, then its error status will be returned.

NOTES:

Interrupt handlers may be installed on an interrupt server with *rtems_interrupt_server_handler_install()* (page 213) and removed with *rtems_interrupt_server_handler_remove()* (page 215) using a server index. In case of an interrupt, the request will be forwarded to the interrupt server. The handlers are executed within the interrupt server context. If one handler blocks on something this may delay the processing of other handlers.

Interrupt servers may be deleted by *rtems_interrupt_server_delete()* (page 218).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.40 rtems_interrupt_server_create()

Creates an interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_create(
  rtems_interrupt_server_control      *control,
  const rtems_interrupt_server_config *config,
  uint32_t                            *server_index
);
```

PARAMETERS:

**control**
This parameter is the pointer to an *rtems_interrupt_server_control* (page 45) object. When the directive call was successful, the ownership of the object was transferred from the caller of the directive to the interrupt server management.

**config**
This parameter is the interrupt server configuration.

**server_index**
This parameter is the pointer to an uint32_t object. When the directive call was successful, the index of the created interrupt server will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

The directive uses *rtems_task_create()* (page 109). If this directive fails, then its error status will be returned.

NOTES:

See also *rtems_interrupt_server_initialize()* (page 210) and *rtems_interrupt_server_delete()* (page 218).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.41 rtems_interrupt_server_handler_install()

Installs the interrupt handler routine and argument at the interrupt vector on the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_handler_install(
  uint32_t                server_index,
  rtems_vector_number     vector,
  const char             *info,
  rtems_option            options,
  rtems_interrupt_handler routine,
  void                   *arg
);
```

PARAMETERS:

**server_index**
This parameter is the interrupt server index. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**vector**
This parameter is the interrupt vector number.

**info**
This parameter is the descriptive information of the interrupt handler to install.

**options**
This parameter is the interrupt handler install option set.

**routine**
This parameter is the interrupt handler routine to install.

**arg**
This parameter is the interrupt handler argument to install.

DESCRIPTION:

The handler routine specified by `routine` will be executed within the context of the interrupt server task specified by `server_index`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

**RTEMS_CALLED_FROM_ISR**
The directive was called from within interrupt context.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_INVALID_NUMBER**
An option specified by `info` was not applicable.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_UNIQUE` option was set in `info` and the interrupt vector was already occupied by a handler.

**RTEMS_RESOURCE_IN_USE**
The `RTEMS_INTERRUPT_SHARED` option was set in `info` and the interrupt vector was already occupied by a unique handler.

**RTEMS_TOO_MANY**
The handler specified by `routine` was already installed for the interrupt vector specified by `vector` with an argument equal to the argument specified by `arg`.

**RTEMS_UNSATISFIED**
The `RTEMS_INTERRUPT_REPLACE` option was set in `info` and no handler to replace was installed.

NOTES:

See also *rtems_interrupt_handler_install()* (page 190).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.42 rtems_interrupt_server_handler_remove()

Removes the interrupt handler routine and argument from the interrupt vector and the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_handler_remove(
  uint32_t                server_index,
  rtems_vector_number     vector,
  rtems_interrupt_handler routine,
  void                   *arg
);
```

PARAMETERS:

**server_index**
This parameter is the interrupt server index. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**vector**
This parameter is the interrupt vector number.

**routine**
This parameter is the interrupt handler routine to remove.

**arg**
This parameter is the interrupt handler argument to remove.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

**RTEMS_UNSATISFIED**
There was no handler routine and argument pair installed specified by `routine` and `arg`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.

### 8.4.43 rtems_interrupt_server_set_affinity()

Sets the processor affinity of the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_set_affinity(
  uint32_t            server_index,
  size_t              affinity_size,
  const cpu_set_t    *affinity,
  rtems_task_priority priority
);
```

PARAMETERS:

**server_index**
This parameter is the interrupt server index. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**affinity_size**
This parameter is the size of the processor set referenced by `affinity` in bytes.

**affinity**
This parameter is the pointer to a `cpu_set_t` object. The processor set defines the new processor affinity set of the interrupt server. A set bit in the processor set means that the corresponding processor shall be in the processor affinity set of the task, otherwise the bit shall be cleared.

**priority**
This parameter is the new *real priority* for the interrupt server.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

The directive uses *rtems_scheduler_ident_by_processor_set()* (page 74), *rtems_task_set_scheduler()* (page 137), and *rtems_task_set_affinity()* (page 141). If one of these directive fails, then its error status will be returned.

NOTES:

The scheduler is set determined by the highest numbered processor in the affinity set specified by `affinity`.

This operation is only reliable in case the interrupt server was suspended via *rtems_interrupt_server_suspend()* (page 219).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may change the processor affinity of a task. This may cause the calling task to be preempted.
- The directive may change the priority of a task. This may cause the calling task to be preempted.

### 8.4.44 rtems_interrupt_server_delete()

Deletes the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_delete( uint32_t server_index );
```

PARAMETERS:

**server_index**
This parameter is the index of the interrupt server to delete.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the server index specified by `server_index`.

NOTES:

The interrupt server deletes itself, so after the return of the directive the interrupt server may be still in the termination process depending on the task priorities of the system.

See also *rtems_interrupt_server_create()* (page 212).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.

### 8.4.45 rtems_interrupt_server_suspend()

Suspends the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_suspend( uint32_t server_index );
```

PARAMETERS:

**server_index**
This parameter is the index of the interrupt server to suspend. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

NOTES:

Interrupt server may be resumed by *rtems_interrupt_server_resume()* (page 220).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.

### 8.4.46 rtems_interrupt_server_resume()

Resumes the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_resume( uint32_t server_index );
```

PARAMETERS:

**server_index**
This parameter is the index of the interrupt server to resume. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

NOTES:

Interrupt server may be suspended by *rtems_interrupt_server_suspend()* (page 219).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.

### 8.4.47 rtems_interrupt_server_move()

Moves the interrupt handlers installed at the interrupt vector and the source interrupt server to the destination interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_move(
  uint32_t            source_server_index,
  rtems_vector_number vector,
  uint32_t            destination_server_index
);
```

PARAMETERS:

**source_server_index**
This parameter is the index of the source interrupt server. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**vector**
This parameter is the interrupt vector number.

**destination_server_index**
This parameter is the index of the destination interrupt server. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `source_server_index`.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `destination_server_index`.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.

### 8.4.48 rtems_interrupt_server_handler_iterate()

Iterates over all interrupt handler installed at the interrupt vector and interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_handler_iterate(
  uint32_t                           server_index,
  rtems_vector_number                vector,
  rtems_interrupt_per_handler_routine routine,
  void                              *arg
);
```

PARAMETERS:

**server_index**
This parameter is the index of the interrupt server.

**vector**
This parameter is the interrupt vector number.

**routine**
This parameter is the visitor routine.

**arg**
This parameter is the visitor argument.

DESCRIPTION:

For each installed handler at the interrupt vector and interrupt server the visitor function specified by `vector` will be called with the argument specified by `routine` and the handler information, options, routine and argument.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

**RTEMS_INVALID_ID**
There was no interrupt vector associated with the number specified by `vector`.

NOTES:

The directive is intended for system information and diagnostics.

Never install or remove an interrupt handler within the visitor function. This may result in a deadlock.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.49 rtems_interrupt_server_entry_initialize()

Initializes the interrupt server entry.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_entry_initialize(
  uint32_t                      server_index,
  rtems_interrupt_server_entry *entry
);
```

PARAMETERS:

**server_index**
This parameter is the interrupt server index. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**entry**
This parameter is the interrupt server entry to initialize.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

NOTES:

After initialization, the list of actions of the interrupt server entry is empty. Actions may be prepended by *rtems_interrupt_server_action_prepend()* (page 225). Interrupt server entries may be moved to another interrupt vector with *rtems_interrupt_server_entry_move()* (page 229). Server entries may be submitted to get serviced by the interrupt server with *rtems_interrupt_server_entry_submit()* (page 228). Server entries may be destroyed by *rtems_interrupt_server_entry_destroy()* (page 227).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.50 rtems_interrupt_server_action_prepend()

Prepends the interrupt server action to the list of actions of the interrupt server entry.

CALLING SEQUENCE:

```
void rtems_interrupt_server_action_prepend(
  rtems_interrupt_server_entry  *entry,
  rtems_interrupt_server_action *action,
  rtems_interrupt_handler        routine,
  void                          *arg
);
```

PARAMETERS:

**entry**
This parameter is the interrupt server entry to prepend the interrupt server action. It shall have been initialized via *rtems_interrupt_server_entry_initialize()* (page 224).

**action**
This parameter is the interrupt server action to initialize and prepend to the list of actions of the entry.

**routine**
This parameter is the interrupt handler routine to set in the action.

**arg**
This parameter is the interrupt handler argument to set in the action.

NOTES:

No error checking is performed by the directive.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.
- The interrupt server entry shall have been initialized by *rtems_interrupt_server_entry_initialize()* (page 224) and further optional calls to *rtems_interrupt_server_action_prepend()* (page 225).
- The directive shall not be called concurrently with *rtems_interrupt_server_action_prepend()* (page 225) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called concurrently with *rtems_interrupt_server_entry_move()* (page 229) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.

- The directive shall not be called concurrently with *rtems_interrupt_server_entry_submit()* (page 228) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called while the interrupt server entry is pending on or serviced by its current interrupt server. Calling the directive under these conditions is undefined behaviour.

### 8.4.51 rtems_interrupt_server_entry_destroy()

Destroys the interrupt server entry.

CALLING SEQUENCE:

```
void rtems_interrupt_server_entry_destroy(
  rtems_interrupt_server_entry *entry
);
```

PARAMETERS:

**entry**
This parameter is the interrupt server entry to destroy.

NOTES:

No error checking is performed by the directive.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.
- The interrupt server entry shall have been initialized by *rtems_interrupt_server_entry_initialize()* (page 224) and further optional calls to *rtems_interrupt_server_action_prepend()* (page 225).

### 8.4.52 rtems_interrupt_server_entry_submit()

Submits the interrupt server entry to be serviced by the interrupt server.

CALLING SEQUENCE:

```
void rtems_interrupt_server_entry_submit(
  rtems_interrupt_server_entry *entry
);
```

PARAMETERS:

**entry**
This parameter is the interrupt server entry to submit.

DESCRIPTION:

The directive appends the entry to the pending entries of the interrupt server. The interrupt server is notified that a new entry is pending. Once the interrupt server is scheduled it services the actions of all pending entries.

NOTES:

This directive may be used to do a two-step interrupt processing. The first step is done from within interrupt context by a call to this directive. The second step is then done from within the context of the interrupt server.

No error checking is performed by the directive.

A submitted entry may be destroyed by *rtems_interrupt_server_entry_destroy()* (page 227).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- The interrupt server entry shall have been initialized by *rtems_interrupt_server_entry_initialize()* (page 224) and further optional calls to *rtems_interrupt_server_action_prepend()* (page 225).
- The directive shall not be called concurrently with *rtems_interrupt_server_action_prepend()* (page 225) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called concurrently with *rtems_interrupt_server_entry_move()* (page 229) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.

### 8.4.53 rtems_interrupt_server_entry_move()

Moves the interrupt server entry to the interrupt server.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_entry_move(
  rtems_interrupt_server_entry *entry,
  uint32_t                     server_index
);
```

PARAMETERS:

**entry**
This parameter is the interrupt server entry to move.

**server_index**
This parameter is the index of the destination interrupt server. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The interrupt server entry shall have been initialized by *rtems_interrupt_server_entry_initialize()* (page 224) and further optional calls to *rtems_interrupt_server_action_prepend()* (page 225).
- The directive shall not be called concurrently with *rtems_interrupt_server_action_prepend()* (page 225) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called concurrently with *rtems_interrupt_server_entry_move()* (page 229) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called concurrently with *rtems_interrupt_server_entry_submit()* (page 228) with the same interrupt server entry. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called while the interrupt server entry is pending on or serviced by its current interrupt server. Calling the directive under these conditions is undefined behaviour.

### 8.4.54 rtems_interrupt_server_request_initialize()

Initializes the interrupt server request.

CALLING SEQUENCE:

```
rtems_status_code rtems_interrupt_server_request_initialize(
  uint32_t                        server_index,
  rtems_interrupt_server_request *request,
  rtems_interrupt_handler         routine,
  void                           *arg
);
```

PARAMETERS:

**server_index**
This parameter is the interrupt server index. The constant `RTEMS_INTERRUPT_SERVER_DEFAULT` may be used to specify the default interrupt server.

**request**
This parameter is the interrupt server request to initialize.

**routine**
This parameter is the interrupt handler routine for the request action.

**arg**
This parameter is the interrupt handler argument for the request action.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no interrupt server associated with the index specified by `server_index`.

NOTES:

An interrupt server requests consists of an interrupt server entry and exactly one interrupt server action. The interrupt vector of the request may be changed with *rtems_interrupt_server_request_set_vector()* (page 231). Interrupt server requests may be submitted to get serviced by the interrupt server with *rtems_interrupt_server_request_submit()* (page 233). Requests may be destroyed by *rtems_interrupt_server_request_destroy()* (page 232).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 8.4.55 rtems_interrupt_server_request_set_vector()

Sets the interrupt vector in the interrupt server request.

CALLING SEQUENCE:

```
void rtems_interrupt_server_request_set_vector(
  rtems_interrupt_server_request *request,
  rtems_vector_number             vector
);
```

PARAMETERS:

**request**
This parameter is the interrupt server request to change.

**vector**
This parameter is the interrupt vector number to be used by the request.

NOTES:

By default, the interrupt vector of an interrupt server request is set to a special value which is outside the range of vectors supported by the interrupt controller hardware.

Calls to *rtems_interrupt_server_request_submit()* (page 233) will disable the interrupt vector of the request. After processing of the request by the interrupt server the interrupt vector will be enabled again.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.
- The interrupt server request shall have been initialized by *rtems_interrupt_server_request_initialize()* (page 230).
- The directive shall not be called concurrently with *rtems_interrupt_server_request_set_vector()* (page 231) with the same interrupt server request. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called concurrently with *rtems_interrupt_server_request_submit()* (page 233) with the same interrupt server request. Calling the directive under this condition is undefined behaviour.
- The directive shall not be called while the interrupt server entry is pending on or serviced by its current interrupt server. Calling the directive under these conditions is undefined behaviour.

### 8.4.56 rtems_interrupt_server_request_destroy()

Destroys the interrupt server request.

CALLING SEQUENCE:

```
void rtems_interrupt_server_request_destroy(
  rtems_interrupt_server_request *request
);
```

PARAMETERS:

**request**
This parameter is the interrupt server request to destroy.

NOTES:

No error checking is performed by the directive.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive shall not be called from within the context of an interrupt server. Calling the directive from within the context of an interrupt server is undefined behaviour.
- The directive sends a request to another task and waits for a response. This may cause the calling task to be blocked and unblocked.
- The interrupt server request shall have been initialized by *rtems_interrupt_server_request_initialize()* (page 230).

### 8.4.57 rtems_interrupt_server_request_submit()

Submits the interrupt server request to be serviced by the interrupt server.

CALLING SEQUENCE:

```
void rtems_interrupt_server_request_submit(
  rtems_interrupt_server_request *request
);
```

PARAMETERS:

**request**
This parameter is the interrupt server request to submit.

DESCRIPTION:

The directive appends the interrupt server entry of the request to the pending entries of the interrupt server. The interrupt server is notified that a new entry is pending. Once the interrupt server is scheduled it services the actions of all pending entries.

NOTES:

This directive may be used to do a two-step interrupt processing. The first step is done from within interrupt context by a call to this directive. The second step is then done from within the context of the interrupt server.

No error checking is performed by the directive.

A submitted request may be destroyed by *rtems_interrupt_server_request_destroy()* (page 232).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- The interrupt server request shall have been initialized by *rtems_interrupt_server_request_initialize()* (page 230).
- The directive shall not be called concurrently with *rtems_interrupt_server_request_set_vector()* (page 231) with the same interrupt server request. Calling the directive under this condition is undefined behaviour.

CHAPTER

**NINE**

# CLOCK MANAGER

## 9.1 Introduction

The Clock Manager provides support for time of day and other time related capabilities. The directives provided by the Clock Manager are:

- *rtems_clock_set()* (page 243) - Sets the *CLOCK_REALTIME* to the time of day.
- *rtems_clock_get_tod()* (page 245) - Gets the time of day associated with the current *CLOCK_REALTIME*.
- *rtems_clock_get_tod_timeval()* (page 246) - Gets the seconds and microseconds elapsed since the *Unix epoch* and the current *CLOCK_REALTIME*.
- *rtems_clock_get_realtime()* (page 247) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in seconds and nanoseconds format.
- *rtems_clock_get_realtime_bintime()* (page 248) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in binary time format.
- *rtems_clock_get_realtime_timeval()* (page 249) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in seconds and microseconds format.
- *rtems_clock_get_realtime_coarse()* (page 250) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in seconds and nanoseconds format.
- *rtems_clock_get_realtime_coarse_bintime()* (page 251) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in binary time format.
- *rtems_clock_get_realtime_coarse_timeval()* (page 252) - Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in seconds and microseconds format.
- *rtems_clock_get_monotonic()* (page 253) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in seconds and nanoseconds format.
- *rtems_clock_get_monotonic_bintime()* (page 254) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in binary time format.
- *rtems_clock_get_monotonic_sbintime()* (page 255) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in signed binary time format.
- *rtems_clock_get_monotonic_timeval()* (page 256) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in seconds and microseconds format.
- *rtems_clock_get_monotonic_coarse()* (page 257) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in seconds and nanoseconds format.
- *rtems_clock_get_monotonic_coarse_bintime()* (page 258) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in binary time format.
- *rtems_clock_get_monotonic_coarse_timeval()* (page 259) - Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in seconds and microseconds format.

- *rtems_clock_get_boot_time()* (page 260) - Gets the time elapsed since the *Unix epoch* at some time point during system initialization in seconds and nanoseconds format.
- *rtems_clock_get_boot_time_bintime()* (page 261) - Gets the time elapsed since the *Unix epoch* at some time point during system initialization in binary time format.
- *rtems_clock_get_boot_time_timeval()* (page 262) - Gets the time elapsed since the *Unix epoch* at some time point during system initialization in seconds and microseconds format.
- *rtems_clock_get_seconds_since_epoch()* (page 263) - Gets the seconds elapsed since the *RTEMS epoch* and the current *CLOCK_REALTIME*.
- *rtems_clock_get_ticks_per_second()* (page 264) - Gets the number of *clock ticks* per second configured for the application.
- *rtems_clock_get_ticks_since_boot()* (page 265) - Gets the number of *clock ticks* since some time point during the system initialization or the last overflow of the clock tick counter.
- *rtems_clock_get_uptime()* (page 266) - Gets the seconds and nanoseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.
- *rtems_clock_get_uptime_timeval()* (page 267) - Gets the seconds and microseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.
- *rtems_clock_get_uptime_seconds()* (page 268) - Gets the seconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.
- *rtems_clock_get_uptime_nanoseconds()* (page 269) - Gets the nanoseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.
- *rtems_clock_tick_later()* (page 270) - Gets a *clock tick* value which is at least delta clock ticks in the future.
- *rtems_clock_tick_later_usec()* (page 271) - Gets a *clock tick* value which is at least delta microseconds in the future.
- *rtems_clock_tick_before()* (page 272) - Indicates if the current *clock tick* counter is before the ticks.

## 9.2 Background

### 9.2.1 Required Support

For the features provided by the Clock Manager to be utilized, a *Clock Driver* is required. The Clock Driver usually provides a clock interrupt which is serviced on each configured processor at each *clock tick*. In addition, the Clock Driver provides three clock sources:

- clock tick
- *CLOCK_REALTIME*
- *CLOCK_MONOTONIC*

The time of these clock sources advances at each clock tick. This yields the time of the clock sources in a coarse resolution. To get the time of the `CLOCK_REALTIME` or `CLOCK_MONOTONIC` clock sources in a higher resolution, the Clock Driver may use a clock device to get the time between clock ticks.

### 9.2.2 Time and Date Data Structures

The clock facilities of the Clock Manager operate upon calendar time. These directives utilize the following date and time structure for the native time and date format:

```
typedef struct {
    uint32_t year;   /* greater than 1987 */
    uint32_t month;  /* 1 - 12 */
    uint32_t day;    /* 1 - 31 */
    uint32_t hour;   /* 0 - 23 */
    uint32_t minute; /* 0 - 59 */
    uint32_t second; /* 0 - 59 */
    uint32_t ticks;  /* elapsed between seconds */
} rtems_time_of_day;
```

The native date and time format is the only format supported when setting the system date and time using the *rtems_clock_set()* (page 243) directive. Some applications expect to operate on a *UNIX-style* date and time data structure. For example, the *rtems_clock_get_tod_timeval()* (page 246) returns the date and time in `struct timeval` format.

Some directives use data structures defined by *POSIX*. The `struct timeval` data structure has two members: `tv_sec` and `tv_usec` which are seconds and microseconds, respectively. The `struct timespec` data structure has two members: `tv_sec` and `tv_nsec` which are seconds and nanoseconds, respectively. For *CLOCK_REALTIME* time points, the `tv_sec` member in these data structures is the number of seconds since the *Unix epoch* but will never be prior to the *RTEMS epoch*.

The `struct bintime` and `sbintime_t` time formats used by some directives originate in FreeBSD. The `struct bintime` data structure which represents time in a binary time format has two members: `sec` and `frac` which are seconds and fractions of a second in units of $1/2^{64}$ seconds, respectively. The `sbintime_t` type is a signed 64-bit integer type used to represent time in units of $1/2^{32}$ seconds.

### 9.2.3 Clock Tick and Timeslicing

Timeslicing is a task scheduling discipline in which tasks of equal priority are executed for a specific period of time before control of the CPU is passed to another task. It is also sometimes referred to as the automatic round-robin scheduling algorithm. The length of time allocated to each task is known as the quantum or timeslice.

The system's timeslice is defined as an integral number of ticks, and is specified by the *CONFIGURE_TICKS_PER_TIMESLICE* (page 586) application configuration option. The timeslice is defined for the entire system of tasks, but timeslicing is enabled and disabled on a per task basis.

The clock tick directives implement timeslicing by decrementing the running task's time-remaining counter when both timeslicing and preemption are enabled. If the task's timeslice has expired, then that task will be preempted if there exists a ready task of equal priority.

### 9.2.4 Delays

A sleep timer allows a task to delay for a given interval or up until a given time, and then wake and continue execution. This type of timer is created automatically by the *rtems_task_wake_after()* (page 134) and *rtems_task_wake_when()* (page 135) directives and, as a result, does not have an object identifier. Once activated, a sleep timer cannot be explicitly deleted. Each task may activate one and only one sleep timer at a time.

### 9.2.5 Timeouts

Timeouts are a special type of timer automatically created when the timeout option is used on the *rtems_barrier_wait()* (page 363), *rtems_event_receive()* (page 401), *rtems_message_queue_receive()* (page 387), *rtems_region_get_segment()* (page 444), and *rtems_semaphore_obtain()* (page 343) directives. Each task may have one and only one timeout active at a time. When a timeout expires, it unblocks the task with a timeout status code.

## 9.3 Operations

### 9.3.1 Announcing a Tick

RTEMS provides the several clock tick directives which are called from the user's real-time clock ISR to inform RTEMS that a tick has elapsed. Depending on the timer hardware capabilities the clock driver must choose the most appropriate clock tick directive. The tick frequency value, defined in microseconds, is a configuration parameter found in the Configuration Table. RTEMS divides one million microseconds (one second) by the number of microseconds per tick to determine the number of calls to the clock tick directive per second. The frequency of clock tick calls determines the resolution (granularity) for all time dependent RTEMS actions. For example, calling the clock tick directive ten times per second yields a higher resolution than calling the clock tick two times per second. The clock tick directives are responsible for maintaining both calendar time and the dynamic set of timers.

### 9.3.2 Setting the Time

The `rtems_clock_set` directive allows a task or an ISR to set the date and time maintained by RTEMS. If setting the date and time causes any outstanding timers to pass their deadline, then the expired timers will be fired during the invocation of the `rtems_clock_set` directive.

### 9.3.3 Obtaining the Time

RTEMS provides multiple directives which can be used by an application to obtain the current date and time or date and time related information. These directives allow a task or an ISR to obtain the current date and time or date and time related information. The current date and time can be returned in either native or *UNIX-style* format. Additionally, the application can obtain date and time related information such as the number of seconds since the RTEMS epoch, the number of ticks since the executive was initialized, and the number of ticks per second. The following directives are available:

**`rtems_clock_get_tod`**
obtain native style date and time

**`rtems_clock_get_time_value`**
obtain *UNIX-style* date and time

**`rtems_clock_get_ticks_since_boot`**
obtain number of ticks since RTEMS was initialized

**`rtems_clock_get_seconds_since_epoch`**
obtain number of seconds since RTEMS epoch

**`rtems_clock_get_ticks_per_second`**
obtain number of clock ticks per second

Calendar time operations will return an error code if invoked before the date and time have been set.

### 9.3.4 Transition Advice for the Removed rtems_clock_get()

The directive *CLOCK_GET - Get date and time information* (page 274) took an untyped pointer with an options argument to indicate the time information desired. This has been replaced with a set of typed directives:

- rtems_clock_get_seconds_since_epoch

- rtems_clock_get_ticks_per_second
- rtems_clock_get_ticks_since_boot
- rtems_clock_get_tod
- rtems_clock_get_tod_timeval

These directives directly correspond to what were previously referred to as *clock options*. These strongly typed directives were available for multiple releases in parallel with `rtems_clock_get()` until that directive was removed.

## 9.4 Directives

This section details the directives of the Clock Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 9.4.1 rtems_clock_set()

Sets the *CLOCK_REALTIME* to the time of day.

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_set( const rtems_time_of_day *time_of_day );
```

PARAMETERS:

**time_of_day**
This parameter is the time of day to set the clock.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `time_of_day` parameter was NULL.

**RTEMS_INVALID_CLOCK**
The time of day specified by `time_of_day` was invalid.

NOTES:

The date, time, and ticks specified by `time_of_day` are all range-checked, and an error is returned if any one is out of its valid range.

RTEMS can represent time points of the *CLOCK_REALTIME* clock in nanoseconds ranging from 1988-01-01T00:00:00.000000000Z to 2514-05-31T01:53:03.999999999Z. The future uptime of the system shall be in this range, otherwise the system behaviour is undefined. Due to implementation constraints, the time of day set by the directive shall be before 2100-01-01:00:00.000000000Z. The latest valid time of day accepted by the POSIX clock_settime() is 2400-01-01T00:00:00.999999999Z.

The specified time is based on the configured *clock tick* rate, see the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration option.

Setting the time forward will fire all *CLOCK_REALTIME* timers which are scheduled at a time point before or at the time set by the directive. This may unblock tasks, which may preempt the calling task. User-provided timer routines will execute in the context of the caller.

It is allowed to call this directive from within interrupt context, however, this is not recommended since an arbitrary number of timers may fire.

The directive shall be called at least once to enable the service of *CLOCK_REALTIME* related directives. If the clock is not set at least once, they may return an error status.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive may change the priority of a task. This may cause the calling task to be preempted.

- The directive may unblock a task. This may cause the calling task to be preempted.
- The time of day set by the directive shall be 1988-01-01T00:00:00.000000000Z or later.
- The time of day set by the directive shall be before 2100-01-01T00:00:00.000000000Z.

### 9.4.2 rtems_clock_get_tod()

Gets the time of day associated with the current *CLOCK_REALTIME*.

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_get_tod( rtems_time_of_day *time_of_day );
```

PARAMETERS:

**time_of_day**
This parameter is the pointer to an *rtems_time_of_day* (page 59) object. When the directive call is successful, the time of day associated with the *CLOCK_REALTIME* at some point during the directive call will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `time_of_day` parameter was NULL.

**RTEMS_NOT_DEFINED**
The *CLOCK_REALTIME* was not set. It can be set with *rtems_clock_set()* (page 243).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.3 rtems_clock_get_tod_timeval()

Gets the seconds and microseconds elapsed since the *Unix epoch* and the current *CLOCK_REALTIME*.

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_get_tod_timeval( struct timeval *time_of_day );
```

PARAMETERS:

**time_of_day**
This parameter is the pointer to a struct timeval object. When the directive call is successful, the seconds and microseconds elapsed since the *Unix epoch* and the *CLOCK_REALTIME* at some point during the directive call will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `time_of_day` parameter was NULL.

**RTEMS_NOT_DEFINED**
The *CLOCK_REALTIME* was not set. It can be set with *rtems_clock_set()* (page 243).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.4 rtems_clock_get_realtime()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in seconds and nanoseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime( struct timespec *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timespec object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_realtime_coarse()* (page 250) directive may be used to get the time in a lower resolution and with less runtime overhead.

See *rtems_clock_get_realtime_bintime()* (page 248) and *rtems_clock_get_realtime_timeval()* (page 249) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.5 rtems_clock_get_realtime_bintime()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in binary time format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime_bintime( struct bintime *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a `struct bintime` object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_realtime_coarse_bintime()* (page 251) directive may be used to get the time in a lower resolution and with less runtime overhead.

See *rtems_clock_get_realtime()* (page 247) and *rtems_clock_get_realtime_timeval()* (page 249) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.6 rtems_clock_get_realtime_timeval()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in seconds and microseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime_timeval( struct timeval *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timeval object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_realtime_coarse_timeval()* (page 252) directive may be used to get the time in a lower resolution and with less runtime overhead.

See *rtems_clock_get_realtime()* (page 247) and *rtems_clock_get_realtime_bintime()* (page 248) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.7 rtems_clock_get_realtime_coarse()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in seconds and nanoseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime_coarse( struct timespec *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timespec object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_realtime()* (page 247) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_realtime_coarse_bintime()* (page 251) and *rtems_clock_get_realtime_coarse_timeval()* (page 252) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.8 rtems_clock_get_realtime_coarse_bintime()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in binary time format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime_coarse_bintime( struct bintime *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a `struct bintime` object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_realtime_bintime()* (page 248) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_realtime_coarse()* (page 250) and *rtems_clock_get_realtime_coarse_timeval()* (page 252) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.9 rtems_clock_get_realtime_coarse_timeval()

Gets the time elapsed since the *Unix epoch* measured using *CLOCK_REALTIME* in coarse resolution in seconds and microseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_realtime_coarse_timeval( struct timeval *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timeval object. The time elapsed since the *Unix epoch* measured using the *CLOCK_REALTIME* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_realtime_timeval()* (page 249) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_realtime_coarse()* (page 250) and *rtems_clock_get_realtime_coarse_timeval()* (page 252) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.10 rtems_clock_get_monotonic()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in seconds and nanoseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic( struct timespec *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timespec object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_monotonic_coarse()* (page 257) directive may be used to get the time with in a lower resolution and with less runtime overhead.

See *rtems_clock_get_monotonic_bintime()* (page 254), *rtems_clock_get_monotonic_sbintime()* (page 255), and *rtems_clock_get_monotonic_timeval()* (page 256) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.11 rtems_clock_get_monotonic_bintime()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in binary time format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic_bintime( struct bintime *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a `struct bintime` object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_monotonic_coarse_bintime()* (page 258) directive may be used to get the time in a lower resolution and with less runtime overhead.

See *rtems_clock_get_monotonic()* (page 253), *rtems_clock_get_monotonic_sbintime()* (page 255), and *rtems_clock_get_monotonic_timeval()* (page 256) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.12 rtems_clock_get_monotonic_sbintime()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in signed binary time format.

CALLING SEQUENCE:

```
int64_t rtems_clock_get_monotonic_sbintime( void );
```

RETURN VALUES:

Returns the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point during the directive call.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system.

See *rtems_clock_get_monotonic()* (page 253), *rtems_clock_get_monotonic_bintime()* (page 254), and *rtems_clock_get_monotonic_timeval()* (page 256) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.13 rtems_clock_get_monotonic_timeval()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in seconds and microseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic_timeval( struct timeval *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timeval object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point during the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive accesses a device provided by the *Clock Driver* to get the time in the highest resolution available to the system. Alternatively, the *rtems_clock_get_monotonic_coarse_timeval()* (page 259) directive may be used to get the time in a lower resolution and with less runtime overhead.

See *rtems_clock_get_monotonic()* (page 253), *rtems_clock_get_monotonic_bintime()* (page 254), and *rtems_clock_get_monotonic_sbintime()* (page 255) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.14 rtems_clock_get_monotonic_coarse()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in seconds and nanoseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic_coarse( struct timespec *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timespec object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_monotonic()* (page 253) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_monotonic_coarse_bintime()* (page 258) and *rtems_clock_get_monotonic_coarse_timeval()* (page 259) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.15 rtems_clock_get_monotonic_coarse_bintime()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in binary time format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic_coarse_bintime( struct bintime *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a `struct bintime` object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_monotonic_bintime()* (page 254) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_monotonic_coarse()* (page 257) and *rtems_clock_get_monotonic_coarse_timeval()* (page 259) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.16 rtems_clock_get_monotonic_coarse_timeval()

Gets the time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* in coarse resolution in seconds and microseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_monotonic_coarse_timeval( struct timeval *time_snapshot );
```

PARAMETERS:

**time_snapshot**
This parameter is the pointer to a struct timeval object. The time elapsed since some fixed time point in the past measured using the *CLOCK_MONOTONIC* at some time point close to the directive call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

The directive does not access a device to get the time. It uses a recent snapshot provided by the *Clock Driver*. Alternatively, the *rtems_clock_get_monotonic_timeval()* (page 256) directive may be used to get the time in a higher resolution and with a higher runtime overhead.

See *rtems_clock_get_monotonic_coarse()* (page 257) and *rtems_clock_get_monotonic_coarse_bintime()* (page 258) to get the time in alternative formats.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.17 rtems_clock_get_boot_time()

Gets the time elapsed since the *Unix epoch* at some time point during system initialization in seconds and nanoseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_boot_time( struct timespec *boot_time );
```

PARAMETERS:

**boot_time**
This parameter is the pointer to a struct timespec object. The time elapsed since the *Unix epoch* at some time point during system initialization call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

See *rtems_clock_get_boot_time_bintime()* (page 261) and *rtems_clock_get_boot_time_timeval()* (page 262) to get the boot time in alternative formats. Setting the *CLOCK_REALTIME* will also set the boot time.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.18 rtems_clock_get_boot_time_bintime()

Gets the time elapsed since the *Unix epoch* at some time point during system initialization in binary time format.

CALLING SEQUENCE:

```
void rtems_clock_get_boot_time_bintime( struct bintime *boot_time );
```

PARAMETERS:

**boot_time**
This parameter is the pointer to a `struct bintime` object. The time elapsed since the *Unix epoch* at some time point during system initialization call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

See *rtems_clock_get_boot_time()* (page 260) and *rtems_clock_get_boot_time_timeval()* (page 262) to get the boot time in alternative formats. Setting the *CLOCK_REALTIME* will also set the boot time.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.19 rtems_clock_get_boot_time_timeval()

Gets the time elapsed since the *Unix epoch* at some time point during system initialization in seconds and microseconds format.

CALLING SEQUENCE:

```
void rtems_clock_get_boot_time_timeval( struct timeval *boot_time );
```

PARAMETERS:

**boot_time**
This parameter is the pointer to a struct timeval object. The time elapsed since the *Unix epoch* at some time point during system initialization call will be stored in this object. Calling the directive with a pointer equal to NULL is undefined behaviour.

NOTES:

See *rtems_clock_get_boot_time()* (page 260) and *rtems_clock_get_boot_time_bintime()* (page 261) to get the boot time in alternative formats. Setting the *CLOCK_REALTIME* will also set the boot time.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.20 rtems_clock_get_seconds_since_epoch()

Gets the seconds elapsed since the *RTEMS epoch* and the current *CLOCK_REALTIME*.

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_get_seconds_since_epoch(
  rtems_interval *seconds_since_rtems_epoch
);
```

PARAMETERS:

**seconds_since_rtems_epoch**
This parameter is the pointer to an *rtems_interval* (page 47) object. When the directive call is successful, the seconds elapsed since the *RTEMS epoch* and the *CLOCK_REALTIME* at some point during the directive call will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `seconds_since_rtems_epoch` parameter was NULL.

**RTEMS_NOT_DEFINED**
The *CLOCK_REALTIME* was not set. It can be set with *rtems_clock_set()* (page 243).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.21 rtems_clock_get_ticks_per_second()

Gets the number of *clock ticks* per second configured for the application.

CALLING SEQUENCE:

```
rtems_interval rtems_clock_get_ticks_per_second( void );
```

RETURN VALUES:

Returns the number of clock ticks per second configured for this application.

NOTES:

The number of clock ticks per second is defined indirectly by the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 9.4.22 rtems_clock_get_ticks_since_boot()

Gets the number of *clock ticks* since some time point during the system initialization or the last overflow of the clock tick counter.

CALLING SEQUENCE:

```
rtems_interval rtems_clock_get_ticks_since_boot( void );
```

RETURN VALUES:

Returns the number of *clock ticks* since some time point during the system initialization or the last overflow of the clock tick counter.

NOTES:

With a 1ms clock tick, this counter overflows after 50 days since boot. This is the historical measure of uptime in an RTEMS system. The newer service *rtems_clock_get_uptime()* (page 266) is another and potentially more accurate way of obtaining similar information.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 9.4.23 rtems_clock_get_uptime()

Gets the seconds and nanoseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.

CALLING SEQUENCE:

```
rtems_status_code rtems_clock_get_uptime( struct timespec *uptime );
```

PARAMETERS:

**uptime**
This parameter is the pointer to a struct timespec object. When the directive call is successful, the seconds and nanoseconds elapsed since some time point during the system initialization and some point during the directive call using *CLOCK_MONOTONIC* will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `uptime` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.24 rtems_clock_get_uptime_timeval()

Gets the seconds and microseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.

CALLING SEQUENCE:

```
void rtems_clock_get_uptime_timeval( struct timeval *uptime );
```

PARAMETERS:

**uptime**
This parameter is the pointer to a struct timeval object. The seconds and microseconds elapsed since some time point during the system initialization and some point during the directive call using *CLOCK_MONOTONIC* will be stored in this object. The pointer shall be valid, otherwise the behaviour is undefined.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.25 rtems_clock_get_uptime_seconds()

Gets the seconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.

CALLING SEQUENCE:

```
time_t rtems_clock_get_uptime_seconds( void );
```

RETURN VALUES:

Returns the seconds elapsed since some time point during the system initialization and some point during the directive call using *CLOCK_MONOTONIC*.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.26 rtems_clock_get_uptime_nanoseconds()

Gets the nanoseconds elapsed since some time point during the system initialization using *CLOCK_MONOTONIC*.

CALLING SEQUENCE:

```
uint64_t rtems_clock_get_uptime_nanoseconds( void );
```

RETURN VALUES:

Returns the nanoseconds elapsed since some time point during the system initialization and some point during the directive call using *CLOCK_MONOTONIC*.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.27 rtems_clock_tick_later()

Gets a *clock tick* value which is at least delta clock ticks in the future.

CALLING SEQUENCE:

```
rtems_interval rtems_clock_tick_later( rtems_interval delta );
```

PARAMETERS:

**delta**
This parameter is the delta value in clock ticks.

RETURN VALUES:

Returns a *clock tick* counter value which is at least `delta` clock ticks in the future.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.28 rtems_clock_tick_later_usec()

Gets a *clock tick* value which is at least delta microseconds in the future.

CALLING SEQUENCE:

```
rtems_interval rtems_clock_tick_later_usec( rtems_interval delta_in_usec );
```

PARAMETERS:

**delta_in_usec**
This parameter is the delta value in microseconds.

RETURN VALUES:

Returns a *clock tick* counter value which is at least `delta_in_usec` microseconds in the future.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

### 9.4.29 rtems_clock_tick_before()

Indicates if the current *clock tick* counter is before the ticks.

CALLING SEQUENCE:

```
bool rtems_clock_tick_before( rtems_interval ticks );
```

PARAMETERS:

**ticks**
This parameter is the ticks value to check.

RETURN VALUES:

Returns true, if current *clock tick* counter indicates a time before the time in ticks, otherwise returns false.

NOTES:

This directive can be used to write busy loops with a timeout.

```
status busy( void )
{
  rtems_interval timeout;

  timeout = rtems_clock_tick_later_usec( 10000 );

  do {
    if ( ok() ) {
      return success;
    }
  } while ( rtems_clock_tick_before( timeout ) );

  return timeout;
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- The directive requires a *Clock Driver*.

## 9.5 Removed Directives

### 9.5.1 CLOCK_GET - Get date and time information

**Warning**

This directive was removed in RTEMS 5.1. See also *Transition Advice for the Removed rtems_clock_get()* (page 240).

**CALLING SEQUENCE:**

```
rtems_status_code rtems_clock_get(
   rtems_clock_get_options  option,
   void                    *time_buffer
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_SUCCESSFUL | current time obtained successfully |
| RTEMS_NOT_DEFINED | system date and time is not set |
| RTEMS_INVALID_ADDRESS | time_buffer is NULL |

**DESCRIPTION:**

This directive obtains the system date and time. If the caller is attempting to obtain the date and time (i.e. option is set to either RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH, RTEMS_CLOCK_GET_TOD, or RTEMS_CLOCK_GET_TIME_VALUE) and the date and time has not been set with a previous call to rtems_clock_set, then the RTEMS_NOT_DEFINED status code is returned. The caller can always obtain the number of ticks per second (option is RTEMS_CLOCK_GET_TICKS_PER_SECOND) and the number of ticks since the executive was initialized option is RTEMS_CLOCK_GET_TICKS_SINCE_BOOT).

The option argument may taken on any value of the enumerated type rtems_clock_get_options. The data type expected for time_buffer is based on the value of option as indicated below:

| Option | Return type |
|---|---|
| RTEMS_CLOCK_GET_TOD | (rtems_time_of_day *) |
| RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH | (rtems_interval *) |
| RTEMS_CLOCK_GET_TICKS_SINCE_BOOT | (rtems_interval *) |
| RTEMS_CLOCK_GET_TICKS_PER_SECOND | (rtems_interval *) |
| RTEMS_CLOCK_GET_TIME_VALUE | (struct timeval *) |

**NOTES:**

This directive is callable from an ISR.

This directive will not cause the running task to be preempted. Re-initializing RTEMS causes the system date and time to be reset to an uninitialized state. Another call to rtems_clock_set is required to re-initialize the system date and time to application specific specifications.

CHAPTER

**TEN**

# TIMER MANAGER

## 10.1 Introduction

The Timer Manager provides support for timer facilities. The directives provided by the Timer Manager are:

- *rtems_timer_create()* (page 281) - Creates a timer.
- *rtems_timer_ident()* (page 283) - Identifies a timer by the object name.
- *rtems_timer_cancel()* (page 284) - Cancels the timer.
- *rtems_timer_delete()* (page 285) - Deletes the timer.
- *rtems_timer_fire_after()* (page 286) - Fires the timer after the interval.
- *rtems_timer_fire_when()* (page 288) - Fires the timer at the time of day.
- *rtems_timer_initiate_server()* (page 290) - Initiates the Timer Server.
- *rtems_timer_server_fire_after()* (page 292) - Fires the timer after the interval using the Timer Server.
- *rtems_timer_server_fire_when()* (page 294) - Fires the timer at the time of day using the Timer Server.
- *rtems_timer_reset()* (page 296) - Resets the timer.
- *rtems_timer_get_information()* (page 297) - Gets information about the timer.

## 10.2 Background

### 10.2.1 Required Support

A clock tick is required to support the functionality provided by this manager.

### 10.2.2 Timers

A timer is an RTEMS object which allows the application to schedule operations to occur at specific times in the future. User supplied timer service routines are invoked by either a clock tick directive or a special Timer Server task when the timer fires. Timer service routines may perform any operations or directives which normally would be performed by the application code which invoked a clock tick directive.

The timer can be used to implement watchdog routines which only fire to denote that an application error has occurred. The timer is reset at specific points in the application to ensure that the watchdog does not fire. Thus, if the application does not reset the watchdog timer, then the timer service routine will fire to indicate that the application has failed to reach a reset point. This use of a timer is sometimes referred to as a "keep alive" or a "deadman" timer.

### 10.2.3 Timer Server

The Timer Server task is responsible for executing the timer service routines associated with all task-based timers. This task executes at a priority specified by rtems_timer_initiate_server() and it may have a priority of zero (the highest priority). In uniprocessor configurations, it is created non-preemptible.

By providing a mechanism where timer service routines execute in task rather than interrupt space, the application is allowed a bit more flexibility in what operations a timer service routine can perform. For example, the Timer Server can be configured to have a floating point context in which case it would be safe to perform floating point operations from a task-based timer. Most of the time, executing floating point instructions from an interrupt service routine is not considered safe. The timer service routines invoked by the Timer Server may block, however, since this blocks the Timer Server itself, other timer service routines that are already pending do not run until the blocked timer service routine finished its work.

The Timer Server is designed to remain blocked until a task-based timer fires. This reduces the execution overhead of the Timer Server.

### 10.2.4 Timer Service Routines

The timer service routine should adhere to C calling conventions and have a prototype similar to the following:

```
rtems_timer_service_routine user_routine(
    rtems_id   timer_id,
    void      *user_data
);
```

Where the timer_id parameter is the RTEMS object ID of the timer which is being fired and user_data is a pointer to user-defined information which may be utilized by the timer service routine. The argument user_data may be NULL.

## 10.3 Operations

### 10.3.1 Creating a Timer

The `rtems_timer_create` directive creates a timer by allocating a Timer Control Block (TMCB), assigning the timer a user-specified name, and assigning it a timer ID. Newly created timers do not have a timer service routine associated with them and are not active.

### 10.3.2 Obtaining Timer IDs

When a timer is created, RTEMS generates a unique timer ID and assigns it to the created timer until it is deleted. The timer ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_timer_create` directive, the timer ID is stored in a user provided location. Second, the timer ID may be obtained later using the `rtems_timer_ident` directive. The timer ID is used by other directives to manipulate this timer.

### 10.3.3 Initiating an Interval Timer

The `rtems_timer_fire_after` and `rtems_timer_server_fire_after` directives initiate a timer to fire a user provided timer service routine after the specified number of clock ticks have elapsed. When the interval has elapsed, the timer service routine will be invoked from a clock tick directive if it was initiated by the `rtems_timer_fire_after` directive and from the Timer Server task if initiated by the `rtems_timer_server_fire_after` directive.

### 10.3.4 Initiating a Time of Day Timer

The `rtems_timer_fire_when` and `rtems_timer_server_fire_when` directive initiate a timer to fire a user provided timer service routine when the specified time of day has been reached. When the interval has elapsed, the timer service routine will be invoked from a clock tick directive by the `rtems_timer_fire_when` directive and from the Timer Server task if initiated by the `rtems_timer_server_fire_when` directive.

### 10.3.5 Canceling a Timer

The `rtems_timer_cancel` directive is used to halt the specified timer. Once canceled, the timer service routine will not fire unless the timer is reinitiated. The timer can be reinitiated using the `rtems_timer_reset`, `rtems_timer_fire_after`, and `rtems_timer_fire_when` directives.

### 10.3.6 Resetting a Timer

The `rtems_timer_reset` directive is used to restore an interval timer initiated by a previous invocation of `rtems_timer_fire_after` or `rtems_timer_server_fire_after` to its original interval length. If the timer has not been used or the last usage of this timer was by the `rtems_timer_fire_when` or `rtems_timer_server_fire_when` directive, then an error is returned. The timer service routine is not changed or fired by this directive.

### 10.3.7 Initiating the Timer Server

The `rtems_timer_initiate_server` directive is used to allocate and start the execution of the Timer Server task. The application can specify both the stack size and attributes of the Timer Server. The Timer Server executes at a priority higher than any application task and thus the user can expect to be preempted as the result of executing the `rtems_timer_initiate_server` directive.

### 10.3.8 Deleting a Timer

The `rtems_timer_delete` directive is used to delete a timer. If the timer is running and has not expired, the timer is automatically canceled. The timer's control block is returned to the TMCB free list when it is deleted. A timer can be deleted by a task other than the task which created the timer. Any subsequent references to the timer's name and ID are invalid.

## 10.4 Directives

This section details the directives of the Timer Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 10.4.1 rtems_timer_create()

Creates a timer.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_create( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name of the timer.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created timer will be stored in this object.

DESCRIPTION:

This directive creates a timer which resides on the local node. The timer has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the timer with other timer related directives.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_TOO_MANY**
There was no inactive object available to create a timer. The number of timers available to the application is configured through the *CONFIGURE_MAXIMUM_TIMERS* (page 622) application configuration option.

NOTES:

The processor used to maintain the timer is the processor of the calling task at some point during the timer creation.

For control and maintenance of the timer, RTEMS allocates a *TMCB* from the local TMCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

- The number of timers available to the application is configured through the *CONFIGURE_MAXIMUM_TIMERS* (page 622) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 10.4.2 rtems_timer_ident()

Identifies a timer by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a timer identifier associated with the timer name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the timer name is not unique, then the timer identifier will match the first timer with that name in the search order. However, this timer identifier is not guaranteed to correspond to the desired timer.

The objects are searched from lowest to the highest index. Only the local node is searched.

The timer identifier is used with other timer related directives to access the timer.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 10.4.3 rtems_timer_cancel()

Cancels the timer.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_cancel( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the timer identifier.

DESCRIPTION:

This directive cancels the timer specified by `id`. This timer will be reinitiated by the next invocation of *rtems_timer_reset()* (page 296), *rtems_timer_fire_after()* (page 286), *rtems_timer_fire_when()* (page 288), *rtems_timer_server_fire_after()* (page 292), or *rtems_timer_server_fire_when()* (page 294) with the same timer identifier.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 10.4.4 rtems_timer_delete()

Deletes the timer.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the timer identifier.

DESCRIPTION:

This directive deletes the timer specified by `id`. If the timer is running, it is automatically canceled.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

NOTES:

The *TMCB* for the deleted timer is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 10.4.5 rtems_timer_fire_after()

Fires the timer after the interval.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_fire_after(
  rtems_id                          id,
  rtems_interval                    ticks,
  rtems_timer_service_routine_entry routine,
  void                             *user_data
);
```

PARAMETERS:

**id**
This parameter is the timer identifier.

**ticks**
This parameter is the interval until the routine is fired in clock ticks.

**routine**
This parameter is the routine to schedule.

**user_data**
This parameter is the argument passed to the routine when it is fired.

DESCRIPTION:

This directive initiates the timer specified by `id`. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire after an interval of clock ticks has passed specified by `ticks`. When the timer fires, the timer service routine `routine` will be invoked with the argument `user_data` in the context of the clock tick *ISR*.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The `ticks` parameter was 0.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

- The directive will not cause the calling task to be preempted.

### 10.4.6 rtems_timer_fire_when()

Fires the timer at the time of day.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_fire_when(
  rtems_id                          id,
  const rtems_time_of_day          *wall_time,
  rtems_timer_service_routine_entry routine,
  void                             *user_data
);
```

PARAMETERS:

**id**
This parameter is the timer identifier.

**wall_time**
This parameter is the time of day when the routine is fired.

**routine**
This parameter is the routine to schedule.

**user_data**
This parameter is the argument passed to the routine when it is fired.

DESCRIPTION:

This directive initiates the timer specified by `id`. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire at the time of day specified by `wall_time`. When the timer fires, the timer service routine `routine` will be invoked with the argument `user_data` in the context of the clock tick *ISR*.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_NOT_DEFINED**
The system date and time was not set.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `wall_time` parameter was NULL.

**RTEMS_INVALID_CLOCK**
The time of day was invalid.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 10.4.7 rtems_timer_initiate_server()

Initiates the Timer Server.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_initiate_server(
  rtems_task_priority priority,
  size_t              stack_size,
  rtems_attribute     attribute_set
);
```

PARAMETERS:

**priority**
This parameter is the task priority.

**stack_size**
This parameter is the task stack size in bytes.

**attribute_set**
This parameter is the task attribute set.

DESCRIPTION:

This directive initiates the Timer Server task. This task is responsible for executing all timers initiated via the *rtems_timer_server_fire_after()* (page 292) or *rtems_timer_server_fire_when()* (page 294) directives.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The Timer Server was already initiated.

**RTEMS_INVALID_PRIORITY**
The task priority was invalid.

**RTEMS_TOO_MANY**
There was no inactive task object available to create the Timer Server task.

**RTEMS_UNSATISFIED**
There was not enough memory to allocate the task storage area. The task storage area contains the task stack, the thread-local storage, and the floating point context.

**RTEMS_UNSATISFIED**
One of the task create extensions failed to create the Timer Server task.

NOTES:

The Timer Server task is created using the *rtems_task_create()* (page 109) directive and must be accounted for when configuring the system.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The number of timers available to the application is configured through the *CONFIGURE_MAXIMUM_TIMERS* (page 622) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 10.4.8 rtems_timer_server_fire_after()

Fires the timer after the interval using the Timer Server.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_server_fire_after(
  rtems_id                          id,
  rtems_interval                    ticks,
  rtems_timer_service_routine_entry routine,
  void                             *user_data
);
```

PARAMETERS:

**id**
This parameter is the timer identifier.

**ticks**
This parameter is the interval until the routine is fired in clock ticks.

**routine**
This parameter is the routine to schedule.

**user_data**
This parameter is the argument passed to the routine when it is fired.

DESCRIPTION:

This directive initiates the timer specified by `id`. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire after an interval of clock ticks has passed specified by `ticks`. When the timer fires, the timer service routine `routine` will be invoked with the argument `user_data` in the context of the Timer Server task.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The Timer Server was not initiated.

**RTEMS_INVALID_NUMBER**
The `ticks` parameter was 0.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 10.4.9 rtems_timer_server_fire_when()

Fires the timer at the time of day using the Timer Server.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_server_fire_when(
  rtems_id                          id,
  const rtems_time_of_day          *wall_time,
  rtems_timer_service_routine_entry routine,
  void                             *user_data
);
```

PARAMETERS:

**id**
This parameter is the timer identifier.

**wall_time**
This parameter is the time of day when the routine is fired.

**routine**
This parameter is the routine to schedule.

**user_data**
This parameter is the argument passed to the routine when it is fired.

DESCRIPTION:

This directive initiates the timer specified by `id`. If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire at the time of day specified by `wall_time`. When the timer fires, the timer service routine `routine` will be invoked with the argument `user_data` in the context of the Timer Server task.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INCORRECT_STATE**
The Timer Server was not initiated.

**RTEMS_NOT_DEFINED**
The system date and time was not set.

**RTEMS_INVALID_ADDRESS**
The `routine` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `wall_time` parameter was NULL.

**RTEMS_INVALID_CLOCK**
The time of day was invalid.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 10.4.10 rtems_timer_reset()

Resets the timer.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_reset( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the timer identifier.

DESCRIPTION:

This directive resets the timer specified by `id`. This timer must have been previously initiated with either the *rtems_timer_fire_after()* (page 286) or *rtems_timer_server_fire_after()* (page 292) directive. If active the timer is canceled, after which the timer is reinitiated using the same interval and timer service routine which the original *rtems_timer_fire_after()* (page 286) or *rtems_timer_server_fire_after()* (page 292) directive used.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

**RTEMS_NOT_DEFINED**
The timer was not of the interval class.

NOTES:

If the timer has not been used or the last usage of this timer was by a *rtems_timer_fire_when()* (page 288) or *rtems_timer_server_fire_when()* (page 294) directive, then the `RTEMS_NOT_DEFINED` error is returned.

Restarting a cancelled after timer results in the timer being reinitiated with its previous timer service routine and interval.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 10.4.11 rtems_timer_get_information()

Gets information about the timer.

CALLING SEQUENCE:

```
rtems_status_code rtems_timer_get_information(
  rtems_id                id,
  rtems_timer_information *the_info
);
```

PARAMETERS:

**id**
This parameter is the timer identifier.

**the_info**
This parameter is the pointer to an *rtems_timer_information* (page 59) object. When the directive call is successful, the information about the timer will be stored in this object.

DESCRIPTION:

This directive returns information about the timer.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `the_info` parameter was NULL.

**RTEMS_INVALID_ID**
There was no timer associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

CHAPTER

**ELEVEN**

# RATE MONOTONIC MANAGER

## 11.1 Introduction

The Rate-Monotonic Manager provides facilities to implement tasks which execute in a periodic fashion. Critically, it also gathers information about the execution of those periods and can provide important statistics to the user which can be used to analyze and tune the application. The directives provided by the Rate-Monotonic Manager are:

- *rtems_rate_monotonic_create()* (page 312) - Creates a period.
- *rtems_rate_monotonic_ident()* (page 314) - Identifies a period by the object name.
- *rtems_rate_monotonic_cancel()* (page 315) - Cancels the period.
- *rtems_rate_monotonic_delete()* (page 316) - Deletes the period.
- *rtems_rate_monotonic_period()* (page 317) - Concludes the current period and start the next period, or gets the period status.
- *rtems_rate_monotonic_get_status()* (page 319) - Gets the detailed status of the period.
- *rtems_rate_monotonic_get_statistics()* (page 321) - Gets the statistics of the period.
- *rtems_rate_monotonic_reset_statistics()* (page 323) - Resets the statistics of the period.
- *rtems_rate_monotonic_reset_all_statistics()* (page 324) - Resets the statistics of all periods.
- *rtems_rate_monotonic_report_statistics()* (page 325) - Reports the period statistics using the *printk()* (page 491) printer.
- *rtems_rate_monotonic_report_statistics_with_plugin()* (page 326) - Reports the period statistics using the printer plugin.

## 11.2 Background

The rate monotonic manager provides facilities to manage the execution of periodic tasks. This manager was designed to support application designers who utilize the Rate Monotonic Scheduling Algorithm (RMS) to ensure that their periodic tasks will meet their deadlines, even under transient overload conditions. Although designed for hard real-time systems, the services provided by the rate monotonic manager may be used by any application which requires periodic tasks.

### 11.2.1 Rate Monotonic Manager Required Support

A clock tick is required to support the functionality provided by this manager.

### 11.2.2 Period Statistics

This manager maintains a set of statistics on each period object. These statistics are reset implictly at period creation time and may be reset or obtained at any time by the application. The following is a list of the information kept:

**owner**
is the id of the thread that owns this period.

**count**
is the total number of periods executed.

**missed_count**
is the number of periods that were missed.

**min_cpu_time**
is the minimum amount of CPU execution time consumed on any execution of the periodic loop.

**max_cpu_time**
is the maximum amount of CPU execution time consumed on any execution of the periodic loop.

**total_cpu_time**
is the total amount of CPU execution time consumed by executions of the periodic loop.

**min_wall_time**
is the minimum amount of wall time that passed on any execution of the periodic loop.

**max_wall_time**
is the maximum amount of wall time that passed on any execution of the periodic loop.

**total_wall_time**
is the total amount of wall time that passed during executions of the periodic loop.

Each period is divided into two consecutive phases. The period starts with the active phase of the task and is followed by the inactive phase of the task. In the inactive phase the task is blocked and waits for the start of the next period. The inactive phase is skipped in case of a period miss. The wall time includes the time during the active phase of the task on which the task is not executing on a processor. The task is either blocked (for example it waits for a resource) or a higher priority tasks executes, thus preventing it from executing. In case the wall time exceeds the period time, then this is a period miss. The gap between the wall time and the period time is the margin between a period miss or success.

The period statistics information is inexpensive to maintain and can provide very useful insights into the execution characteristics of a periodic task loop. But it is just information. The period statistics reported must be analyzed by the user in terms of what the applications is. For example, in an application where priorities are assigned by the Rate Monotonic Algorithm, it would be very undesirable for high priority (i.e. frequency) tasks to miss their period. Similarly, in nearly any application, if a task were supposed to execute its periodic loop every 10 milliseconds and it averaged 11 milliseconds, then application requirements are not being met.

The information reported can be used to determine the "hot spots" in the application. Given a period's id, the user can determine the length of that period. From that information and the CPU usage, the user can calculate the percentage of CPU time consumed by that periodic task. For example, a task executing for 20 milliseconds every 200 milliseconds is consuming 10 percent of the processor's execution time. This is usually enough to make it a good candidate for optimization.

However, execution time alone is not enough to gauge the value of optimizing a particular task. It is more important to optimize a task executing 2 millisecond every 10 milliseconds (20 percent of the CPU) than one executing 10 milliseconds every 100 (10 percent of the CPU). As a general rule of thumb, the higher frequency at which a task executes, the more important it is to optimize that task.

### 11.2.3 Periodicity Definitions

A periodic task is one which must be executed at a regular interval. The interval between successive iterations of the task is referred to as its period. Periodic tasks can be characterized by the length of their period and execution time. The period and execution time of a task can be used to determine the processor utilization for that task. Processor utilization is the percentage of processor time used and can be calculated on a per-task or system-wide basis. Typically, the task's worst-case execution time will be less than its period. For example, a periodic task's requirements may state that it should execute for 10 milliseconds every 100 milliseconds. Although the execution time may be the average, worst, or best case, the worst-case execution time is more appropriate for use when analyzing system behavior under transient overload conditions.

In contrast, an aperiodic task executes at irregular intervals and has only a soft deadline. In other words, the deadlines for aperiodic tasks are not rigid, but adequate response times are desirable. For example, an aperiodic task may process user input from a terminal.

Finally, a sporadic task is an aperiodic task with a hard deadline and minimum interarrival time. The minimum interarrival time is the minimum period of time which exists between successive iterations of the task. For example, a sporadic task could be used to process the pressing of a fire button on a joystick. The mechanical action of the fire button ensures a minimum time period between successive activations, but the missile must be launched by a hard deadline.

### 11.2.4 Rate Monotonic Scheduling Algorithm

The Rate Monotonic Scheduling Algorithm (RMS) is important to real-time systems designers because it allows one to sufficiently guarantee that a set of tasks is schedulable (see [LL73], [LSD89], [SG90], [Bur91]).

A set of tasks is said to be schedulable if all of the tasks can meet their deadlines. RMS provides a set of rules which can be used to perform a guaranteed schedulability analysis for a task set. This analysis determines whether a task set is schedulable under worst-case conditions and emphasizes the predictability of the system's behavior. It has been proven that:

***RMS***

RMS is an optimal fixed-priority algorithm for scheduling independent, preemptible, periodic tasks on a single processor.

RMS is optimal in the sense that if a set of tasks can be scheduled by any fixed-priority algorithm, then RMS will be able to schedule that task set. RMS bases it schedulability analysis on the processor utilization level below which all deadlines can be met.

RMS calls for the static assignment of task priorities based upon their period. The shorter a task's period, the higher its priority. For example, a task with a 1 millisecond period has higher priority than a task with a 100 millisecond period. If two tasks have the same period, then RMS does not distinguish between the tasks. However, RTEMS specifies that when given tasks of equal priority, the task which has been ready longest will execute first. RMS's priority assignment scheme does not provide one with exact numeric values for task priorities. For example, consider the following task set and priority assignments:

| Task | Period (in milliseconds) | Priority |
|---|---|---|
| 1 | 100 | Low |
| 2 | 50 | Medium |
| 3 | 50 | Medium |
| 4 | 25 | High |

RMS only calls for task 1 to have the lowest priority, task 4 to have the highest priority, and tasks 2 and 3 to have an equal priority between that of tasks 1 and 4. The actual RTEMS priorities assigned to the tasks must only adhere to those guidelines.

Many applications have tasks with both hard and soft deadlines. The tasks with hard deadlines are typically referred to as the critical task set, with the soft deadline tasks being the non-critical task set. The critical task set can be scheduled using RMS, with the non-critical tasks not executing under transient overload, by simply assigning priorities such that the lowest priority critical task (i.e. longest period) has a higher priority than the highest priority non-critical task. Although RMS may be used to assign priorities to the non-critical tasks, it is not necessary. In this instance, schedulability is only guaranteed for the critical task set.

### 11.2.5 Schedulability Analysis

RMS allows application designers to ensure that tasks can meet all deadlines under fixed-priority assignment, even under transient overload, without knowing exactly when any given task will execute by applying proven schedulability analysis rules.

#### 11.2.5.1 Assumptions

The schedulability analysis rules for RMS were developed based on the following assumptions:

- The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests.
- Each task must complete before the next request for it occurs.
- The tasks are independent in that a task does not depend on the initiation or completion of requests for other tasks.

- The execution time for each task without preemption or interruption is constant and does not vary.
- Any non-periodic tasks in the system are special. These tasks should not displace periodic tasks while executing and do not have hard, critical deadlines.

Once the basic schedulability analysis is understood, some of the above assumptions can be relaxed and the side-effects accounted for.

#### 11.2.5.2 Processor Utilization Rule

The Processor Utilization Rule requires that processor utilization be calculated based upon the period and execution time of each task. The fraction of processor time spent executing task index is `Time(i) / Period(i)`. The processor utilization can be calculated as follows where n is the number of tasks in the set being analyzed:

$$Utilization = \sum_{i=1}^{n} Time_i / Period_i$$

To ensure schedulability even under transient overload, the processor utilization must adhere to the following rule:

$$maximumUtilization = n * (2^{\frac{1}{n}} - 1)$$

As the number of tasks increases, the above formula approaches ln(2) for a worst-case utilization factor of approximately 0.693. Many tasks sets can be scheduled with a greater utilization factor. In fact, the average processor utilization threshold for a randomly generated task set is approximately 0.88. See more detail in [LL73].

#### 11.2.5.3 Processor Utilization Rule Example

This example illustrates the application of the Processor Utilization Rule to an application with three critical periodic tasks. The following table details the RMS priority, period, execution time, and processor utilization for each task:

| Task | RMS Priority | Period | Execution Time | Processor Utilization |
|---|---|---|---|---|
| 1 | High | 100 | 15 | 0.15 |
| 2 | Medium | 200 | 50 | 0.25 |
| 3 | Low | 300 | 100 | 0.33 |

The total processor utilization for this task set is 0.73 which is below the upper bound of 3 * (2**(1/3) - 1), or 0.779, imposed by the Processor Utilization Rule. Therefore, this task set is guaranteed to be schedulable using RMS.

#### 11.2.5.4 First Deadline Rule

If a given set of tasks do exceed the processor utilization upper limit imposed by the Processor Utilization Rule, they can still be guaranteed to meet all their deadlines by application of the First Deadline Rule. This rule can be stated as follows:

> For a given set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times.

A key point with this rule is that ALL periodic tasks are assumed to start at the exact same instant in time. Although this assumption may seem to be invalid, RTEMS makes it quite easy to ensure. By having a non-preemptible user initialization task, all application tasks, regardless of priority, can be created and started before the initialization deletes itself. This technique ensures that all tasks begin to compete for execution time at the same instant - when the user initialization task deletes itself. See more detail in [LSD89].

#### 11.2.5.5 First Deadline Rule Example

The First Deadline Rule can ensure schedulability even when the Processor Utilization Rule fails. The example below is a modification of the Processor Utilization Rule example where task execution time has been increased from 15 to 25 units. The following table details the RMS priority, period, execution time, and processor utilization for each task:

| Task | RMS Priority | Period | Execution Time | Processor Utilization |
|---|---|---|---|---|
| 1 | High | 100 | 25 | 0.25 |
| 2 | Medium | 200 | 50 | 0.25 |
| 3 | Low | 300 | 100 | 0.33 |

The total processor utilization for the modified task set is 0.83 which is above the upper bound of 3 * (2**(1/3) - 1), or 0.779, imposed by the Processor Utilization Rule. Therefore, this task set is not guaranteed to be schedulable using RMS. However, the First Deadline Rule can guarantee the schedulability of this task set. This rule calls for one to examine each occurrence of deadline until either all tasks have met their deadline or one task failed to meet its first deadline. The following table details the time of each deadline occurrence, the maximum number of times each task may have run, the total execution time, and whether all the deadlines have been met:

| Deadline Time | Task 1 | Task 2 | Task 3 | Total Execution Time | All Deadlines Met? |
|---|---|---|---|---|---|
| 100 | 1 | 1 | 1 | 25 + 50 + 100 = 175 | NO |
| 200 | 2 | 1 | 1 | 50 + 50 + 100 = 200 | YES |

The key to this analysis is to recognize when each task will execute. For example at time 100, task 1 must have met its first deadline, but tasks 2 and 3 may also have begun execution. In this example, at time 100 tasks 1 and 2 have completed execution and thus have met their first deadline. Tasks 1 and 2 have used (25 + 50) = 75 time units, leaving (100 - 75) = 25 time units for task 3 to begin. Because task 3 takes 100 ticks to execute, it will not have completed execution at time 100. Thus at time 100, all of the tasks except task 3 have met their first deadline.

At time 200, task 1 must have met its second deadline and task 2 its first deadline. As a result, of the first 200 time units, task 1 uses (2 * 25) = 50 and task 2 uses 50, leaving (200 - 100) time units for task 3. Task 3 requires 100 time units to execute, thus it will have completed execution at time 200. Thus, all of the tasks have met their first deadlines at time 200, and the task set is schedulable using the First Deadline Rule.

#### 11.2.5.6 Relaxation of Assumptions

The assumptions used to develop the RMS schedulability rules are uncommon in most real-time systems. For example, it was assumed that tasks have constant unvarying execution time. It is possible to relax this assumption, simply by using the worst-case execution time of each task.

Another assumption is that the tasks are independent. This means that the tasks do not wait for one another or contend for resources. This assumption can be relaxed by accounting for the amount of time a task spends waiting to acquire resources. Similarly, each task's execution time must account for any I/O performed and any RTEMS directive calls.

In addition, the assumptions did not account for the time spent executing interrupt service routines. This can be accounted for by including all the processor utilization by interrupt service routines in the utilization calculation. Similarly, one should also account for the impact of delays in accessing local memory caused by direct memory access and other processors accessing local dual-ported memory.

The assumption that nonperiodic tasks are used only for initialization or failure-recovery can be relaxed by placing all periodic tasks in the critical task set. This task set can be scheduled and analyzed using RMS. All nonperiodic tasks are placed in the non-critical task set. Although the critical task set can be guaranteed to execute even under transient overload, the non-critical task set is not guaranteed to execute.

In conclusion, the application designer must be fully cognizant of the system and its run-time behavior when performing schedulability analysis for a system using RMS. Every hardware and software factor which impacts the execution time of each task must be accounted for in the schedulability analysis.

## 11.3 Operations

### 11.3.1 Creating a Rate Monotonic Period

The `rtems_rate_monotonic_create` directive creates a rate monotonic period which is to be used by the calling task to delineate a period. RTEMS allocates a Period Control Block (PCB) from the PCB free list. This data structure is used by RTEMS to manage the newly created rate monotonic period. RTEMS returns a unique period ID to the application which is used by other rate monotonic manager directives to access this rate monotonic period.

### 11.3.2 Manipulating a Period

The `rtems_rate_monotonic_period` directive is used to establish and maintain periodic execution utilizing a previously created rate monotonic period. Once initiated by the `rtems_rate_monotonic_period` directive, the period is said to run until it either expires or is reinitiated. The state of the rate monotonic period results in one of the following scenarios:

- If the rate monotonic period is running, the calling task will be blocked for the remainder of the outstanding period and, upon completion of that period, the period will be reinitiated with the specified period.
- If the rate monotonic period is not currently running and has not expired, it is initiated with a length of period ticks and the calling task returns immediately.
- If the rate monotonic period has expired before the task invokes the `rtems_rate_monotonic_period` directive, the postponed job will be released until there is no more postponed jobs. The calling task returns immediately with a timeout error status. In the watchdog routine, the period will still be updated periodically and track the count of the postponed jobs [CvdBruggenC16]. Please note, the count of the postponed jobs is only saturated until 0xffffffff.

### 11.3.3 Obtaining the Status of a Period

If the `rtems_rate_monotonic_period` directive is invoked with a period of `RTEMS_PERIOD_STATUS` ticks, the current state of the specified rate monotonic period will be returned. The following table details the relationship between the period's status and the directive status code returned by the `rtems_rate_monotonic_period` directive:

| | |
|---|---|
| `RTEMS_SUCCESSFUL` | period is running |
| `RTEMS_TIMEOUT` | period has expired |
| `RTEMS_NOT_DEFINED` | period has never been initiated |

Obtaining the status of a rate monotonic period does not alter the state or length of that period.

### 11.3.4 Canceling a Period

The `rtems_rate_monotonic_cancel` directive is used to stop the period maintained by the specified rate monotonic period. The period is stopped and the rate monotonic period can be reinitiated using the `rtems_rate_monotonic_period` directive.

### 11.3.5 Deleting a Rate Monotonic Period

The `rtems_rate_monotonic_delete` directive is used to delete a rate monotonic period. If the period is running and has not expired, the period is automatically canceled. The rate monotonic period's control block is returned to the PCB free list when it is deleted. A rate monotonic period can be deleted by a task other than the task which created the period.

### 11.3.6 Examples

The following sections illustrate common uses of rate monotonic periods to construct periodic tasks.

### 11.3.7 Simple Periodic Task

This example consists of a single periodic task which, after initialization, executes every 100 clock ticks.

```
rtems_task Periodic_task(rtems_task_argument arg)
{
    rtems_name        name;
    rtems_id          period;
    rtems_status_code status;
    name = rtems_build_name( 'P', 'E', 'R', 'D' );
    status = rtems_rate_monotonic_create( name, &period );
    if ( status != RTEMS_SUCCESSFUL ) {
        printf( "rtems_monotonic_create failed with status of %d.\n", status );
        exit( 1 );
    }
    while ( 1 ) {
        if ( rtems_rate_monotonic_period( period, 100 ) == RTEMS_TIMEOUT )
            break;
        /* Perform some periodic actions */
    }
    /* missed period so delete period and SELF */
    status = rtems_rate_monotonic_delete( period );
    if ( status != RTEMS_SUCCESSFUL ) {
        printf( "rtems_rate_monotonic_delete failed with status of %d.\n", status␣
↪);
        exit( 1 );
    }
    status = rtems_task_delete( RTEMS_SELF );    /* should not return */
    printf( "rtems_task_delete returned with status of %d.\n", status );
    exit( 1 );
}
```

The above task creates a rate monotonic period as part of its initialization. The first time the loop is executed, the `rtems_rate_monotonic_period` directive will initiate the period for 100 ticks and return immediately. Subsequent invocations of the `rtems_rate_monotonic_period` directive will result in the task blocking for the remainder of the 100 tick period. If, for any reason, the body of the loop takes more than 100 ticks to execute, the `rtems_rate_monotonic_period` directive will return the `RTEMS_TIMEOUT` status. If the above task misses its deadline, it will delete the rate monotonic period and itself.

### 11.3.8 Task with Multiple Periods

This example consists of a single periodic task which, after initialization, performs two sets of actions every 100 clock ticks. The first set of actions is performed in the first forty clock ticks of every 100 clock ticks, while the second set of actions is performed between the fortieth and seventieth clock ticks. The last thirty clock ticks are not used by this task.

```
rtems_task Periodic_task(rtems_task_argument arg)
{
    rtems_name        name_1, name_2;
    rtems_id          period_1, period_2;
    name_1 = rtems_build_name( 'P', 'E', 'R', '1' );
    name_2 = rtems_build_name( 'P', 'E', 'R', '2' );
    (void ) rtems_rate_monotonic_create( name_1, &period_1 );
    (void ) rtems_rate_monotonic_create( name_2, &period_2 );
    while ( 1 ) {
        if ( rtems_rate_monotonic_period( period_1, 100 ) == RTEMS_TIMEOUT )
            break;
        if ( rtems_rate_monotonic_period( period_2, 40 ) == RTEMS_TIMEOUT )
        break;
        /*
         *  Perform first set of actions between clock
         *  ticks 0 and 39 of every 100 ticks.
         */
        if ( rtems_rate_monotonic_period( period_2, 30 ) == RTEMS_TIMEOUT )
            break;
        /*
         *  Perform second set of actions between clock 40 and 69
         *  of every 100 ticks.  THEN ...
         *
         *  Check to make sure we didn't miss the period_2 period.
         */
        if ( rtems_rate_monotonic_period( period_2, RTEMS_PERIOD_STATUS ) ==
↪RTEMS_TIMEOUT )
            break;
        (void) rtems_rate_monotonic_cancel( period_2 );
    }
    /* missed period so delete period and SELF */
    (void ) rtems_rate_monotonic_delete( period_1 );
    (void ) rtems_rate_monotonic_delete( period_2 );
    (void ) rtems_task_delete( RTEMS_SELF );
}
```

The above task creates two rate monotonic periods as part of its initialization. The first time the loop is executed, the `rtems_rate_monotonic_period` directive will initiate the period_1 period for 100 ticks and return immediately. Subsequent invocations of the `rtems_rate_monotonic_period` directive for period_1 will result in the task blocking for the remainder of the 100 tick period. The period_2 period is used to control the execution time of the two sets of actions within each 100 tick period established by period_1. The `rtems_rate_monotonic_cancel( period_2 )` call is performed to ensure that the period_2 period does not expire while the task is blocked on the period_1 period. If this cancel operation were not performed, every time the `rtems_rate_monotonic_period( period_2, 40 )` call is ex-

ecuted, except for the initial one, a directive status of `RTEMS_TIMEOUT` is returned. It is important to note that every time this call is made, the period_2 period will be initiated immediately and the task will not block.

If, for any reason, the task misses any deadline, the `rtems_rate_monotonic_period` directive will return the `RTEMS_TIMEOUT` directive status. If the above task misses its deadline, it will delete the rate monotonic periods and itself.

## 11.4 Directives

This section details the directives of the Rate-Monotonic Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 11.4.1 rtems_rate_monotonic_create()

Creates a period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_create( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name of the period.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created period will be stored in this object.

DESCRIPTION:

This directive creates a period which resides on the local node. The period has the user-defined object name specified in `name` The assigned object identifier is returned in `id`. This identifier is used to access the period with other rate monotonic related directives.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_TOO_MANY**
There was no inactive object available to create a period. The number of periods available to the application is configured through the *CONFIGURE_MAXIMUM_PERIODS* (page 617) application configuration option.

NOTES:

The calling task is registered as the owner of the created period. Some directives can be only used by this task for the created period.

For control and maintenance of the period, RTEMS allocates a *PCB* from the local PCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of periods available to the application is configured through the *CONFIGURE_MAXIMUM_PERIODS* (page 617) application configuration option.

- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 11.4.2 rtems_rate_monotonic_ident()

Identifies a period by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a period identifier associated with the period name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the period name is not unique, then the period identifier will match the first period with that name in the search order. However, this period identifier is not guaranteed to correspond to the desired period.

The objects are searched from lowest to the highest index. Only the local node is searched.

The period identifier is used with other rate monotonic related directives to access the period.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 11.4.3 rtems_rate_monotonic_cancel()

Cancels the period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_cancel( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the rate monotonic period identifier.

DESCRIPTION:

This directive cancels the rate monotonic period specified by `id`. This period may be reinitiated by the next invocation of *rtems_rate_monotonic_period()* (page 317).

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no rate monotonic period associated with the identifier specified by `id`.

**RTEMS_NOT_OWNER_OF_RESOURCE**
The rate monotonic period was not created by the calling task.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.
- The directive may be used exclusively by the task which created the associated object.

### 11.4.4 rtems_rate_monotonic_delete()

Deletes the period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the period identifier.

DESCRIPTION:

This directive deletes the period specified by `id`. If the period is running, it is automatically canceled.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no period associated with the identifier specified by `id`.

NOTES:

The *PCB* for the deleted period is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 11.4.5 rtems_rate_monotonic_period()

Concludes the current period and start the next period, or gets the period status.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_period(
  rtems_id       id,
  rtems_interval length
);
```

PARAMETERS:

**id**
This parameter is the rate monotonic period identifier.

**length**
This parameter is the period length in *clock ticks* or `RTEMS_PERIOD_STATUS` to get the period status.

DESCRIPTION:

This directive initiates the rate monotonic period specified by `id` with a length of period ticks specified by `length`. If the period is running, then the calling task will block for the remainder of the period before reinitiating the period with the specified period length. If the period was not running (either expired or never initiated), the period is immediately initiated and the directive returns immediately. If the period has expired, the postponed job will be released immediately and the following calls of this directive will release postponed jobs until there is no more deadline miss.

If invoked with a period length of `RTEMS_PERIOD_STATUS` ticks, the current state of the period will be returned. The directive status indicates the current state of the period. This does not alter the state or period length of the period.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no rate monotonic period associated with the identifier specified by `id`.

**RTEMS_NOT_OWNER_OF_RESOURCE**
The rate monotonic period was not created by the calling task.

**RTEMS_NOT_DEFINED**
The rate monotonic period has never been initiated (only possible when the `length` parameter was equal to `RTEMS_PERIOD_STATUS`).

**RTEMS_TIMEOUT**
The rate monotonic period has expired.

NOTES:

Resetting the processor usage time of tasks has no impact on the period status and statistics.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be used exclusively by the task which created the associated object.

### 11.4.6 rtems_rate_monotonic_get_status()

Gets the detailed status of the period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_get_status(
  rtems_id                          id,
  rtems_rate_monotonic_period_status *status
);
```

PARAMETERS:

**id**
This parameter is the rate monotonic period identifier.

**status**
This parameter is the pointer to an *rtems_rate_monotonic_period_status* (page 50) object. When the directive call is successful, the detailed period status will be stored in this object.

DESCRIPTION:

This directive returns the detailed status of the rate monotonic period specified by `id`. The detailed status of the period will be returned in the members of the period status object referenced by `status`:

- The `owner` member is set to the identifier of the owner task of the period.
- The `state` member is set to the current state of the period.
- The `postponed_jobs_count` member is set to the count of jobs which are not released yet.
- If the current state of the period is `RATE_MONOTONIC_INACTIVE`, the `since_last_period` and `executed_since_last_period` members will be set to zero. Otherwise, both members will contain time information since the last successful invocation of the *rtems_rate_monotonic_period()* (page 317) directive by the owner task. More specifically, the `since_last_period` member will be set to the time elapsed since the last successful invocation. The `executed_since_last_period` member will be set to the processor time consumed by the owner task since the last successful invocation.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no rate monotonic period associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `status` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive will not cause the calling task to be preempted.

### 11.4.7 rtems_rate_monotonic_get_statistics()

Gets the statistics of the period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_get_statistics(
  rtems_id                                id,
  rtems_rate_monotonic_period_statistics *status
);
```

PARAMETERS:

**id**
This parameter is the rate monotonic period identifier.

**status**
This parameter is the pointer to an *rtems_rate_monotonic_period_statistics* (page 49) object. When the directive call is successful, the period statistics will be stored in this object.

DESCRIPTION:

This directive returns the statistics of the rate monotonic period specified by `id`. The statistics of the period will be returned in the members of the period statistics object referenced by `status`:

- The `count` member is set to the number of periods executed.
- The `missed_count` member is set to the number of periods missed.
- The `min_cpu_time` member is set to the least amount of processor time used in the period.
- The `max_cpu_time` member is set to the highest amount of processor time used in the period.
- The `total_cpu_time` member is set to the total amount of processor time used in the period.
- The `min_wall_time` member is set to the least amount of *CLOCK_MONOTONIC* time used in the period.
- The `max_wall_time` member is set to the highest amount of *CLOCK_MONOTONIC* time used in the period.
- The `total_wall_time` member is set to the total amount of *CLOCK_MONOTONIC* time used in the period.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no rate monotonic period associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `status` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive will not cause the calling task to be preempted.

### 11.4.8 rtems_rate_monotonic_reset_statistics()

Resets the statistics of the period.

CALLING SEQUENCE:

```
rtems_status_code rtems_rate_monotonic_reset_statistics( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the rate monotonic period identifier.

DESCRIPTION:

This directive resets the statistics of the rate monotonic period specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no rate monotonic period associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive will not cause the calling task to be preempted.

### 11.4.9 rtems_rate_monotonic_reset_all_statistics()

Resets the statistics of all periods.

CALLING SEQUENCE:

```
void rtems_rate_monotonic_reset_all_statistics( void );
```

DESCRIPTION:

This directive resets the statistics information associated with all rate monotonic period instances.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 11.4.10 rtems_rate_monotonic_report_statistics()

Reports the period statistics using the *printk()* (page 491) printer.

CALLING SEQUENCE:

```
void rtems_rate_monotonic_report_statistics( void );
```

DESCRIPTION:

This directive prints a report on all active periods which have executed at least one period using the *printk()* (page 491) printer.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 11.4.11 rtems_rate_monotonic_report_statistics_with_plugin()

Reports the period statistics using the printer plugin.

CALLING SEQUENCE:

```
void rtems_rate_monotonic_report_statistics_with_plugin(
  const struct rtems_printer *printer
);
```

PARAMETERS:

**printer**
This parameter is the printer plugin to output the report.

DESCRIPTION:

This directive prints a report on all active periods which have executed at least one period using the printer plugin specified by `printer`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

CHAPTER

**TWELVE**

# SEMAPHORE MANAGER

## 12.1 Introduction

The Semaphore Manager utilizes standard Dijkstra counting semaphores to provide synchronization and mutual exclusion capabilities. The directives provided by the Semaphore Manager are:

- *rtems_semaphore_create()* (page 335) - Creates a semaphore.
- *rtems_semaphore_ident()* (page 339) - Identifies a semaphore by the object name.
- *rtems_semaphore_delete()* (page 341) - Deletes the semaphore.
- *rtems_semaphore_obtain()* (page 343) - Obtains the semaphore.
- *rtems_semaphore_release()* (page 346) - Releases the semaphore.
- *rtems_semaphore_flush()* (page 348) - Flushes the semaphore.
- *rtems_semaphore_set_priority()* (page 350) - Sets the priority by scheduler for the semaphore.

## 12.2 Background

A semaphore can be viewed as a protected variable whose value can be modified only with the `rtems_semaphore_create`, `rtems_semaphore_obtain`, and `rtems_semaphore_release` directives. RTEMS supports both binary and counting semaphores. A binary semaphore is restricted to values of zero or one, while a counting semaphore can assume any non-negative integer value.

A binary semaphore (not a simple binary semaphore) can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code (mutex). In this instance, the semaphore would be created with an initial count of one to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task must issue the `rtems_semaphore_obtain` directive to prevent other tasks from entering the critical section. Upon exit from the critical section, the task that obtained the binary semaphore must issue the `rtems_semaphore_release` directive to allow another task to execute the critical section. A binary semaphore must be released by the task that obtained it.

A counting semaphore can be used to control access to a pool of two or more resources. For example, access to three printers could be administered by a semaphore created with an initial count of three. When a task requires access to one of the printers, it issues the `rtems_semaphore_obtain` directive to obtain access to a printer. If a printer is not currently available, the task can wait for a printer to become available or return immediately. When the task has completed printing, it should issue the `rtems_semaphore_release` directive to allow other tasks access to the printer.

Task synchronization may be achieved by creating a semaphore with an initial count of zero. One task waits for the arrival of another task by issuing a `rtems_semaphore_obtain` directive when it reaches a synchronization point. The other task performs a corresponding `rtems_semaphore_release` operation when it reaches its synchronization point, thus unblocking the pending task.

### 12.2.1 Nested Resource Access

Deadlock occurs when a task owning a binary semaphore attempts to acquire that same semaphore and blocks as result. Since the semaphore is allocated to a task, it cannot be deleted. Therefore, the task that currently holds the semaphore and is also blocked waiting for that semaphore will never execute again.

RTEMS addresses this problem by allowing the task holding the binary semaphore to obtain the same binary semaphore multiple times in a nested manner. Each `rtems_semaphore_obtain` must be accompanied with a `rtems_semaphore_release`. The semaphore will only be made available for acquisition by other tasks when the outermost `rtems_semaphore_obtain` is matched with a `rtems_semaphore_release`.

Simple binary semaphores do not allow nested access and so can be used for task synchronization.

### 12.2.2 Priority Inheritance

RTEMS supports *priority inheritance* (page 29) for local, binary semaphores that use the priority task wait queue blocking discipline. In SMP configurations, the *O(m) Independence-Preserving Protocol (OMIP)* (page 30) is used instead.

### 12.2.3 Priority Ceiling

RTEMS supports *priority ceiling* (page 29) for local, binary semaphores that use the priority task wait queue blocking discipline.

### 12.2.4 Multiprocessor Resource Sharing Protocol

RTEMS supports the *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30) for local, binary semaphores that use the priority task wait queue blocking discipline. In uniprocessor configurations, the *Immediate Ceiling Priority Protocol (ICPP)* (page 29) is used instead.

### 12.2.5 Building a Semaphore Attribute Set

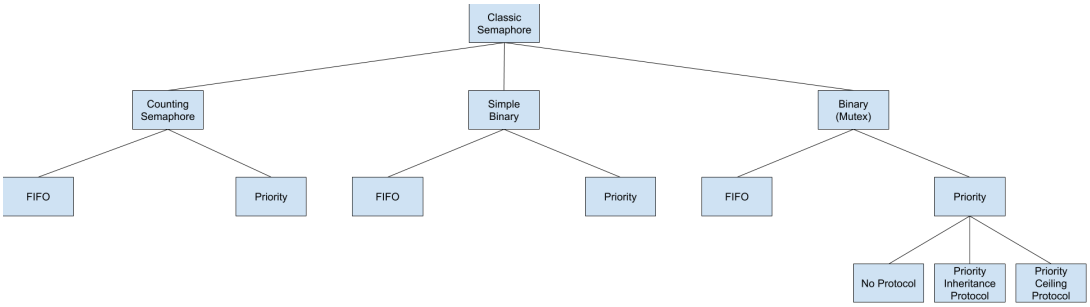
In general, an attribute set is built by a bitwise OR of the desired attribute components. The following table lists the set of valid semaphore attributes:

| | |
|---|---|
| RTEMS_FIFO | tasks wait by FIFO (default) |
| RTEMS_PRIORITY | tasks wait by priority |
| RTEMS_BINARY_SEMAPHORE | restrict values to 0 and 1 |
| RTEMS_COUNTING_SEMAPHORE | no restriction on values (default) |
| RTEMS_SIMPLE_BINARY_SEMAPHORE | restrict values to 0 and 1, do not allow nested access, allow deletion of locked semaphore. |
| RTEMS_NO_INHERIT_PRIORITY | do not use priority inheritance (default) |
| RTEMS_INHERIT_PRIORITY | use priority inheritance |
| RTEMS_NO_PRIORITY_CEILING | do not use priority ceiling (default) |
| RTEMS_PRIORITY_CEILING | use priority ceiling |
| RTEMS_NO_MULTIPROCESSOR_RESOURCE_SHARING | do not use Multiprocessor Resource Sharing Protocol (default) |
| RTEMS_MULTIPROCESSOR_RESOURCE_SHARING | use Multiprocessor Resource Sharing Protocol |
| RTEMS_LOCAL | local semaphore (default) |
| RTEMS_GLOBAL | global semaphore |

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute RTEMS_DEFAULT_ATTRIBUTES should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a local semaphore with the task priority waiting queue discipline. The attribute_set parameter passed to the rtems_semaphore_create directive could be either RTEMS_PRIORITY or RTEMS_LOCAL | RTEMS_PRIORITY. The attribute_set parameter can be set to RTEMS_PRIORITY because RTEMS_LOCAL is the default for all created tasks. If a similar semaphore were to be known globally, then the attribute_set parameter would be RTEMS_GLOBAL | RTEMS_PRIORITY.

Some combinatinos of these attributes are invalid. For example, priority ordered blocking discipline must be applied to a binary semaphore in order to use either the priority inheritance or priority ceiling functionality. The following tree figure illustrates the valid combinations.




### 12.2.6 Building a SEMAPHORE_OBTAIN Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_semaphore_obtain` directive are listed in the following table:

| | |
|---|---|
| `RTEMS_WAIT` | task will wait for semaphore (default) |
| `RTEMS_NO_WAIT` | task should not wait |

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An option listed as a default is not required to appear in the list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a semaphore. The option parameter passed to the `rtems_semaphore_obtain` directive should be `RTEMS_NO_WAIT`.

## 12.3 Operations

### 12.3.1 Creating a Semaphore

The `rtems_semaphore_create` directive creates a binary or counting semaphore with a user-specified name as well as an initial count. If a binary semaphore is created with a count of zero (0) to indicate that it has been allocated, then the task creating the semaphore is considered the current holder of the semaphore. At create time the method for ordering waiting tasks in the semaphore's task wait queue (by FIFO or task priority) is specified. Additionally, the priority inheritance or priority ceiling algorithm may be selected for local, binary semaphores that use the priority task wait queue blocking discipline. If the priority ceiling algorithm is selected, then the highest priority of any task which will attempt to obtain this semaphore must be specified. RTEMS allocates a Semaphore Control Block (SMCB) from the SMCB free list. This data structure is used by RTEMS to manage the newly created semaphore. Also, a unique semaphore ID is generated and returned to the calling task.

### 12.3.2 Obtaining Semaphore IDs

When a semaphore is created, RTEMS generates a unique semaphore ID and assigns it to the created semaphore until it is deleted. The semaphore ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_semaphore_create` directive, the semaphore ID is stored in a user provided location. Second, the semaphore ID may be obtained later using the `rtems_semaphore_ident` directive. The semaphore ID is used by other semaphore manager directives to access this semaphore.

### 12.3.3 Acquiring a Semaphore

The `rtems_semaphore_obtain` directive is used to acquire the specified semaphore. A simplified version of the `rtems_semaphore_obtain` directive can be described as follows:

> If the semaphore's count is greater than zero then decrement the semaphore's count
> else wait for release of semaphore then return SUCCESSFUL.

When the semaphore cannot be immediately acquired, one of the following situations applies:

- By default, the calling task will wait forever to acquire the semaphore.
- Specifying `RTEMS_NO_WAIT` forces an immediate return with an error status code.
- Specifying a timeout limits the interval the task will wait before returning with an error status code.

If the task waits to acquire the semaphore, then it is placed in the semaphore's task wait queue in either FIFO or task priority order. If the task blocked waiting for a binary semaphore using priority inheritance and the task's priority is greater than that of the task currently holding the semaphore, then the holding task will inherit the priority of the blocking task. All tasks waiting on a semaphore are returned an error code when the semaphore is deleted.

When a task successfully obtains a semaphore using priority ceiling and the priority ceiling for this semaphore is greater than that of the holder, then the holder's priority will be elevated.

### 12.3.4 Releasing a Semaphore

The `rtems_semaphore_release` directive is used to release the specified semaphore. A simplified version of the `rtems_semaphore_release` directive can be described as follows:

> If there are no tasks are waiting on this semaphore then increment the semaphore's count else assign semaphore to a waiting task and return SUCCESSFUL.

If this is the outermost release of a binary semaphore that uses priority inheritance or priority ceiling and the task does not currently hold any other binary semaphores, then the task performing the `rtems_semaphore_release` will have its priority restored to its normal value.

### 12.3.5 Deleting a Semaphore

The `rtems_semaphore_delete` directive removes a semaphore from the system and frees its control block. A semaphore can be deleted by any local task that knows the semaphore's ID. As a result of this directive, all tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. Any subsequent references to the semaphore's name and ID are invalid.

## 12.4 Directives

This section details the directives of the Semaphore Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 12.4.1 rtems_semaphore_create()

Creates a semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_create(
  rtems_name          name,
  uint32_t            count,
  rtems_attribute     attribute_set,
  rtems_task_priority priority_ceiling,
  rtems_id           *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the semaphore.

**count**
This parameter is the initial count of the semaphore. If the semaphore is a binary semaphore, then a count of 0 will make the calling task the owner of the binary semaphore and a count of 1 will create a binary semaphore without an owner.

**attribute_set**
This parameter is the attribute set of the semaphore.

**priority_ceiling**
This parameter is the priority ceiling if the semaphore is a binary semaphore with the priority ceiling or MrsP locking protocol as defined by the attribute set.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created semaphore will be stored in this object.

DESCRIPTION:

This directive creates a semaphore which resides on the local node. The semaphore has the user-defined object name specified in `name` and the initial count specified in `count`. The assigned object identifier is returned in `id`. This identifier is used to access the semaphore with other semaphore related directives.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant. The attribute set defines

- the scope of the semaphore: `RTEMS_LOCAL` (default) or `RTEMS_GLOBAL`,
- the task wait queue discipline used by the semaphore: `RTEMS_FIFO` (default) or `RTEMS_PRIORITY`,
- the class of the semaphore: `RTEMS_COUNTING_SEMAPHORE` (default), `RTEMS_BINARY_SEMAPHORE`, or `RTEMS_SIMPLE_BINARY_SEMAPHORE`, and

- the locking protocol of a binary semaphore: no locking protocol (default), RTEMS_INHERIT_PRIORITY, RTEMS_PRIORITY_CEILING, or RTEMS_MULTIPROCESSOR_RESOURCE_SHARING.

The semaphore has a local or global **scope** in a multiprocessing network (this attribute does not refer to SMP systems). The scope is selected by the mutually exclusive RTEMS_LOCAL and RTEMS_GLOBAL attributes.

- A **local scope** is the default and can be emphasized through the use of the RTEMS_LOCAL attribute. A local semaphore can be only used by the node which created it.
- A **global scope** is established if the RTEMS_GLOBAL attribute is set. Setting the global attribute in a single node system has no effect.

The **task wait queue discipline** is selected by the mutually exclusive RTEMS_FIFO and RTEMS_PRIORITY attributes.

- The **FIFO discipline** is the default and can be emphasized through use of the RTEMS_FIFO attribute.
- The **priority discipline** is selected by the RTEMS_PRIORITY attribute. The locking protocols require the priority discipline.

The **semaphore class** is selected by the mutually exclusive RTEMS_COUNTING_SEMAPHORE, RTEMS_BINARY_SEMAPHORE, and RTEMS_SIMPLE_BINARY_SEMAPHORE attributes.

- The **counting semaphore class** is the default and can be emphasized through use of the RTEMS_COUNTING_SEMAPHORE attribute.
- The **binary semaphore class** is selected by the RTEMS_BINARY_SEMAPHORE attribute. Binary semaphores are mutual exclusion (mutex) synchronization primitives which may have an owner. The count of a binary semaphore is restricted to 0 and 1 values.
- The **simple binary semaphore class** is selected by the RTEMS_SIMPLE_BINARY_SEMAPHORE attribute. Simple binary semaphores have no owner. They may be used for task and interrupt synchronization. The count of a simple binary semaphore is restricted to 0 and 1 values.

Binary semaphores may use a **locking protocol**. If a locking protocol is selected, then the scope shall be local and the priority task wait queue discipline shall be selected. The locking protocol is selected by the mutually exclusive RTEMS_INHERIT_PRIORITY, RTEMS_PRIORITY_CEILING, and RTEMS_MULTIPROCESSOR_RESOURCE_SHARING attributes.

- The default is **no locking protocol**. This can be emphasized through use of the RTEMS_NO_INHERIT_PRIORITY, RTEMS_NO_MULTIPROCESSOR_RESOURCE_SHARING, and RTEMS_NO_PRIORITY_CEILING attributes.
- The **priority inheritance locking protocol** is selected by the RTEMS_INHERIT_PRIORITY attribute.
- The **priority ceiling locking protocol** is selected by the RTEMS_PRIORITY_CEILING attribute. For this locking protocol a priority ceiling shall be specified in priority_ceiling.
- The **MrsP locking protocol** is selected by the RTEMS_MULTIPROCESSOR_RESOURCE_SHARING attribute in SMP configurations, otherwise this attribute selects the **priority ceiling locking protocol**. For these locking protocols a priority ceiling shall be specified in priority_ceiling. This priority is used to set the priority ceiling for all schedulers. This can be changed later with the *rtems_semaphore_set_priority()* (page 350) directive using the returned object identifier.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NUMBER**
The `count` parameter was invalid.

**RTEMS_NOT_DEFINED**
The `attribute_set` parameter was invalid.

**RTEMS_TOO_MANY**
There was no inactive object available to create a semaphore. The number of semaphores available to the application is configured through the *CONFIGURE_MAXIMUM_SEMAPHORES* (page 620) application configuration option.

**RTEMS_TOO_MANY**
In multiprocessing configurations, there was no inactive global object available to create a global semaphore. The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

**RTEMS_INVALID_PRIORITY**
The `priority_ceiling` parameter was invalid.

NOTES:

For control and maintenance of the semaphore, RTEMS allocates a *SMCB* from the local SMCB free pool and initializes it.

The SMCB for a global semaphore is allocated on the local node. Semaphores should not be made global unless remote tasks must interact with the semaphore. This is to avoid the system overhead incurred by the creation of a global semaphore. When a global semaphore is created, the semaphore's name and identifier must be transmitted to every node in the system for insertion in the local copy of the global object table.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of semaphores available to the application is configured through the *CONFIGURE_MAXIMUM_SEMAPHORES* (page 620) application configuration option.

- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 12.4.2 rtems_semaphore_ident()

Identifies a semaphore by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_ident(
  rtems_name name,
  uint32_t   node,
  rtems_id  *id
);
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**node**
This parameter is the node or node set to search for a matching object.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a semaphore identifier associated with the semaphore name specified in `name`.

The node to search is specified in `node`. It shall be

- a valid node number,
- the constant `RTEMS_SEARCH_ALL_NODES` to search in all nodes,
- the constant `RTEMS_SEARCH_LOCAL_NODE` to search in the local node only, or
- the constant `RTEMS_SEARCH_OTHER_NODES` to search in all nodes except the local node.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the specified nodes.

**RTEMS_INVALID_NODE**
In multiprocessing configurations, the specified node was invalid.

NOTES:

If the semaphore name is not unique, then the semaphore identifier will match the first semaphore with that name in the search order. However, this semaphore identifier is not guaranteed to correspond to the desired semaphore.

The objects are searched from lowest to the highest index. If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched from lowest to the highest node number.

If node is a valid node number which does not represent the local node, then only the semaphores exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

The semaphore identifier is used with other semaphore related directives to access the semaphore.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 12.4.3 rtems_semaphore_delete()

Deletes the semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the semaphore identifier.

DESCRIPTION:

This directive deletes the semaphore specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no semaphore associated with the identifier specified by `id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The semaphore resided on a remote node.

**RTEMS_RESOURCE_IN_USE**
The binary semaphore had an owner.

NOTES:

Binary semaphores with an owner cannot be deleted.

When a semaphore is deleted, all tasks blocked waiting to obtain the semaphore will be readied and returned a status code which indicates that the semaphore was deleted.

The *SMCB* for the deleted semaphore is reclaimed by RTEMS.

When a global semaphore is deleted, the semaphore identifier must be transmitted to every node in the system for deletion from the local copy of the global object table.

The semaphore must reside on the local node, even if the semaphore was created with the `RTEMS_GLOBAL` attribute.

Proxies, used to represent remote tasks, are reclaimed when the semaphore is deleted.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 12.4.4 rtems_semaphore_obtain()

Obtains the semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_obtain(
  rtems_id       id,
  rtems_option   option_set,
  rtems_interval timeout
);
```

PARAMETERS:

**id**
This parameter is the semaphore identifier.

**option_set**
This parameter is the option set.

**timeout**
This parameter is the timeout in *clock ticks* if the RTEMS_WAIT option is set. Use RTEMS_NO_TIMEOUT to wait potentially forever.

DESCRIPTION:

This directive obtains the semaphore specified by id.

The **option set** specified in option_set is built through a *bitwise or* of the option constants described below. Not all combinations of options are allowed. Some options are mutually exclusive. If mutually exclusive options are combined, the behaviour is undefined. Options not mentioned below are not evaluated by this directive and have no effect. Default options can be selected by using the RTEMS_DEFAULT_OPTIONS constant.

The calling task can **wait** or **try to obtain** the semaphore according to the mutually exclusive RTEMS_WAIT and RTEMS_NO_WAIT options.

- **Waiting to obtain** the semaphore is the default and can be emphasized through the use of the RTEMS_WAIT option. The timeout parameter defines how long the calling task is willing to wait. Use RTEMS_NO_TIMEOUT to wait potentially forever, otherwise set a timeout interval in clock ticks.
- **Trying to obtain** the semaphore is selected by the RTEMS_NO_WAIT option. If this option is defined, then the timeout parameter is ignored. When the semaphore cannot be immediately obtained, then the RTEMS_UNSATISFIED status is returned.

With either RTEMS_WAIT or RTEMS_NO_WAIT if the current semaphore count is positive, then it is decremented by one and the semaphore is successfully obtained by returning immediately with the RTEMS_SUCCESSFUL status code.

If the calling task chooses to return immediately and the current semaphore count is zero, then the RTEMS_UNSATISFIED status code is returned indicating that the semaphore is not available.

If the calling task chooses to wait for a semaphore and the current semaphore count is zero, then the calling task is placed on the semaphore's wait queue and blocked. If a local, binary semaphore was created with the RTEMS_INHERIT_PRIORITY attribute, then the priority of the task currently holding the binary semaphore will inherit the current priority set of the blocking

task. The priority inheritance is carried out recursively. This means, that if the task currently holding the binary semaphore is blocked on another local, binary semaphore using the priority inheritance locking protocol, then the owner of this semaphore will inherit the current priority sets of both tasks, and so on. A task has a current priority for each scheduler.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no semaphore associated with the identifier specified by `id`.

**RTEMS_NOT_DEFINED**
The semaphore uses a priority ceiling and there was no priority ceiling defined for the *home scheduler* of the calling task.

**RTEMS_UNSATISFIED**
The semaphore could not be obtained immediately.

**RTEMS_INVALID_PRIORITY**
The semaphore uses a priority ceiling and the calling task had a current priority less than the priority ceiling.

**RTEMS_INCORRECT_STATE**
Acquiring of the local, binary semaphore by the calling task would have cased a deadlock.

**RTEMS_INCORRECT_STATE**
The calling task attempted to recursively obtain a local, binary semaphore using the MrsP locking protocol.

**RTEMS_UNSATISFIED**
The semaphore was flushed while the calling task was waiting to obtain the semaphore.

**RTEMS_TIMEOUT**
The timeout happened while the calling task was waiting to obtain the semaphore.

**RTEMS_OBJECT_WAS_DELETED**
The semaphore was deleted while the calling task was waiting to obtain the semaphore.

NOTES:

If a local, binary semaphore was created with the `RTEMS_PRIORITY_CEILING` or `RTEMS_MULTIPROCESSOR_RESOURCE_SHARING` attribute, a task successfully obtains the semaphore, and the priority of that task is greater than the ceiling priority for this semaphore, then the priority of the task acquiring the semaphore is elevated to that of the ceiling.

Deadlock situations are detected for local, binary semaphores. If a deadlock is detected, then the directive immediately returns the `RTEMS_INCORRECT_STATE` status code.

It is not allowed to recursively obtain (nested access) a local, binary semaphore using the MrsP locking protocol and any attempt to do this will just return the `RTEMS_INCORRECT_STATE` status code. This error can only happen in SMP configurations.

If the semaphore was created with the `RTEMS_PRIORITY` attribute, then the calling task is inserted into the wait queue according to its priority. However, if the semaphore was created with the `RTEMS_FIFO` attribute, then the calling task is placed at the rear of the wait queue.

Attempting to obtain a global semaphore which does not reside on the local node will generate a request to the remote node to access the semaphore. If the semaphore is not available and `RTEMS_NO_WAIT` was not specified, then the task must be blocked until the semaphore is released. A proxy is allocated on the remote node to represent the task until the semaphore is released.

CONSTRAINTS:

The following constraints apply to this directive:

- When a local, counting semaphore or a local, simple binary semaphore is accessed and the `RTEMS_NO_WAIT` option is set, the directive may be called from within interrupt context.
- When a local semaphore is accessed and the request can be immediately satisfied, the directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- When the request cannot be immediately satisfied and the `RTEMS_WAIT` option is set, the calling task blocks at some point during the directive call.
- The timeout functionality of the directive requires a *clock tick*.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 12.4.5 rtems_semaphore_release()

Releases the semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_release( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the semaphore identifier.

DESCRIPTION:

This directive releases the semaphore specified by `id`. If the semaphore's wait queue is not empty, then

- the first task on the wait queue is removed and unblocked, the semaphore's count is not changed, otherwise
- the semaphore's count is incremented by one for counting semaphores and set to one for binary and simple binary semaphores.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no semaphore associated with the identifier specified by `id`.

**RTEMS_NOT_OWNER_OF_RESOURCE**
The calling task was not the owner of the semaphore.

**RTEMS_UNSATISFIED**
The semaphore's count already had the maximum value of UINT32_MAX.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

The outermost release of a local, binary semaphore using the priority inheritance, priority ceiling, or MrsP locking protocol may result in the calling task having its priority lowered. This will occur if the highest priority of the calling task was available due to the ownership of the released semaphore. If a task was on the semaphore's wait queue, then the priority associated with the semaphore will be transferred to the new owner.

Releasing a global semaphore which does not reside on the local node will generate a request telling the remote node to release the semaphore.

If the task to be unblocked resides on a different node from the semaphore, then the semaphore allocation is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

CONSTRAINTS:

The following constraints apply to this directive:

- When a local, counting semaphore or a local, simple binary semaphore is accessed, the directive may be called from within interrupt context.
- When a local semaphore is accessed, the directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 12.4.6 rtems_semaphore_flush()

Flushes the semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_flush( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the semaphore identifier.

DESCRIPTION:

This directive unblocks all tasks waiting on the semaphore specified by `id`. The semaphore's count is not changed by this directive. Tasks which are unblocked as the result of this directive will return from the *rtems_semaphore_obtain()* (page 343) directive with a status code of `RTEMS_UNSATISFIED` to indicate that the semaphore was not obtained.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no semaphore associated with the identifier specified by `id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The semaphore resided on a remote node.

**RTEMS_NOT_DEFINED**
Flushing a semaphore using the MrsP locking protocol is undefined behaviour.

NOTES:

If the task to be unblocked resides on a different node from the semaphore, then the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

It is not allowed to flush a local, binary semaphore using the MrsP locking protocol and any attempt to do this will just return the `RTEMS_NOT_DEFINED` status code. This error can only happen in SMP configurations.

For barrier synchronization, the *Barrier Manager* (page 355) offers a cleaner alternative to using the semaphore flush directive. Unlike POSIX barriers, they have a manual release option.

Using the semaphore flush directive for condition synchronization in concert with another semaphore may be subject to the lost wake-up problem. The following attempt to implement a condition variable is broken.

```
#include <rtems.h>
#include <assert.h>

void cnd_wait( rtems_id cnd, rtems_id mtx )
{
```

(continues on next page)

(continued from previous page)

```
  rtems_status_code sc;

  sc = rtems_semaphore_release( mtx );
  assert( sc == RTEMS_SUCCESSFUL );

  // Here, a higher priority task may run and satisfy the condition.
  // We may never wake up from the next semaphore obtain.

  sc = rtems_semaphore_obtain( cnd, RTEMS_WAIT, RTEMS_NO_TIMEOUT );
  assert( sc == RTEMS_UNSATISFIED );

  sc = rtems_semaphore_obtain( mtx, RTEMS_WAIT, RTEMS_NO_TIMEOUT );
  assert( sc == RTEMS_SUCCESSFUL );
}

void cnd_broadcast( rtems_id cnd )
{
  rtems_status_code sc;

  sc = rtems_semaphore_flush( cnd );
  assert( sc == RTEMS_SUCCESSFUL );
}
```

CONSTRAINTS:

The following constraints apply to this directive:

- When a local, counting semaphore or a local, simple binary semaphore is accessed, the directive may be called from within interrupt context.
- When a local semaphore is accessed, the directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 12.4.7 rtems_semaphore_set_priority()

Sets the priority by scheduler for the semaphore.

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_set_priority(
  rtems_id             semaphore_id,
  rtems_id             scheduler_id,
  rtems_task_priority  new_priority,
  rtems_task_priority *old_priority
);
```

PARAMETERS:

**semaphore_id**
This parameter is the semaphore identifier.

**scheduler_id**
This parameter is the identifier of the scheduler corresponding to the new priority.

**new_priority**
This parameter is the new priority corresponding to the specified scheduler.

**old_priority**
This parameter is the pointer to an *rtems_task_priority* (page 57) object. When the directive call is successful, the old priority of the semaphore corresponding to the specified scheduler will be stored in this object.

DESCRIPTION:

This directive sets the priority of the semaphore specified by `semaphore_id`. The priority corresponds to the scheduler specified by `scheduler_id`.

The special priority value `RTEMS_CURRENT_PRIORITY` can be used to get the current priority without changing it.

The availability and use of a priority depends on the class and locking protocol of the semaphore:

- For local, binary semaphores using the MrsP locking protocol, the ceiling priority for each scheduler can be set by this directive.
- For local, binary semaphores using the priority ceiling protocol, the ceiling priority can be set by this directive.
- For other semaphore classes and locking protocols, setting a priority is undefined behaviour.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `old_priority` parameter was NULL.

**RTEMS_INVALID_ID**
There was no scheduler associated with the identifier specified by `scheduler_id`.

**RTEMS_INVALID_ID**
There was no semaphore associated with the identifier specified by `semaphore_id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The semaphore resided on a remote node.

**RTEMS_INVALID_PRIORITY**
The `new_priority` parameter was invalid.

**RTEMS_NOT_DEFINED**
Setting a priority for the class or locking protocol of the semaphore is undefined behaviour.

NOTES:

Please have a look at the following example:

```
#include <assert.h>
#include <rtems.h>

#define SCHED_A rtems_build_name( ' ', ' ', ' ', 'A' )
#define SCHED_B rtems_build_name( ' ', ' ', ' ', 'B' )

static void Init( rtems_task_argument arg )
{
  rtems_status_code   sc;
  rtems_id            semaphore_id;
  rtems_id            scheduler_a_id;
  rtems_id            scheduler_b_id;
  rtems_task_priority prio;

  (void) arg;

  // Get the scheduler identifiers
  sc = rtems_scheduler_ident( SCHED_A, &scheduler_a_id );
  assert( sc == RTEMS_SUCCESSFUL );
  sc = rtems_scheduler_ident( SCHED_B, &scheduler_b_id );
  assert( sc == RTEMS_SUCCESSFUL );

  // Create a local, binary semaphore using the MrsP locking protocol
  sc = rtems_semaphore_create(
    rtems_build_name( 'M', 'R', 'S', 'P' ),
    1,
    RTEMS_BINARY_SEMAPHORE | RTEMS_PRIORITY |
      RTEMS_MULTIPROCESSOR_RESOURCE_SHARING,
    1,
    &semaphore_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  // The ceiling priority for each scheduler is equal to the priority
```

(continues on next page)

(continued from previous page)

```
  // specified for the semaphore creation.
  prio = RTEMS_CURRENT_PRIORITY;
  sc = rtems_semaphore_set_priority( semaphore_id, scheduler_a_id, prio, &prio );
  assert( sc == RTEMS_SUCCESSFUL );
  assert( prio == 1 );

  // Check the old value and set a new ceiling priority for scheduler B
  prio = 2;
  sc = rtems_semaphore_set_priority( semaphore_id, scheduler_b_id, prio, &prio );
  assert( sc == RTEMS_SUCCESSFUL );
  assert( prio == 1 );

  // Check the ceiling priority values
  prio = RTEMS_CURRENT_PRIORITY;
  sc = rtems_semaphore_set_priority( semaphore_id, scheduler_a_id, prio, &prio );
  assert( sc == RTEMS_SUCCESSFUL );
  assert( prio == 1 );
  prio = RTEMS_CURRENT_PRIORITY;
  sc = rtems_semaphore_set_priority( semaphore_id, scheduler_b_id, prio, &prio );
  assert( sc == RTEMS_SUCCESSFUL );
  assert( prio == 2 );

  sc = rtems_semaphore_delete( semaphore_id );
  assert( sc == RTEMS_SUCCESSFUL );

  rtems_shutdown_executive( 0 );
}

#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_MAXIMUM_SEMAPHORES 1
#define CONFIGURE_MAXIMUM_PROCESSORS 2

#define CONFIGURE_SCHEDULER_SIMPLE_SMP

#include <rtems/scheduler.h>

RTEMS_SCHEDULER_CONTEXT_SIMPLE_SMP( a );
RTEMS_SCHEDULER_CONTEXT_SIMPLE_SMP( b );

#define CONFIGURE_SCHEDULER_TABLE_ENTRIES \
    RTEMS_SCHEDULER_TABLE_SIMPLE_SMP( a, SCHED_A ), \
    RTEMS_SCHEDULER_TABLE_SIMPLE_SMP( b, SCHED_B )

#define CONFIGURE_SCHEDULER_ASSIGNMENTS \
    RTEMS_SCHEDULER_ASSIGN( 0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY ), \
    RTEMS_SCHEDULER_ASSIGN( 1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY )

```

(continues on next page)

(continued from previous page)

```
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may change the priority of a task. This may cause the calling task to be preempted.

CHAPTER

**THIRTEEN**

# BARRIER MANAGER

## 13.1 Introduction

The Barrier Manager provides a unique synchronization capability which can be used to have a set of tasks block and be unblocked as a set. The directives provided by the Barrier Manager are:

- *rtems_barrier_create()* (page 359) - Creates a barrier.
- *rtems_barrier_ident()* (page 361) - Identifies a barrier by the object name.
- *rtems_barrier_delete()* (page 362) - Deletes the barrier.
- *rtems_barrier_wait()* (page 363) - Waits at the barrier.
- *rtems_barrier_release()* (page 364) - Releases the barrier.

## 13.2 Background

A barrier can be viewed as a gate at which tasks wait until the gate is opened. This has many analogies in the real world. Horses and other farm animals may approach a closed gate and gather in front of it, waiting for someone to open the gate so they may proceed. Similarly, ticket holders gather at the gates of arenas before concerts or sporting events waiting for the arena personnel to open the gates so they may enter.

Barriers are useful during application initialization. Each application task can perform its local initialization before waiting for the application as a whole to be initialized. Once all tasks have completed their independent initializations, the "application ready" barrier can be released.

### 13.2.1 Automatic Versus Manual Barriers

Just as with a real-world gate, barriers may be configured to be manually opened or automatically opened. All tasks calling the `rtems_barrier_wait` directive will block until a controlling task invokes the `rtems_barrier_release` directive.

Automatic barriers are created with a limit to the number of tasks which may simultaneously block at the barrier. Once this limit is reached, all of the tasks are released. For example, if the automatic limit is ten tasks, then the first nine tasks calling the `rtems_barrier_wait` directive will block. When the tenth task calls the `rtems_barrier_wait` directive, the nine blocked tasks will be released and the tenth task returns to the caller without blocking.

### 13.2.2 Building a Barrier Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The following table lists the set of valid barrier attributes:

**RTEMS_BARRIER_AUTOMATIC_RELEASE**
automatically release the barrier when the configured number of tasks are blocked

**RTEMS_BARRIER_MANUAL_RELEASE**
only release the barrier when the application invokes the `rtems_barrier_release` directive. (default)

> **Note**
>
> Barriers only support FIFO blocking order because all waiting tasks are released as a set. Thus the released tasks will all become ready to execute at the same time and compete for the processor based upon their priority.

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a barrier with the automatic release policy. The `attribute_set` parameter passed to the `rtems_barrier_create` directive will be `RTEMS_BARRIER_AUTOMATIC_RELEASE`. In this case, the user must also specify the `maximum_waiters` parameter.

## 13.3 Directives

This section details the directives of the Barrier Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 13.3.1 rtems_barrier_create()

Creates a barrier.

CALLING SEQUENCE:

```
rtems_status_code rtems_barrier_create(
  rtems_name      name,
  rtems_attribute attribute_set,
  uint32_t        maximum_waiters,
  rtems_id       *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the barrier.

**attribute_set**
This parameter is the attribute set of the barrier.

**maximum_waiters**
This parameter is the maximum count of waiters on an automatic release barrier.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created barrier will be stored in this object.

DESCRIPTION:

This directive creates a barrier which resides on the local node. The barrier has the user-defined object name specified in `name` and the initial count specified in `attribute_set`. The assigned object identifier is returned in `id`. This identifier is used to access the barrier with other barrier related directives.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant.

The **barrier class** is selected by the mutually exclusive `RTEMS_BARRIER_MANUAL_RELEASE` and `RTEMS_BARRIER_AUTOMATIC_RELEASE` attributes.

- The **manual release class** is the default and can be emphasized through use of the `RTEMS_BARRIER_MANUAL_RELEASE` attribute. For this class, there is no limit on the number of tasks that will block at the barrier. Only when the *rtems_barrier_release()* (page 364) directive is invoked, are the tasks waiting at the barrier unblocked.
- The **automatic release class** is selected by the `RTEMS_BARRIER_AUTOMATIC_RELEASE` attribute. For this class, tasks calling the *rtems_barrier_wait()* (page 363) directive will block until there are `maximum_waiters` minus one tasks waiting at the barrier. When the `maximum_waiters` task invokes the *rtems_barrier_wait()* (page 363) directive, the previous `maximum_waiters` - 1 tasks are automatically released and the caller returns.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NUMBER**
The `maximum_waiters` parameter was 0 for an automatic release barrier.

**RTEMS_TOO_MANY**
There was no inactive object available to create a barrier. The number of barriers available to the application is configured through the *CONFIGURE_MAXIMUM_BARRIERS* (page 614) application configuration option.

NOTES:

For control and maintenance of the barrier, RTEMS allocates a *BCB* from the local BCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of barriers available to the application is configured through the *CONFIGURE_MAXIMUM_BARRIERS* (page 614) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 13.3.2 rtems_barrier_ident()

Identifies a barrier by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_barrier_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a barrier identifier associated with the barrier name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the barrier name is not unique, then the barrier identifier will match the first barrier with that name in the search order. However, this barrier identifier is not guaranteed to correspond to the desired barrier.

The objects are searched from lowest to the highest index. Only the local node is searched.

The barrier identifier is used with other barrier related directives to access the barrier.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 13.3.3 rtems_barrier_delete()

Deletes the barrier.

CALLING SEQUENCE:

```
rtems_status_code rtems_barrier_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the barrier identifier.

DESCRIPTION:

This directive deletes the barrier specified by `id`. All tasks blocked waiting for the barrier to be released will be readied and returned a status code which indicates that the barrier was deleted.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no barrier associated with the identifier specified by `id`.

NOTES:

The *BCB* for the deleted barrier is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 13.3.4 rtems_barrier_wait()

Waits at the barrier.

CALLING SEQUENCE:

```
rtems_status_code rtems_barrier_wait( rtems_id id, rtems_interval timeout );
```

PARAMETERS:

**id**
This parameter is the barrier identifier.

**timeout**
This parameter is the timeout in clock ticks. Use `RTEMS_NO_TIMEOUT` to wait potentially forever.

DESCRIPTION:

This directive waits at the barrier specified by `id`. The `timeout` parameter defines how long the calling task is willing to wait. Use `RTEMS_NO_TIMEOUT` to wait potentially forever, otherwise set a timeout interval in clock ticks.

Conceptually, the calling task should always be thought of as blocking when it makes this call and being unblocked when the barrier is released. If the barrier is configured for manual release, this rule of thumb will always be valid. If the barrier is configured for automatic release, all callers will block except for the one which trips the automatic release condition.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no barrier associated with the identifier specified by `id`.

**RTEMS_TIMEOUT**
The timeout happened while the calling task was waiting at the barrier.

**RTEMS_OBJECT_WAS_DELETED**
The barrier was deleted while the calling task was waiting at the barrier.

NOTES:

For automatic release barriers, the maximum count of waiting tasks is defined during barrier creation, see *rtems_barrier_create()* (page 359).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The timeout functionality of the directive requires a *clock tick*.

### 13.3.5 rtems_barrier_release()

Releases the barrier.

CALLING SEQUENCE:

```
rtems_status_code rtems_barrier_release( rtems_id id, uint32_t *released );
```

PARAMETERS:

**id**
This parameter is the barrier identifier.

**released**
This parameter is the pointer to an uint32_t object. When the directive call is successful, the number of released tasks will be stored in this object.

DESCRIPTION:

This directive releases the barrier specified by `id`. All tasks waiting at the barrier will be unblocked. The number of released tasks will be returned in `released`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `released` parameter was NULL.

**RTEMS_INVALID_ID**
There was no barrier associated with the identifier specified by `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.

CHAPTER

**FOURTEEN**

# MESSAGE MANAGER

## 14.1 Introduction

The Message Manager provides communication and synchronization capabilities using RTEMS message queues. The directives provided by the Message Manager are:

- *rtems_message_queue_create()* (page 372) - Creates a message queue.
- *rtems_message_queue_construct()* (page 375) - Constructs a message queue from the specified the message queue configuration.
- *rtems_message_queue_ident()* (page 377) - Identifies a message queue by the object name.
- *rtems_message_queue_delete()* (page 379) - Deletes the message queue.
- *rtems_message_queue_send()* (page 381) - Puts the message at the rear of the queue.
- *rtems_message_queue_urgent()* (page 383) - Puts the message at the front of the queue.
- *rtems_message_queue_broadcast()* (page 385) - Broadcasts the messages to the tasks waiting at the queue.
- *rtems_message_queue_receive()* (page 387) - Receives a message from the queue.
- *rtems_message_queue_get_number_pending()* (page 389) - Gets the number of messages pending on the queue.
- *rtems_message_queue_flush()* (page 390) - Flushes all messages on the queue.
- *RTEMS_MESSAGE_QUEUE_BUFFER()* (page 391) - Defines a structure which can be used as a message queue buffer for messages of the specified maximum size.

## 14.2 Background

### 14.2.1 Messages

A message is a variable length buffer where information can be stored to support communication. The length of the message and the information stored in that message are user-defined and can be actual data, pointer(s), or empty.

### 14.2.2 Message Queues

A message queue permits the passing of messages among tasks and ISRs. Message queues can contain a variable number of messages. Normally messages are sent to and received from the queue in FIFO order using the `rtems_message_queue_send` directive. However, the `rtems_message_queue_urgent` directive can be used to place messages at the head of a queue in LIFO order.

Synchronization can be accomplished when a task can wait for a message to arrive at a queue. Also, a task may poll a queue for the arrival of a message.

The maximum length message which can be sent is set on a per message queue basis. The message content must be copied in general to/from an internal buffer of the message queue or directly to a peer in certain cases. This copy operation is performed with interrupts disabled. So it is advisable to keep the messages as short as possible.

### 14.2.3 Building a Message Queue Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid message queue attributes is provided in the following table:

| | |
|---|---|
| `RTEMS_FIFO` | tasks wait by FIFO (default) |
| `RTEMS_PRIORITY` | tasks wait by priority |
| `RTEMS_LOCAL` | local message queue (default) |
| `RTEMS_GLOBAL` | global message queue |

An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a local message queue with the task priority waiting queue discipline. The attribute_set parameter to the `rtems_message_queue_create` directive could be either `RTEMS_PRIORITY` or `RTEMS_LOCAL | RTEMS_PRIORITY`. The attribute_set parameter can be set to `RTEMS_PRIORITY` because `RTEMS_LOCAL` is the default for all created message queues. If a similar message queue were to be known globally, then the attribute_set parameter would be `RTEMS_GLOBAL | RTEMS_PRIORITY`.

### 14.2.4 Building a MESSAGE_QUEUE_RECEIVE Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_message_queue_receive` directive are listed in the following table:

| | |
|---|---|
| `RTEMS_WAIT` | task will wait for a message (default) |
| `RTEMS_NO_WAIT` | task should not wait |

An option listed as a default is not required to appear in the option OR list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a message to arrive. The option parameter passed to the `rtems_message_queue_receive` directive should be `RTEMS_NO_WAIT`.

## 14.3 Operations

### 14.3.1 Creating a Message Queue

The `rtems_message_queue_create` directive creates a message queue with the user-defined name. The user specifies the maximum message size and maximum number of messages which can be placed in the message queue at one time. The user may select FIFO or task priority as the method for placing waiting tasks in the task wait queue. RTEMS allocates a Queue Control Block (QCB) from the QCB free list to maintain the newly created queue as well as memory for the message buffer pool associated with this message queue. RTEMS also generates a message queue ID which is returned to the calling task.

For GLOBAL message queues, the maximum message size is effectively limited to the longest message which the MPCI is capable of transmitting.

### 14.3.2 Obtaining Message Queue IDs

When a message queue is created, RTEMS generates a unique message queue ID. The message queue ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_message_queue_create` directive, the queue ID is stored in a user provided location. Second, the queue ID may be obtained later using the `rtems_message_queue_ident` directive. The queue ID is used by other message manager directives to access this message queue.

### 14.3.3 Receiving a Message

The `rtems_message_queue_receive` directive attempts to retrieve a message from the specified message queue. If at least one message is in the queue, then the message is removed from the queue, copied to the caller's message buffer, and returned immediately along with the length of the message. When messages are unavailable, one of the following situations applies:

- By default, the calling task will wait forever for the message to arrive.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the period the task will wait before returning with an error status.

If the task waits for a message, then it is placed in the message queue's task wait queue in either FIFO or task priority order. All tasks waiting on a message queue are returned an error code when the message queue is deleted.

### 14.3.4 Sending a Message

Messages can be sent to a queue with the `rtems_message_queue_send` and `rtems_message_queue_urgent` directives. These directives work identically when tasks are waiting to receive a message. A task is removed from the task waiting queue, unblocked, and the message is copied to a waiting task's message buffer.

When no tasks are waiting at the queue, `rtems_message_queue_send` places the message at the rear of the message queue, while `rtems_message_queue_urgent` places the message at the front of the queue. The message is copied to a message buffer from this message queue's buffer pool and then placed in the message queue. Neither directive can successfully send a message to a message queue which has a full queue of pending messages.

### 14.3.5 Broadcasting a Message

The `rtems_message_queue_broadcast` directive sends the same message to every task waiting on the specified message queue as an atomic operation. The message is copied to each waiting task's message buffer and each task is unblocked. The number of tasks which were unblocked is returned to the caller.

### 14.3.6 Deleting a Message Queue

The `rtems_message_queue_delete` directive removes a message queue from the system and frees its control block as well as the memory associated with this message queue's message buffer pool. A message queue can be deleted by any local task that knows the message queue's ID. As a result of this directive, all tasks blocked waiting to receive a message from the message queue will be readied and returned a status code which indicates that the message queue was deleted. Any subsequent references to the message queue's name and ID are invalid. Any messages waiting at the message queue are also deleted and deallocated.

## 14.4 Directives

This section details the directives of the Message Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 14.4.1 rtems_message_queue_create()

Creates a message queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_create(
  rtems_name      name,
  uint32_t        count,
  size_t          max_message_size,
  rtems_attribute attribute_set,
  rtems_id       *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the message queue.

**count**
This parameter is the maximum count of pending messages supported by the message queue.

**max_message_size**
This parameter is the maximum size in bytes of a message supported by the message queue.

**attribute_set**
This parameter is the attribute set of the message queue.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created message queue will be stored in this object.

DESCRIPTION:

This directive creates a message queue which resides on the local node. The message queue has the user-defined object name specified in `name`. Memory is allocated from the RTEMS Workspace for the count of messages specified in `count`, each of `max_message_size` bytes in length. The assigned object identifier is returned in `id`. This identifier is used to access the message queue with other message queue related directives.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant. The attribute set defines

- the scope of the message queue: `RTEMS_LOCAL` (default) or `RTEMS_GLOBAL` and
- the task wait queue discipline used by the message queue: `RTEMS_FIFO` (default) or `RTEMS_PRIORITY`.

The message queue has a local or global **scope** in a multiprocessing network (this attribute does not refer to SMP systems). The scope is selected by the mutually exclusive `RTEMS_LOCAL` and `RTEMS_GLOBAL` attributes.

- A **local scope** is the default and can be emphasized through the use of the `RTEMS_LOCAL` attribute. A local message queue can be only used by the node which created it.
- A **global scope** is established if the `RTEMS_GLOBAL` attribute is set. Setting the global attribute in a single node system has no effect.

The **task wait queue discipline** is selected by the mutually exclusive `RTEMS_FIFO` and `RTEMS_PRIORITY` attributes. The discipline defines the order in which tasks wait for a message to receive on a currently empty message queue.

- The **FIFO discipline** is the default and can be emphasized through use of the `RTEMS_FIFO` attribute.
- The **priority discipline** is selected by the `RTEMS_PRIORITY` attribute.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NUMBER**
The `count` parameter was invalid.

**RTEMS_INVALID_SIZE**
The `max_message_size` parameter was invalid.

**RTEMS_TOO_MANY**
There was no inactive object available to create a message queue. The number of message queue available to the application is configured through the *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615) application configuration option.

**RTEMS_TOO_MANY**
In multiprocessing configurations, there was no inactive global object available to create a global message queue. The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

**RTEMS_INVALID_NUMBER**
The product of `count` and `max_message_size` is greater than the maximum storage size.

**RTEMS_UNSATISFIED**
There was not enough memory available in the RTEMS Workspace to allocate the message buffers for the message queue.

NOTES:

For message queues with a global scope, the maximum message size is effectively limited to the longest message which the *MPCI* is capable of transmitting.

For control and maintenance of the message queue, RTEMS allocates a *QCB* from the local QCB free pool and initializes it.

The QCB for a global message queue is allocated on the local node. Message queues should not be made global unless remote tasks must interact with the message queue. This is to avoid the system overhead incurred by the creation of a global message queue. When a global message queue is created, the message queue's name and identifier must be transmitted to every node in the system for insertion in the local copy of the global object table.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of message queues available to the application is configured through the *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 14.4.2 rtems_message_queue_construct()

Constructs a message queue from the specified the message queue configuration.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_construct(
  const rtems_message_queue_config *config,
  rtems_id                         *id
);
```

PARAMETERS:

**config**
This parameter is the pointer to an *rtems_message_queue_config* (page 47) object. It configures the message queue.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the constructed message queue will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `config` parameter was NULL.

**RTEMS_INVALID_NAME**
The message queue name in the configuration was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NUMBER**
The maximum number of pending messages in the configuration was zero.

**RTEMS_INVALID_SIZE**
The maximum message size in the configuration was zero.

**RTEMS_TOO_MANY**
There was no inactive message queue object available to construct a message queue.

**RTEMS_TOO_MANY**
In multiprocessing configurations, there was no inactive global object available to construct a global message queue.

**RTEMS_INVALID_SIZE**
The maximum message size in the configuration was too big and resulted in integer overflows in calculations carried out to determine the size of the message buffer area.

**RTEMS_INVALID_NUMBER**
The maximum number of pending messages in the configuration was too big and resulted in integer overflows in calculations carried out to determine the size of the message buffer area.

**RTEMS_UNSATISFIED**
The message queue storage area begin pointer in the configuration was NULL.

**RTEMS_UNSATISFIED**
The message queue storage area size in the configuration was not equal to the size calculated from the maximum number of pending messages and the maximum message size.

NOTES:

In contrast to message queues created by *rtems_message_queue_create()* (page 372), the message queues constructed by this directive use a user-provided message buffer storage area.

This directive is intended for applications which do not want to use the RTEMS Workspace and instead statically allocate all operating system resources. An application based solely on static allocation can avoid any runtime memory allocators. This can simplify the application architecture as well as any analysis that may be required.

The value for *CONFIGURE_MESSAGE_BUFFER_MEMORY* (page 581) should not include memory for message queues constructed by *rtems_message_queue_construct()* (page 375).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of message queues available to the application is configured through the *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 14.4.3 rtems_message_queue_ident()

Identifies a message queue by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_ident(
  rtems_name name,
  uint32_t   node,
  rtems_id  *id
);
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**node**
This parameter is the node or node set to search for a matching object.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a message queue identifier associated with the message queue name specified in `name`.

The node to search is specified in `node`. It shall be

- a valid node number,
- the constant `RTEMS_SEARCH_ALL_NODES` to search in all nodes,
- the constant `RTEMS_SEARCH_LOCAL_NODE` to search in the local node only, or
- the constant `RTEMS_SEARCH_OTHER_NODES` to search in all nodes except the local node.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the specified nodes.

**RTEMS_INVALID_NODE**
In multiprocessing configurations, the specified node was invalid.

NOTES:

If the message queue name is not unique, then the message queue identifier will match the first message queue with that name in the search order. However, this message queue identifier is not guaranteed to correspond to the desired message queue.

The objects are searched from lowest to the highest index. If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched from lowest to the highest node number.

If node is a valid node number which does not represent the local node, then only the message queues exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

The message queue identifier is used with other message related directives to access the message queue.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 14.4.4 rtems_message_queue_delete()

Deletes the message queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the message queue identifier.

DESCRIPTION:

This directive deletes the message queue specified by `id`. As a result of this directive, all tasks blocked waiting to receive a message from this queue will be readied and returned a status code which indicates that the message queue was deleted.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no message queue associated with the identifier specified by `id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The message queue resided on a remote node.

NOTES:

When the message queue is deleted, any messages in the queue are returned to the free message buffer pool. Any information stored in those messages is lost. The message buffers allocated for the message queue are reclaimed.

The *QCB* for the deleted message queue is reclaimed by RTEMS.

When a global message queue is deleted, the message queue identifier must be transmitted to every node in the system for deletion from the local copy of the global object table.

The message queue must reside on the local node, even if the message queue was created with the `RTEMS_GLOBAL` attribute.

Proxies, used to represent remote tasks, are reclaimed when the message queue is deleted.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 14.4.5 rtems_message_queue_send()

Puts the message at the rear of the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_send(
  rtems_id    id,
  const void *buffer,
  size_t      size
);
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**buffer**
This parameter is the begin address of the message buffer to send.

**size**
This parameter is the size in bytes of the message buffer to send.

DESCRIPTION:

This directive sends the message `buffer` of `size` bytes in length to the queue specified by `id`. If a task is waiting at the queue, then the message is copied to the waiting task's buffer and the task is unblocked. If no tasks are waiting at the queue, then the message is copied to a message buffer which is obtained from this message queue's message buffer pool. The message buffer is then placed at the rear of the queue.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `buffer` parameter was NULL.

**RTEMS_INVALID_SIZE**
The size of the message exceeded the maximum message size of the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375).

**RTEMS_TOO_MANY**
The maximum number of pending messages supported by the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375) has been reached.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 14.4.6 rtems_message_queue_urgent()

Puts the message at the front of the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_urgent(
  rtems_id    id,
  const void *buffer,
  size_t      size
);
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**buffer**
This parameter is the begin address of the message buffer to send urgently.

**size**
This parameter is the size in bytes of the message buffer to send urgently.

DESCRIPTION:

This directive sends the message `buffer` of `size` bytes in length to the queue specified by `id`. If a task is waiting at the queue, then the message is copied to the waiting task's buffer and the task is unblocked. If no tasks are waiting at the queue, then the message is copied to a message buffer which is obtained from this message queue's message buffer pool. The message buffer is then placed at the front of the queue.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `buffer` parameter was NULL.

**RTEMS_INVALID_SIZE**
The size of the message exceeded the maximum message size of the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375).

**RTEMS_TOO_MANY**
The maximum number of pending messages supported by the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375) has been reached.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 14.4.7 rtems_message_queue_broadcast()

Broadcasts the messages to the tasks waiting at the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_broadcast(
  rtems_id    id,
  const void *buffer,
  size_t      size,
  uint32_t   *count
);
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**buffer**
This parameter is the begin address of the message buffer to broadcast.

**size**
This parameter is the size in bytes of the message buffer to broadcast.

**count**
This parameter is the pointer to an uint32_t object. When the directive call is successful, the number of unblocked tasks will be stored in this object.

DESCRIPTION:

This directive causes all tasks that are waiting at the queue specified by `id` to be unblocked and sent the message contained in `buffer`. Before a task is unblocked, the message `buffer` of `size` bytes in length is copied to that task's message buffer. The number of tasks that were unblocked is returned in `count`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `buffer` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `count` parameter was NULL.

**RTEMS_INVALID_SIZE**
The size of the message exceeded the maximum message size of the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375).

NOTES:

The execution time of this directive is directly related to the number of tasks waiting on the message queue, although it is more efficient than the equivalent number of invocations of *rtems_message_queue_send()* (page 381).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 14.4.8 rtems_message_queue_receive()

Receives a message from the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_receive(
  rtems_id       id,
  void          *buffer,
  size_t        *size,
  rtems_option   option_set,
  rtems_interval timeout
);
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**buffer**
This parameter is the begin address of the buffer to receive the message. The buffer shall be large enough to receive a message of the maximum length of the queue as defined by *rtems_message_queue_create()* (page 372) or *rtems_message_queue_construct()* (page 375). The `size` parameter cannot be used to specify the size of the buffer.

**size**
This parameter is the pointer to a size_t object. When the directive call is successful, the size in bytes of the received messages will be stored in this object. This parameter cannot be used to specify the size of the buffer.

**option_set**
This parameter is the option set.

**timeout**
This parameter is the timeout in *clock ticks* if the `RTEMS_WAIT` option is set. Use `RTEMS_NO_TIMEOUT` to wait potentially forever.

DESCRIPTION:

This directive receives a message from the queue specified by `id`.

The **option set** specified in `option_set` is built through a *bitwise or* of the option constants described below. Not all combinations of options are allowed. Some options are mutually exclusive. If mutually exclusive options are combined, the behaviour is undefined. Options not mentioned below are not evaluated by this directive and have no effect. Default options can be selected by using the `RTEMS_DEFAULT_OPTIONS` constant.

The calling task can **wait** or **try to receive** a message from the queue according to the mutually exclusive `RTEMS_WAIT` and `RTEMS_NO_WAIT` options.

- **Waiting to receive** a message from the queue is the default and can be emphasized through the use of the `RTEMS_WAIT` option. The `timeout` parameter defines how long the calling task is willing to wait. Use `RTEMS_NO_TIMEOUT` to wait potentially forever, otherwise set a timeout interval in clock ticks.

- **Trying to receive** a message from the queue is selected by the `RTEMS_NO_WAIT` option. If this option is defined, then the `timeout` parameter is ignored. When a message from the queue cannot be immediately received, then the `RTEMS_UNSATISFIED` status is returned.

With either `RTEMS_WAIT` or `RTEMS_NO_WAIT` if there is at least one message in the queue, then it is copied to the buffer, the size is set to return the length of the message in bytes, and this directive returns immediately with the `RTEMS_SUCCESSFUL` status code. The buffer has to be big enough to receive a message of the maximum length with respect to this message queue.

If the calling task chooses to return immediately and the queue is empty, then the directive returns immediately with the `RTEMS_UNSATISFIED` status code. If the calling task chooses to wait at the message queue and the queue is empty, then the calling task is placed on the message wait queue and blocked. If the queue was created with the `RTEMS_PRIORITY` option specified, then the calling task is inserted into the wait queue according to its priority. But, if the queue was created with the `RTEMS_FIFO` option specified, then the calling task is placed at the rear of the wait queue.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `buffer` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `size` parameter was NULL.

**RTEMS_UNSATISFIED**
The queue was empty.

**RTEMS_TIMEOUT**
The timeout happened while the calling task was waiting to receive a message

**RTEMS_OBJECT_WAS_DELETED**
The queue was deleted while the calling task was waiting to receive a message.

CONSTRAINTS:

The following constraints apply to this directive:

- When a local queue is accessed and the `RTEMS_NO_WAIT` option is set, the directive may be called from within interrupt context.
- The directive may be called from within task context.
- When the request cannot be immediately satisfied and the `RTEMS_WAIT` option is set, the calling task blocks at some point during the directive call.
- The timeout functionality of the directive requires a *clock tick*.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 14.4.9 rtems_message_queue_get_number_pending()

Gets the number of messages pending on the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_get_number_pending(
  rtems_id  id,
  uint32_t *count
);
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**count**
This parameter is the pointer to an uint32_t object. When the directive call is successful, the number of pending messages will be stored in this object.

DESCRIPTION:

This directive returns the number of messages pending on the queue specified by `id` in `count`. If no messages are present on the queue, count is set to zero.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `count` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may be called from within interrupt context.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 14.4.10 rtems_message_queue_flush()

Flushes all messages on the queue.

CALLING SEQUENCE:

```
rtems_status_code rtems_message_queue_flush( rtems_id id, uint32_t *count );
```

PARAMETERS:

**id**
This parameter is the queue identifier.

**count**
This parameter is the pointer to an uint32_t object. When the directive call is successful, the number of pending messages removed from the queue will be stored in this object.

DESCRIPTION:

This directive removes all pending messages from the queue specified by `id`. The number of messages removed is returned in `count`. If no messages are present on the queue, count is set to zero.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no queue associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `count` parameter was NULL.

NOTES:

The directive does not flush tasks waiting to receive a message from the *wait queue* of the message queue.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 14.4.11 RTEMS_MESSAGE_QUEUE_BUFFER()

Defines a structure which can be used as a message queue buffer for messages of the specified maximum size.

CALLING SEQUENCE:

```
RTEMS_MESSAGE_QUEUE_BUFFER( size_t maximum_message_size );
```

PARAMETERS:

**maximum_message_size**
This parameter is the maximum message size in bytes.

NOTES:

Use this macro to define the message buffer storage area for *rtems_message_queue_construct()* (page 375).

CHAPTER

**FIFTEEN**

# EVENT MANAGER

## 15.1 Introduction

The Event Manager provides a high performance method of inter-task communication and synchronization. The directives provided by the Event Manager are:

- *rtems_event_send()* (page 399) - Sends the event set to the task.
- *rtems_event_receive()* (page 401) - Receives or gets an event set from the calling task.

## 15.2 Background

### 15.2.1 Event Sets

An event flag is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty-two event flags are associated with each task. A collection of one or more event flags is referred to as an event set. The data type `rtems_event_set` is used to manage event sets.

The application developer should remember the following key characteristics of event operations when utilizing the event manager:

- Events provide a simple synchronization facility.
- Events are aimed at tasks.
- Tasks can wait on more than one event simultaneously.
- Events are independent of one another.
- Events do not hold or transport data.
- Events are not queued. In other words, if an event is sent more than once to a task before being received, the second and subsequent send operations to that same task have no effect.

An event set is posted when it is directed (or sent) to a task. A pending event is an event that has been posted but not received. An event condition is used to specify the event set which the task desires to receive and the algorithm which will be used to determine when the request is satisfied. An event condition is satisfied based upon one of two algorithms which are selected by the user. The `RTEMS_EVENT_ANY` algorithm states that an event condition is satisfied when at least a single requested event is posted. The `RTEMS_EVENT_ALL` algorithm states that an event condition is satisfied when every requested event is posted.

### 15.2.2 Building an Event Set or Condition

An event set or condition is built by a bitwise OR of the desired events. The set of valid events is `RTEMS_EVENT_0` through `RTEMS_EVENT_31`. If an event is not explicitly specified in the set or condition, then it is not present. Events are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each event appears exactly once in the event set list.

For example, when sending the event set consisting of `RTEMS_EVENT_6`, `RTEMS_EVENT_15`, and `RTEMS_EVENT_31`, the event parameter to the `rtems_event_send` directive should be `RTEMS_EVENT_6 | RTEMS_EVENT_15 | RTEMS_EVENT_31`.

### 15.2.3 Building an EVENT_RECEIVE Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_event_receive` directive are listed in the following table:

| | |
|---|---|
| `RTEMS_WAIT` | task will wait for event (default) |
| `RTEMS_NO_WAIT` | task should not wait |
| `RTEMS_EVENT_ALL` | return after all events (default) |
| `RTEMS_EVENT_ANY` | return after any events |

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each option appears exactly once in the component list. An option listed as a default is not required to appear in the option list, although it is a good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for all events in a particular event condition to arrive. The option parameter passed to the `rtems_event_receive` directive should be either `RTEMS_EVENT_ALL | RTEMS_NO_WAIT` or `RTEMS_NO_WAIT`. The option parameter can be set to `RTEMS_NO_WAIT` because `RTEMS_EVENT_ALL` is the default condition for `rtems_event_receive`.

## 15.3 Operations

### 15.3.1 Sending an Event Set

The `rtems_event_send` directive allows a task (or an ISR) to direct an event set to a target task. Based upon the state of the target task, one of the following situations applies:

- Target Task is Blocked Waiting for Events
  - If the waiting task's input event condition is satisfied, then the task is made ready for execution.
  - If the waiting task's input event condition is not satisfied, then the event set is posted but left pending and the task remains blocked.
- Target Task is Not Waiting for Events
  - The event set is posted and left pending.

### 15.3.2 Receiving an Event Set

The `rtems_event_receive` directive is used by tasks to accept a specific input event condition. The task also specifies whether the request is satisfied when all requested events are available or any single requested event is available. If the requested event condition is satisfied by pending events, then a successful return code and the satisfying event set are returned immediately. If the condition is not satisfied, then one of the following situations applies:

- By default, the calling task will wait forever for the event condition to be satisfied.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the period the task will wait before returning with an error status code.

### 15.3.3 Determining the Pending Event Set

A task can determine the pending event set by calling the `rtems_event_receive` directive with a value of `RTEMS_PENDING_EVENTS` for the input event condition. The pending events are returned to the calling task but the event set is left unaltered.

### 15.3.4 Receiving all Pending Events

A task can receive all of the currently pending events by calling the `rtems_event_receive` directive with a value of `RTEMS_ALL_EVENTS` for the input event condition and `RTEMS_NO_WAIT | RTEMS_EVENT_ANY` for the option set. The pending events are returned to the calling task and the event set is cleared. If no events are pending then the `RTEMS_UNSATISFIED` status code will be returned.

## 15.4 Directives

This section details the directives of the Event Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 15.4.1 rtems_event_send()

Sends the event set to the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_event_send( rtems_id id, rtems_event_set event_in );
```

PARAMETERS:

**id**
This parameter is the identifier of the target task to receive the event set.

**event_in**
This parameter is the event set to send.

DESCRIPTION:

This directive sends the event set, `event_in`, to the target task identified by `id`. Based upon the state of the target task, one of the following situations applies:

- The target task is blocked waiting for events, then
  - if the waiting task's input event condition is satisfied, then the task is made ready for execution, or
  - otherwise, the event set is posted but left pending and the task remains blocked.
- The target task is not waiting for events, then the event set is posted and left pending.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

NOTES:

Events can be sent by tasks or an *ISR*.

Specifying `RTEMS_SELF` for `id` results in the event set being sent to the calling task.

The event set to send shall be built by a *bitwise or* of the desired events. The set of valid events is `RTEMS_EVENT_0` through `RTEMS_EVENT_31`. If an event is not explicitly specified in the set, then it is not present.

Identical events sent to a task are not queued. In other words, the second, and subsequent, posting of an event to a task before it can perform an *rtems_event_receive()* (page 401) has no effect.

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending an event set to a global task which does not reside on the local node will generate a request telling the remote node to send the event set to the appropriate task.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.

### 15.4.2 rtems_event_receive()

Receives or gets an event set from the calling task.

CALLING SEQUENCE:

```
rtems_status_code rtems_event_receive(
  rtems_event_set  event_in,
  rtems_option     option_set,
  rtems_interval   ticks,
  rtems_event_set *event_out
);
```

PARAMETERS:

**event_in**
This parameter is the event set of interest. Use `RTEMS_PENDING_EVENTS` to get the pending events.

**option_set**
This parameter is the option set.

**ticks**
This parameter is the timeout in clock ticks if the `RTEMS_WAIT` option is set. Use `RTEMS_NO_TIMEOUT` to wait potentially forever.

**event_out**
This parameter is the pointer to an event set. The received or pending events are stored in the referenced event set if the operation was successful.

DESCRIPTION:

This directive can be used to

- get the pending events of the calling task, or
- receive events.

To **get the pending events** use the constant `RTEMS_PENDING_EVENTS` for the `event_in` parameter. The pending events are returned to the calling task but the event set of the calling task is left unaltered. The `option_set` and `ticks` parameters are ignored in this case. The directive returns immediately and does not block.

To **receive events** you have to define an input event condition and some options.

The **option set** specified in `option_set` is built through a *bitwise or* of the option constants described below. Not all combinations of options are allowed. Some options are mutually exclusive. If mutually exclusive options are combined, the behaviour is undefined. Options not mentioned below are not evaluated by this directive and have no effect. Default options can be selected by using the `RTEMS_DEFAULT_OPTIONS` constant. The option set defines

- if the calling task will wait or poll for the events, and
- if the calling task wants to receive all or any of the input events.

The calling task can **wait** or **poll** for the events.

- **Waiting** for events is the default and can be emphasized through the use of the `RTEMS_WAIT` option. The `ticks` parameter defines how long the calling task is willing to wait. Use `RTEMS_NO_TIMEOUT` to wait potentially forever, otherwise set a timeout interval in clock ticks.
- Not waiting for events (**polling**) is selected by the `RTEMS_NO_WAIT` option. If this option is defined, then the `ticks` parameter is ignored.

The calling task can receive **all** or **any** of the input events specified in `event_in`.

- Receiving **all** input events is the default and can be emphasized through the use of the `RTEMS_EVENT_ALL` option.
- Receiving **any** of the input events is selected by the `RTEMS_EVENT_ANY` option.

RETURN VALUES:

**`RTEMS_SUCCESSFUL`**
The requested operation was successful.

**`RTEMS_INVALID_ADDRESS`**
The `event_out` parameter was NULL.

**`RTEMS_UNSATISFIED`**
The events of interest were not immediately available.

**`RTEMS_TIMEOUT`**
The events of interest were not available within the specified timeout interval.

NOTES:

This directive only affects the events specified in `event_in`. Any pending events that do not correspond to any of the events specified in `event_in` will be left pending.

To receive all events use the event set constant `RTEMS_ALL_EVENTS` for the `event_in` parameter. Do not confuse this event set constant with the directive option `RTEMS_EVENT_ALL`.

A task can **receive all of the pending events** by calling the directive with a value of `RTEMS_ALL_EVENTS` for the `event_in` parameter and the bitwise or of the `RTEMS_NO_WAIT` and `RTEMS_EVENT_ANY` options for the `option_set` parameter. The pending events are returned and the event set of the task is cleared. If no events are pending then the `RTEMS_UNSATISFIED` status code will be returned.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The timeout functionality of the directive requires a *clock tick*.

CHAPTER

**SIXTEEN**

# SIGNAL MANAGER

## 16.1 Introduction

The Signal Manager provides the capabilities required for asynchronous communication. The directives provided by the Signal Manager are:

- *rtems_signal_catch()* (page 410) - Establishes an asynchronous signal routine (ASR) for the calling task.
- *rtems_signal_send()* (page 412) - Sends the signal set to the task.

## 16.2 Background

### 16.2.1 Signal Manager Definitions

The signal manager allows a task to optionally define an asynchronous signal routine (ASR). An ASR is to a task what an ISR is to an application's set of tasks. When the processor is interrupted, the execution of an application is also interrupted and an ISR is given control. Similarly, when a signal is sent to a task, that task's execution path will be "interrupted" by the ASR. Sending a signal to a task has no effect on the receiving task's current execution state.

A signal flag is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty-two signal flags are associated with each task. A collection of one or more signals is referred to as a signal set. The data type `rtems_signal_set` is used to manipulate signal sets.

A signal set is posted when it is directed (or sent) to a task. A pending signal is a signal that has been sent to a task with a valid ASR, but has not been processed by that task's ASR.

### 16.2.2 A Comparison of ASRs and ISRs

The format of an ASR is similar to that of an ISR with the following exceptions:

- ISRs are scheduled by the processor hardware. ASRs are scheduled by RTEMS.
- ISRs do not execute in the context of a task and may invoke only a subset of directives. ASRs execute in the context of a task and may execute any directive.
- When an ISR is invoked, it is passed the vector number as its argument. When an ASR is invoked, it is passed the signal set as its argument.
- An ASR has a task mode which can be different from that of the task. An ISR does not execute as a task and, as a result, does not have a task mode.

### 16.2.3 Building a Signal Set

A signal set is built by a bitwise OR of the desired signals. The set of valid signals is `RTEMS_SIGNAL_0` through `RTEMS_SIGNAL_31`. If a signal is not explicitly specified in the signal set, then it is not present. Signal values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each signal appears exactly once in the component list.

This example demonstrates the signal parameter used when sending the signal set consisting of `RTEMS_SIGNAL_6`, `RTEMS_SIGNAL_15`, and `RTEMS_SIGNAL_31`. The signal parameter provided to the `rtems_signal_send` directive should be `RTEMS_SIGNAL_6 | RTEMS_SIGNAL_15 | RTEMS_SIGNAL_31`.

### 16.2.4 Building an ASR Mode

In general, an ASR's mode is built by a bitwise OR of the desired mode components. The set of valid mode components is the same as those allowed with the task_create and task_mode directives. A complete list of mode options is provided in the following table:

| | |
|---|---|
| RTEMS_PREEMPT | is masked by RTEMS_PREEMPT_MASK and enables preemption |
| RTEMS_NO_PREEMPT | is masked by RTEMS_PREEMPT_MASK and disables preemption |
| RTEMS_NO_TIMESLICE | is masked by RTEMS_TIMESLICE_MASK and disables timeslicing |
| RTEMS_TIMESLICE | is masked by RTEMS_TIMESLICE_MASK and enables timeslicing |
| RTEMS_ASR | is masked by RTEMS_ASR_MASK and enables ASR processing |
| RTEMS_NO_ASR | is masked by RTEMS_ASR_MASK and disables ASR processing |
| RTEMS_INTERRUPT_LEVEL(0) | is masked by RTEMS_INTERRUPT_MASK and enables all interrupts |
| RTEMS_INTERRUPT_LEVEL(n) | is masked by RTEMS_INTERRUPT_MASK and sets interrupts level n |

Mode values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each mode appears exactly once in the component list. A mode component listed as a default is not required to appear in the mode list, although it is a good programming practice to specify default components. If all defaults are desired, the mode DEFAULT_MODES should be specified on this call.

This example demonstrates the mode parameter used with the rtems_signal_catch to establish an ASR which executes at interrupt level three and is non-preemptible. The mode should be set to RTEMS_INTERRUPT_LEVEL(3) | RTEMS_NO_PREEMPT to indicate the desired processor mode and interrupt level.

## 16.3 Operations

### 16.3.1 Establishing an ASR

The `rtems_signal_catch` directive establishes an ASR for the calling task. The address of the ASR and its execution mode are specified to this directive. The ASR's mode is distinct from the task's mode. For example, the task may allow preemption, while that task's ASR may have preemption disabled. Until a task calls `rtems_signal_catch` the first time, its ASR is invalid, and no signal sets can be sent to the task.

A task may invalidate its ASR and discard all pending signals by calling `rtems_signal_catch` with a value of NULL for the ASR's address. When a task's ASR is invalid, new signal sets sent to this task are discarded.

A task may disable ASR processing (`RTEMS_NO_ASR`) via the task_mode directive. When a task's ASR is disabled, the signals sent to it are left pending to be processed later when the ASR is enabled.

Any directive that can be called from a task can also be called from an ASR. A task is only allowed one active ASR. Thus, each call to `rtems_signal_catch` replaces the previous one.

Normally, signal processing is disabled for the ASR's execution mode, but if signal processing is enabled for the ASR, the ASR must be reentrant.

### 16.3.2 Sending a Signal Set

The `rtems_signal_send` directive allows both tasks and ISRs to send signals to a target task. The target task and a set of signals are specified to the `rtems_signal_send` directive. The sending of a signal to a task has no effect on the execution state of that task. If the task is not the currently running task, then the signals are left pending and processed by the task's ASR the next time the task is dispatched to run. The ASR is executed immediately before the task is dispatched. If the currently running task sends a signal to itself or is sent a signal from an ISR, its ASR is immediately dispatched to run provided signal processing is enabled.

If an ASR with signals enabled is preempted by another task or an ISR and a new signal set is sent, then a new copy of the ASR will be invoked, nesting the preempted ASR. Upon completion of processing the new signal set, control will return to the preempted ASR. In this situation, the ASR must be reentrant.

Like events, identical signals sent to a task are not queued. In other words, sending the same signal multiple times to a task (without any intermediate signal processing occurring for the task), has the same result as sending that signal to that task once.

### 16.3.3 Processing an ASR

Asynchronous signals were designed to provide the capability to generate software interrupts. The processing of software interrupts parallels that of hardware interrupts. As a result, the differences between the formats of ASRs and ISRs is limited to the meaning of the single argument passed to an ASR. The ASR should have the following calling sequence and adhere to C calling conventions:

```
rtems_asr user_routine(
    rtems_signal_set signals
);
```

When the ASR returns to RTEMS the mode and execution path of the interrupted task (or ASR) is restored to the context prior to entering the ASR.

## 16.4 Directives

This section details the directives of the Signal Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 16.4.1 rtems_signal_catch()

Establishes an asynchronous signal routine (ASR) for the calling task.

CALLING SEQUENCE:

```
rtems_status_code rtems_signal_catch(
  rtems_asr_entry asr_handler,
  rtems_mode      mode_set
);
```

PARAMETERS:

**asr_handler**
This parameter is the handler to process an asynchronous signal set.

**mode_set**
This parameter is the task mode while an asynchronous signal set is processed by the handler. See *rtems_task_mode()* (page 132).

DESCRIPTION:

This directive establishes an asynchronous signal routine (ASR) for the calling task. The `asr_handler` parameter specifies the entry point of the ASR. A task may have at most one handler installed at a time. The most recently installed handler is used. When `asr_handler` is NULL, the ASR for the calling task is invalidated and all pending signals are cleared. Any signals sent to a task with an invalid ASR are discarded. The `mode_set` parameter specifies the execution mode for the ASR. This execution mode supersedes the task's execution mode while the ASR is executing.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_NOT_IMPLEMENTED**
The `RTEMS_NO_PREEMPT` was set in `mode_set` and the system configuration had no implementation for this mode.

**RTEMS_NOT_IMPLEMENTED**
The `RTEMS_INTERRUPT_LEVEL()` was set to a positive level in `mode_set` and the system configuration had no implementation for this mode.

NOTES:

It is strongly recommended to disable ASR processing during ASR processing by setting `RTEMS_NO_ASR` in `mode_set`, otherwise a recursion may happen during ASR processing. Uncontrolled recursion may lead to stack overflows.

Using the same mutex (in particular a recursive mutex) in normal task context and during ASR processing may result in undefined behaviour.

Asynchronous signal handlers can access thread-local storage (*TLS*). When thread-local storage is shared between normal task context and ASR processing, it may be protected by disabled interrupts.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 16.4.2 rtems_signal_send()

Sends the signal set to the task.

CALLING SEQUENCE:

```
rtems_status_code rtems_signal_send(
  rtems_id         id,
  rtems_signal_set signal_set
);
```

PARAMETERS:

**id**
This parameter is the identifier of the target task to receive the signal set.

**signal_set**
This parameter is the signal set to send.

DESCRIPTION:

This directive sends the signal set, `signal_set`, to the target task identified by `id`.

If a caller sends a signal set to a task with an invalid *ASR*, then an error code is returned to the caller. If a caller sends a signal set to a task whose ASR is valid but disabled, then the signal set will be caught and left pending for the ASR to process when it is enabled. If a caller sends a signal set to a task with an ASR that is both valid and enabled, then the signal set is caught and the ASR will execute the next time the task is dispatched to run.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The `signal_set` parameter was 0.

**RTEMS_INVALID_ID**
There was no task associated with the identifier specified by `id`.

**RTEMS_NOT_DEFINED**
The target task had no valid ASR installed.

NOTES:

Sending a signal set to a task has no effect on that task's state. If a signal set is sent to a blocked task, then the task will remain blocked and the signals will be processed when the task becomes the running task.

Sending a signal set to a global task which does not reside on the local node will generate a request telling the remote node to send the signal set to the specified task.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- When the directive operates on a local object, the directive will not cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

CHAPTER

**SEVENTEEN**

# PARTITION MANAGER

## 17.1 Introduction

The Partition Manager provides facilities to dynamically allocate memory in fixed-size units. The directives provided by the Partition Manager are:

- *rtems_partition_create()* (page 420) - Creates a partition.
- *rtems_partition_ident()* (page 423) - Identifies a partition by the object name.
- *rtems_partition_delete()* (page 425) - Deletes the partition.
- *rtems_partition_get_buffer()* (page 427) - Tries to get a buffer from the partition.
- *rtems_partition_return_buffer()* (page 429) - Returns the buffer to the partition.

## 17.2 Background

### 17.2.1 Partition Manager Definitions

A partition is a physically contiguous memory area divided into fixed-size buffers that can be dynamically allocated and deallocated.

Partitions are managed and maintained as a list of buffers. Buffers are obtained from the front of the partition's free buffer chain and returned to the rear of the same chain. When a buffer is on the free buffer chain, RTEMS uses two pointers of memory from each buffer as the free buffer chain. When a buffer is allocated, the entire buffer is available for application use. Therefore, modifying memory that is outside of an allocated buffer could destroy the free buffer chain or the contents of an adjacent allocated buffer.

### 17.2.2 Building a Partition Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid partition attributes is provided in the following table:

| | |
|---|---|
| RTEMS_LOCAL | local partition (default) |
| RTEMS_GLOBAL | global partition |

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute RTEMS_DEFAULT_ATTRIBUTES should be specified on this call. The attribute_set parameter should be RTEMS_GLOBAL to indicate that the partition is to be known globally.

## 17.3 Operations

### 17.3.1 Creating a Partition

The `rtems_partition_create` directive creates a partition with a user-specified name. The partition's name, starting address, length and buffer size are all specified to the `rtems_partition_create` directive. RTEMS allocates a Partition Control Block (PTCB) from the PTCB free list. This data structure is used by RTEMS to manage the newly created partition. The number of buffers in the partition is calculated based upon the specified partition length and buffer size. If successful,the unique partition ID is returned to the calling task.

### 17.3.2 Obtaining Partition IDs

When a partition is created, RTEMS generates a unique partition ID and assigned it to the created partition until it is deleted. The partition ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_partition_create` directive, the partition ID is stored in a user provided location. Second, the partition ID may be obtained later using the `rtems_partition_ident` directive. The partition ID is used by other partition manager directives to access this partition.

### 17.3.3 Acquiring a Buffer

A buffer can be obtained by calling the `rtems_partition_get_buffer` directive. If a buffer is available, then it is returned immediately with a successful return code. Otherwise, an unsuccessful return code is returned immediately to the caller. Tasks cannot block to wait for a buffer to become available.

### 17.3.4 Releasing a Buffer

Buffers are returned to a partition's free buffer chain with the `rtems_partition_return_buffer` directive. This directive returns an error status code if the returned buffer was not previously allocated from this partition.

### 17.3.5 Deleting a Partition

The `rtems_partition_delete` directive allows a partition to be removed and returned to RTEMS. When a partition is deleted, the PTCB for that partition is returned to the PTCB free list. A partition with buffers still allocated cannot be deleted. Any task attempting to do so will be returned an error status code.

## 17.4 Directives

This section details the directives of the Partition Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 17.4.1 rtems_partition_create()

Creates a partition.

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_create(
  rtems_name      name,
  void           *starting_address,
  uintptr_t       length,
  size_t          buffer_size,
  rtems_attribute attribute_set,
  rtems_id       *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the partition.

**starting_address**
This parameter is the starting address of the buffer area used by the partition.

**length**
This parameter is the length in bytes of the buffer area used by the partition.

**buffer_size**
This parameter is the size in bytes of a buffer managed by the partition.

**attribute_set**
This parameter is the attribute set of the partition.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created partition will be stored in this object.

DESCRIPTION:

This directive creates a partition of fixed size buffers from a physically contiguous memory space which starts at `starting_address` and is `length` bytes in size. Each allocated buffer is to be of `buffer_size` in bytes. The partition has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the partition with other partition related directives.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant.

The partition has a local or global **scope** in a multiprocessing network (this attribute does not refer to SMP systems). The scope is selected by the mutually exclusive `RTEMS_LOCAL` and `RTEMS_GLOBAL` attributes.

- A **local scope** is the default and can be emphasized through the use of the `RTEMS_LOCAL` attribute. A local partition can be only used by the node which created it.

- A **global scope** is established if the `RTEMS_GLOBAL` attribute is set. The memory space used for the partition must reside in shared memory. Setting the global attribute in a single node system has no effect.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_SIZE**
The `length` parameter was 0.

**RTEMS_INVALID_SIZE**
The `buffer_size` parameter was 0.

**RTEMS_INVALID_SIZE**
The `length` parameter was less than the `buffer_size` parameter.

**RTEMS_INVALID_SIZE**
The `buffer_size` parameter was not an integral multiple of the pointer size.

**RTEMS_INVALID_SIZE**
The `buffer_size` parameter was less than two times the pointer size.

**RTEMS_INVALID_ADDRESS**
The `starting_address` parameter was not on a pointer size boundary.

**RTEMS_TOO_MANY**
There was no inactive object available to create a partition. The number of partitions available to the application is configured through the *CONFIGURE_MAXIMUM_PARTITIONS* (page 616) application configuration option.

**RTEMS_TOO_MANY**
In multiprocessing configurations, there was no inactive global object available to create a global semaphore. The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

NOTES:

The partition buffer area specified by the `starting_address` must be properly aligned. It must be possible to directly store target architecture pointers and also the user data. For example, if the user data contains some long double or vector data types, the partition buffer area and the buffer size must take the alignment of these types into account which is usually larger than the pointer alignment. A cache line alignment may be also a factor. Use `RTEMS_PARTITION_ALIGNMENT` to specify the minimum alignment of a partition buffer type.

The `buffer_size` parameter must be an integral multiple of the pointer size on the target architecture. Additionally, `buffer_size` must be large enough to hold two pointers on the target architecture. This is required for RTEMS to manage the buffers when they are free.

For control and maintenance of the partition, RTEMS allocates a *PTCB* from the local PTCB free pool and initializes it. Memory from the partition buffer area is not used by RTEMS to store the PTCB.

The PTCB for a global partition is allocated on the local node. Partitions should not be made global unless remote tasks must interact with the partition. This is to avoid the overhead incurred by the creation of a global partition. When a global partition is created, the partition's name and identifier must be transmitted to every node in the system for insertion in the local copy of the global object table.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The number of partitions available to the application is configured through the *CONFIGURE_MAXIMUM_PARTITIONS* (page 616) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.
- The number of global objects available to the application is configured through the *CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS* (page 735) application configuration option.

### 17.4.2 rtems_partition_ident()

Identifies a partition by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_ident(
  rtems_name name,
  uint32_t   node,
  rtems_id  *id
);
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**node**
This parameter is the node or node set to search for a matching object.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a partition identifier associated with the partition name specified in `name`.

The node to search is specified in `node`. It shall be

- a valid node number,
- the constant `RTEMS_SEARCH_ALL_NODES` to search in all nodes,
- the constant `RTEMS_SEARCH_LOCAL_NODE` to search in the local node only, or
- the constant `RTEMS_SEARCH_OTHER_NODES` to search in all nodes except the local node.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the specified nodes.

**RTEMS_INVALID_NODE**
In multiprocessing configurations, the specified node was invalid.

NOTES:

If the partition name is not unique, then the partition identifier will match the first partition with that name in the search order. However, this partition identifier is not guaranteed to correspond to the desired partition.

The objects are searched from lowest to the highest index. If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched from lowest to the highest node number.

If node is a valid node number which does not represent the local node, then only the partitions exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

The partition identifier is used with other partition related directives to access the partition.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 17.4.3 rtems_partition_delete()

Deletes the partition.

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the partition identifier.

DESCRIPTION:

This directive deletes the partition specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no partition associated with the identifier specified by `id`.

**RTEMS_ILLEGAL_ON_REMOTE_OBJECT**
The partition resided on a remote node.

**RTEMS_RESOURCE_IN_USE**
There were buffers of the partition still in use.

NOTES:

The partition cannot be deleted if any of its buffers are still allocated.

The *PTCB* for the deleted partition is reclaimed by RTEMS.

When a global partition is deleted, the partition identifier must be transmitted to every node in the system for deletion from the local copy of the global object table.

The partition must reside on the local node, even if the partition was created with the `RTEMS_GLOBAL` attribute.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the directive operates on a global object, the directive sends a message to remote nodes. This may preempt the calling task.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.

- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 17.4.4 rtems_partition_get_buffer()

Tries to get a buffer from the partition.

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_get_buffer( rtems_id id, void **buffer );
```

PARAMETERS:

**id**
This parameter is the partition identifier.

**buffer**
This parameter is the pointer to a `void` pointer object. When the directive call is successful, the pointer to the allocated buffer will be stored in this object.

DESCRIPTION:

This directive allows a buffer to be obtained from the partition specified by `id`. The address of the allocated buffer is returned through the `buffer` parameter.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no partition associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The `buffer` parameter was NULL.

**RTEMS_UNSATISFIED**
There was no free buffer available to allocate and return.

NOTES:

The buffer start alignment is determined by the memory area and buffer size used to create the partition.

A task cannot wait on a buffer to become available.

Getting a buffer from a global partition which does not reside on the local node will generate a request telling the remote node to allocate a buffer from the partition.

CONSTRAINTS:

The following constraints apply to this directive:

- When the directive operates on a local object, the directive may be called from within interrupt context.
- The directive may be called from within task context.
- When the directive operates on a local object, the directive will not cause the calling task to be preempted.

- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

### 17.4.5 rtems_partition_return_buffer()

Returns the buffer to the partition.

CALLING SEQUENCE:

```
rtems_status_code rtems_partition_return_buffer( rtems_id id, void *buffer );
```

PARAMETERS:

**id**
This parameter is the partition identifier.

**buffer**
This parameter is the pointer to the buffer to return.

DESCRIPTION:

This directive returns the buffer specified by `buffer` to the partition specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no partition associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The buffer referenced by `buffer` was not in the partition.

NOTES:

Returning a buffer multiple times is an error. It will corrupt the internal state of the partition.

CONSTRAINTS:

The following constraints apply to this directive:

- When the directive operates on a local object, the directive may be called from within interrupt context.
- The directive may be called from within task context.
- When the directive operates on a local object, the directive will not cause the calling task to be preempted.
- When the directive operates on a remote object, the directive sends a message to the remote node and waits for a reply. This will preempt the calling task.

CHAPTER

**EIGHTEEN**

# REGION MANAGER

## 18.1 Introduction

The Region Manager provides facilities to dynamically allocate memory in variable sized units. The directives provided by the Region Manager are:

- *rtems_region_create()* (page 438) - Creates a region.
- *rtems_region_ident()* (page 440) - Identifies a region by the object name.
- *rtems_region_delete()* (page 441) - Deletes the region.
- *rtems_region_extend()* (page 442) - Extends the region.
- *rtems_region_get_segment()* (page 444) - Gets a segment from the region.
- *rtems_region_return_segment()* (page 446) - Returns the segment to the region.
- *rtems_region_resize_segment()* (page 447) - Changes the size of the segment.
- *rtems_region_get_information()* (page 449) - Gets the region information.
- *rtems_region_get_free_information()* (page 450) - Gets the region free information.
- *rtems_region_get_segment_size()* (page 452) - Gets the size of the region segment.

## 18.2 Background

### 18.2.1 Region Manager Definitions

A region makes up a physically contiguous memory space with user-defined boundaries from which variable-sized segments are dynamically allocated and deallocated. A segment is a variable size section of memory which is allocated in multiples of a user-defined page size. This page size is required to be a multiple of four greater than or equal to four. For example, if a request for a 350-byte segment is made in a region with 256-byte pages, then a 512-byte segment is allocated.

Regions are organized as doubly linked chains of variable sized memory blocks. Memory requests are allocated using a first-fit algorithm. If available, the requester receives the number of bytes requested (rounded up to the next page size). RTEMS requires some overhead from the region's memory for each segment that is allocated. Therefore, an application should only modify the memory of a segment that has been obtained from the region. The application should NOT modify the memory outside of any obtained segments and within the region's boundaries while the region is currently active in the system.

Upon return to the region, the free block is coalesced with its neighbors (if free) on both sides to produce the largest possible unused block.

### 18.2.2 Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The set of valid region attributes is provided in the following table:

| | |
|---|---|
| `RTEMS_FIFO` | tasks wait by FIFO (default) |
| `RTEMS_PRIORITY` | tasks wait by priority |

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. An attribute listed as a default is not required to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute `RTEMS_DEFAULT_ATTRIBUTES` should be specified on this call.

This example demonstrates the attribute_set parameter needed to create a region with the task priority waiting queue discipline. The attribute_set parameter to the `rtems_region_create` directive should be `RTEMS_PRIORITY`.

### 18.2.3 Building an Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the `rtems_region_get_segment` directive are listed in the following table:

| | |
|---|---|
| `RTEMS_WAIT` | task will wait for segment (default) |
| `RTEMS_NO_WAIT` | task should not wait |

Option values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each option appears exactly once in the component list. An option listed as a default is not required to appear in the option list, although it is a

good programming practice to specify default options. If all defaults are desired, the option `RTEMS_DEFAULT_OPTIONS` should be specified on this call.

This example demonstrates the option parameter needed to poll for a segment. The option parameter passed to the `rtems_region_get_segment` directive should be `RTEMS_NO_WAIT`.

## 18.3 Operations

### 18.3.1 Creating a Region

The `rtems_region_create` directive creates a region with the user-defined name. The user may select FIFO or task priority as the method for placing waiting tasks in the task wait queue. RTEMS allocates a Region Control Block (RNCB) from the RNCB free list to maintain the newly created region. RTEMS also generates a unique region ID which is returned to the calling task.

It is not possible to calculate the exact number of bytes available to the user since RTEMS requires overhead for each segment allocated. For example, a region with one segment that is the size of the entire region has more available bytes than a region with two segments that collectively are the size of the entire region. This is because the region with one segment requires only the overhead for one segment, while the other region requires the overhead for two segments.

Due to automatic coalescing, the number of segments in the region dynamically changes. Therefore, the total overhead required by RTEMS dynamically changes.

### 18.3.2 Obtaining Region IDs

When a region is created, RTEMS generates a unique region ID and assigns it to the created region until it is deleted. The region ID may be obtained by either of two methods. First, as the result of an invocation of the `rtems_region_create` directive, the region ID is stored in a user provided location. Second, the region ID may be obtained later using the `rtems_region_ident` directive. The region ID is used by other region manager directives to access this region.

### 18.3.3 Adding Memory to a Region

The `rtems_region_extend` directive may be used to add memory to an existing region. The caller specifies the size in bytes and starting address of the memory being added.

### 18.3.4 Acquiring a Segment

The `rtems_region_get_segment` directive attempts to acquire a segment from a specified region. If the region has enough available free memory, then a segment is returned successfully to the caller. When the segment cannot be allocated, one of the following situations applies:

- By default, the calling task will wait forever to acquire the segment.
- Specifying the `RTEMS_NO_WAIT` option forces an immediate return with an error status code.
- Specifying a timeout limits the interval the task will wait before returning with an error status code.

If the task waits for the segment, then it is placed in the region's task wait queue in either FIFO or task priority order. All tasks waiting on a region are returned an error when the message queue is deleted.

### 18.3.5 Releasing a Segment

When a segment is returned to a region by the `rtems_region_return_segment` directive, it is merged with its unallocated neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

### 18.3.6 Obtaining the Size of a Segment

The `rtems_region_get_segment_size` directive returns the size in bytes of the specified segment. The size returned includes any "extra" memory included in the segment because of rounding up to a page size boundary.

### 18.3.7 Changing the Size of a Segment

The `rtems_region_resize_segment` directive is used to change the size in bytes of the specified segment. The size may be increased or decreased. When increasing the size of a segment, it is possible that the request cannot be satisfied. This directive provides functionality similar to the `realloc()` function in the Standard C Library.

### 18.3.8 Deleting a Region

A region can be removed from the system and returned to RTEMS with the `rtems_region_delete` directive. When a region is deleted, its control block is returned to the RNCB free list. A region with segments still allocated is not allowed to be deleted. Any task attempting to do so will be returned an error. As a result of this directive, all tasks blocked waiting to obtain a segment from the region will be readied and returned a status code which indicates that the region was deleted.

## 18.4 Directives

This section details the directives of the Region Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 18.4.1 rtems_region_create()

Creates a region.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_create(
  rtems_name      name,
  void           *starting_address,
  uintptr_t       length,
  uintptr_t       page_size,
  rtems_attribute attribute_set,
  rtems_id       *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the region.

**starting_address**
This parameter is the starting address of the memory area managed by the region.

**length**
This parameter is the length in bytes of the memory area managed by the region.

**page_size**
This parameter is the alignment of the starting address and length of each allocated segment of the region.

**attribute_set**
This parameter is the attribute set of the region.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created region will be stored in this object.

DESCRIPTION:

This directive creates a region which resides on the local node. The region has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the region with other region related directives.

The region manages the **contiguous memory area** which starts at `starting_address` and is `length` bytes long. The memory area shall be large enough to contain some internal region administration data.

The **starting address** and **length of segments** allocated from the region will be an integral multiple of `page_size`. The specified page size will be aligned to an implementation-dependent minimum alignment if necessary.

The **attribute set** specified in `attribute_set` is built through a *bitwise or* of the attribute constants described below. Not all combinations of attributes are allowed. Some attributes are mutually exclusive. If mutually exclusive attributes are combined, the behaviour is undefined. Attributes not mentioned below are not evaluated by this directive and have no effect. Default attributes can be selected by using the `RTEMS_DEFAULT_ATTRIBUTES` constant.

The **task wait queue discipline** is selected by the mutually exclusive `RTEMS_FIFO` and `RTEMS_PRIORITY` attributes. The discipline defines the order in which tasks wait for allocatable segments on a currently empty region.

- The **FIFO discipline** is the default and can be emphasized through use of the `RTEMS_FIFO` attribute.
- The **priority discipline** is selected by the `RTEMS_PRIORITY` attribute.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `starting_address` parameter was NULL.

**RTEMS_TOO_MANY**
There was no inactive object available to create a region. The number of regions available to the application is configured through the *CONFIGURE_MAXIMUM_REGIONS* (page 619) application configuration option.

**RTEMS_INVALID_SIZE**
The `page_size` parameter was invalid.

**RTEMS_INVALID_SIZE**
The memory area specified in `starting_address` and `length` was too small.

NOTES:

For control and maintenance of the region, RTEMS allocates a *RNCB* from the local RNCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of regions available to the application is configured through the *CONFIGURE_MAXIMUM_REGIONS* (page 619) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 18.4.2 rtems_region_ident()

Identifies a region by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a region identifier associated with the region name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the region name is not unique, then the region identifier will match the first region with that name in the search order. However, this region identifier is not guaranteed to correspond to the desired region.

The objects are searched from lowest to the highest index. Only the local node is searched.

The region identifier is used with other region related directives to access the region.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 18.4.3 rtems_region_delete()

Deletes the region.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the region identifier.

DESCRIPTION:

This directive deletes the region specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

**RTEMS_RESOURCE_IN_USE**
There were segments of the region still in use.

NOTES:

The region cannot be deleted if any of its segments are still allocated.

The *RNCB* for the deleted region is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 18.4.4 rtems_region_extend()

Extends the region.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_extend(
  rtems_id  id,
  void     *starting_address,
  uintptr_t length
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**starting_address**
This parameter is the starting address of the memory area to extend the region.

**length**
This parameter is the length in bytes of the memory area to extend the region.

DESCRIPTION:

This directive adds the memory area which starts at `starting_address` for `length` bytes to the region specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `starting_address` parameter was NULL.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The memory area specified by `starting_address` and `length` was insufficient to extend the heap.

NOTES:

There are no alignment requirements for the memory area. The memory area must be big enough to contain some maintenance blocks. It must not overlap parts of the current heap memory areas. Disconnected memory areas added to the heap will lead to used blocks which cover the gaps. Extending with an inappropriate memory area will corrupt the heap resulting in undefined behaviour.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 18.4.5 rtems_region_get_segment()

Gets a segment from the region.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_segment(
  rtems_id       id,
  uintptr_t      size,
  rtems_option   option_set,
  rtems_interval timeout,
  void         **segment
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**size**
This parameter is the size in bytes of the segment to allocate.

**option_set**
This parameter is the option set.

**timeout**
This parameter is the timeout in *clock ticks* if the `RTEMS_WAIT` option is set. Use `RTEMS_NO_TIMEOUT` to wait potentially forever.

**segment**
This parameter is the pointer to a `void` pointer object. When the directive call is successful, the begin address of the allocated segment will be stored in this object.

DESCRIPTION:

This directive gets a segment from the region specified by `id`.

The **option set** specified in `option_set` is built through a *bitwise or* of the option constants described below. Not all combinations of options are allowed. Some options are mutually exclusive. If mutually exclusive options are combined, the behaviour is undefined. Options not mentioned below are not evaluated by this directive and have no effect. Default options can be selected by using the `RTEMS_DEFAULT_OPTIONS` constant.

The calling task can **wait** or **try to get** a segment from the region according to the mutually exclusive `RTEMS_WAIT` and `RTEMS_NO_WAIT` options.

- **Waiting to get** a segment from the region is the default and can be emphasized through the use of the `RTEMS_WAIT` option. The `timeout` parameter defines how long the calling task is willing to wait. Use `RTEMS_NO_TIMEOUT` to wait potentially forever, otherwise set a timeout interval in clock ticks.
- **Trying to get** a segment from the region is selected by the `RTEMS_NO_WAIT` option. If this option is defined, then the `timeout` parameter is ignored. When a segment from the region cannot be immediately allocated, then the `RTEMS_UNSATISFIED` status is returned.

With either RTEMS_WAIT or RTEMS_NO_WAIT if there is a segment of the requested size is available, then it is returned in segment and this directive returns immediately with the RTEMS_SUCCESSFUL status code.

If the calling task chooses to return immediately and the region has no segment of the requested size available, then the directive returns immediately with the RTEMS_UNSATISFIED status code. If the calling task chooses to wait for a segment, then the calling task is placed on the region wait queue and blocked. If the region was created with the RTEMS_PRIORITY option specified, then the calling task is inserted into the wait queue according to its priority. But, if the region was created with the RTEMS_FIFO option specified, then the calling task is placed at the rear of the wait queue.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The segment parameter was NULL.

**RTEMS_INVALID_SIZE**
The size parameter was zero.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by id.

**RTEMS_INVALID_SIZE**
The size parameter exceeded the maximum segment size which is possible for the region.

**RTEMS_UNSATISFIED**
The region had no segment of the requested size immediately available.

**RTEMS_TIMEOUT**
The timeout happened while the calling task was waiting to get a segment from the region.

NOTES:

The actual length of the allocated segment may be larger than the requested size because a segment size is always a multiple of the region's page size.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- When the request cannot be immediately satisfied and the RTEMS_WAIT option is set, the calling task blocks at some point during the directive call.
- The timeout functionality of the directive requires a *clock tick*.

### 18.4.6 rtems_region_return_segment()

Returns the segment to the region.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_return_segment( rtems_id id, void *segment );
```

PARAMETERS:

**id**
This parameter is the region identifier.

**segment**
This parameter is the begin address of the segment to return.

DESCRIPTION:

This directive returns the segment specified by `segment` to the region specified by `id`. The returned segment is merged with its neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The segment was not within the region.

NOTES:

This directive will cause the calling task to be preempted if one or more local tasks are waiting for a segment and the following conditions exist:

- A waiting task has a higher priority than the calling task.
- The size of the segment required by the waiting task is less than or equal to the size of the segment returned.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 18.4.7 rtems_region_resize_segment()

Changes the size of the segment.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_resize_segment(
  rtems_id   id,
  void      *segment,
  uintptr_t  size,
  uintptr_t *old_size
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**segment**
This parameter is the begin address of the segment to resize.

**size**
This parameter is the requested new size of the segment.

**old_size**
This parameter is the pointer to an uintptr_t object. When the directive call is successful, the old size of the segment will be stored in this object.

DESCRIPTION:

This directive is used to increase or decrease the size of the `segment` of the region specified by `id`. When increasing the size of a segment, it is possible that there is no memory available contiguous to the segment. In this case, the request is unsatisfied.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `old_size` parameter was NULL.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The segment was not within the region.

**RTEMS_UNSATISFIED**
The region was unable to resize the segment.

NOTES:

If an attempt to increase the size of a segment fails, then the application may want to allocate a new segment of the desired size, copy the contents of the original segment to the new, larger segment and then return the original segment.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 18.4.8 rtems_region_get_information()

Gets the region information.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_information(
  rtems_id                id,
  Heap_Information_block *the_info
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**the_info**
This parameter is the pointer to a `Heap_Information_block` object. When the directive call is successful, the information of the region will be stored in this object.

DESCRIPTION:

This directive is used to obtain information about the used and free memory in the region specified by `id`. This is a snapshot at the time of the call. The information will be returned in the structure pointed to by `the_info`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `the_info` parameter was NULL.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

NOTES:

This is primarily intended as a mechanism to obtain a diagnostic information. This method forms am O(n) scan of the free and an O(n) scan of the used blocks in the region to calculate the information provided. Given that the execution time is driven by the number of used and free blocks, it can take a non-deterministic time to execute.

To get only the free information of the region use *rtems_region_get_free_information()* (page 450).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 18.4.9 rtems_region_get_free_information()

Gets the region free information.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_free_information(
  rtems_id                id,
  Heap_Information_block *the_info
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**the_info**
This parameter is the pointer to a `Heap_Information_block` object. When the directive call is successful, the free information of the region will be stored in this object.

DESCRIPTION:

This directive is used to obtain information about the free memory in the region specified by `id`. This is a snapshot at the time of the call. The information will be returned in the structure pointed to by `the_info`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `the_info` parameter was NULL.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

NOTES:

This directive uses the same structure to return information as the *rtems_region_get_information()* (page 449) directive but does not fill in the used information.

This is primarily intended as a mechanism to obtain a diagnostic information. This method forms am O(n) scan of the free in the region to calculate the information provided. Given that the execution time is driven by the number of used and free blocks, it can take a non-deterministic time to execute. Typically, there are many used blocks and a much smaller number of used blocks making a call to this directive less expensive than a call to *rtems_region_get_information()* (page 449).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 18.4.10 rtems_region_get_segment_size()

Gets the size of the region segment.

CALLING SEQUENCE:

```
rtems_status_code rtems_region_get_segment_size(
  rtems_id   id,
  void      *segment,
  uintptr_t *size
);
```

PARAMETERS:

**id**
This parameter is the region identifier.

**segment**
This parameter is the begin address of the segment.

**size**
This parameter is the pointer to a uintptr_t object. When the directive call is successful, the size of the segment in bytes will be stored in this object.

DESCRIPTION:

This directive obtains the size in bytes of the segment specified by `segment` of the region specified by `id` in `size`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `segment` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `size` parameter was NULL.

**RTEMS_INVALID_ID**
There was no region associated with the identifier specified by `id`.

**RTEMS_INVALID_ADDRESS**
The segment was not within the region.

NOTES:

The actual length of the allocated segment may be larger than the requested size because a segment size is always a multiple of the region's page size.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

CHAPTER

**NINETEEN**

# DUAL-PORTED MEMORY MANAGER

## 19.1 Introduction

The Dual-Ported Memory Manager provides a mechanism for converting addresses between internal and external representations for multiple dual-ported memory areas (DPMA). The directives provided by the Dual-Ported Memory Manager are:

- *rtems_port_create()* (page 460) - Creates a port.
- *rtems_port_ident()* (page 462) - Identifies a port by the object name.
- *rtems_port_delete()* (page 463) - Deletes the port.
- *rtems_port_external_to_internal()* (page 464) - Converts the external address to the internal address.
- *rtems_port_internal_to_external()* (page 465) - Converts the internal address to the external address.

## 19.2 Background

A dual-ported memory area (DPMA) is an contiguous block of RAM owned by a particular processor but which can be accessed by other processors in the system. The owner accesses the memory using internal addresses, while other processors must use external addresses. RTEMS defines a port as a particular mapping of internal and external addresses.

There are two system configurations in which dual-ported memory is commonly found. The first is tightly-coupled multiprocessor computer systems where the dual-ported memory is shared between all nodes and is used for inter-node communication. The second configuration is computer systems with intelligent peripheral controllers. These controllers typically utilize the DPMA for high-performance data transfers.

## 19.3 Operations

### 19.3.1 Creating a Port

The `rtems_port_create` directive creates a port into a DPMA with the user-defined name. The user specifies the association between internal and external representations for the port being created. RTEMS allocates a Dual-Ported Memory Control Block (DPCB) from the DPCB free list to maintain the newly created DPMA. RTEMS also generates a unique dual-ported memory port ID which is returned to the calling task. RTEMS does not initialize the dual-ported memory area or access any memory within it.

### 19.3.2 Obtaining Port IDs

When a port is created, RTEMS generates a unique port ID and assigns it to the created port until it is deleted. The port ID may be obtained by either of two methods. First, as the result of an invocation of the``rtems_port_create`` directive, the task ID is stored in a user provided location. Second, the port ID may be obtained later using the `rtems_port_ident` directive. The port ID is used by other dual-ported memory manager directives to access this port.

### 19.3.3 Converting an Address

The `rtems_port_external_to_internal` directive is used to convert an address from external to internal representation for the specified port. The `rtems_port_internal_to_external` directive is used to convert an address from internal to external representation for the specified port. If an attempt is made to convert an address which lies outside the specified DPMA, then the address to be converted will be returned.

### 19.3.4 Deleting a DPMA Port

A port can be removed from the system and returned to RTEMS with the `rtems_port_delete` directive. When a port is deleted, its control block is returned to the DPCB free list.

## 19.4 Directives

This section details the directives of the Dual-Ported Memory Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 19.4.1 rtems_port_create()

Creates a port.

CALLING SEQUENCE:

```
rtems_status_code rtems_port_create(
  rtems_name name,
  void      *internal_start,
  void      *external_start,
  uint32_t   length,
  rtems_id  *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the port.

**internal_start**
This parameter is the internal start address of the memory area.

**external_start**
This parameter is the external start address of the memory area.

**length**
This parameter is the length in bytes of the memory area.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created port will be stored in this object.

DESCRIPTION:

This directive creates a port which resides on the local node. The port has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the port with other dual-ported memory port related directives.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `internal_start` parameter was not properly aligned.

**RTEMS_INVALID_ADDRESS**
The `external_start` parameter was not properly aligned.

**RTEMS_TOO_MANY**
There was no inactive object available to create a port. The number of port available to the

application is configured through the *CONFIGURE_MAXIMUM_PORTS* (page 618) application configuration option.

NOTES:

The `internal_start` and `external_start` parameters must be on a boundary defined by the target processor architecture.

For control and maintenance of the port, RTEMS allocates a *DPCB* from the local DPCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of ports available to the application is configured through the *CONFIGURE_MAXIMUM_PORTS* (page 618) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 19.4.2 rtems_port_ident()

Identifies a port by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_port_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains a port identifier associated with the port name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the port name is not unique, then the port identifier will match the first port with that name in the search order. However, this port identifier is not guaranteed to correspond to the desired port.

The objects are searched from lowest to the highest index. Only the local node is searched.

The port identifier is used with other dual-ported memory related directives to access the port.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 19.4.3 rtems_port_delete()

Deletes the port.

CALLING SEQUENCE:

```
rtems_status_code rtems_port_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the port identifier.

DESCRIPTION:

This directive deletes the port specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no port associated with the identifier specified by `id`.

NOTES:

The *DPCB* for the deleted port is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 19.4.4 rtems_port_external_to_internal()

Converts the external address to the internal address.

CALLING SEQUENCE:

```
rtems_status_code rtems_port_external_to_internal(
  rtems_id id,
  void    *external,
  void   **internal
);
```

PARAMETERS:

**id**
This parameter is the port identifier.

**external**
This parameter is the external address to convert.

**internal**
This parameter is the pointer to a `void` pointer object. When the directive call is successful, the external address associated with the internal address will be stored in this object.

DESCRIPTION:

This directive converts a dual-ported memory address from external to internal representation for the specified port. If the given external address is invalid for the specified port, then the internal address is set to the given external address.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `id` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `internal` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

### 19.4.5 rtems_port_internal_to_external()

Converts the internal address to the external address.

CALLING SEQUENCE:

```
rtems_status_code rtems_port_internal_to_external(
  rtems_id id,
  void    *internal,
  void   **external
);
```

PARAMETERS:

**id**
This parameter is the port identifier.

**internal**
This parameter is the internal address to convert.

**external**
This parameter is the pointer to a `void` pointer object. When the directive call is successful, the external address associated with the internal address will be stored in this object.

DESCRIPTION:

This directive converts a dual-ported memory address from internal to external representation so that it can be passed to owner of the DPMA represented by the specified port. If the given internal address is an invalid dual-ported address, then the external address is set to the given internal address.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `id` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `external` parameter was NULL.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive will not cause the calling task to be preempted.

CHAPTER

**TWENTY**

# I/O MANAGER

## 20.1 Introduction

The Input/Output (I/O) Manager provides a well-defined mechanism for accessing device drivers and a structured methodology for organizing device drivers. The directives provided by the I/O Manager are:

- *rtems_io_register_driver()* (page 474) - Registers and initializes the device with the specified device driver address table and device major number in the Device Driver Table.
- *rtems_io_unregister_driver()* (page 476) - Removes a device driver specified by the device major number from the Device Driver Table.
- *rtems_io_initialize()* (page 477) - Initializes the device specified by the device major and minor numbers.
- *rtems_io_register_name()* (page 478) - Registers the device specified by the device major and minor numbers in the file system under the specified name.
- *rtems_io_open()* (page 479) - Opens the device specified by the device major and minor numbers.
- *rtems_io_close()* (page 480) - Closes the device specified by the device major and minor numbers.
- *rtems_io_read()* (page 481) - Reads from the device specified by the device major and minor numbers.
- *rtems_io_write()* (page 482) - Writes to the device specified by the device major and minor numbers.
- *rtems_io_control()* (page 483) - Controls the device specified by the device major and minor numbers.

## 20.2 Background

### 20.2.1 Device Driver Table

Each application utilizing the RTEMS I/O manager must specify the address of a Device Driver Table in its Configuration Table. This table contains each device driver's entry points that is to be initialised by RTEMS during initialization. Each device driver may contain the following entry points:

- Initialization
- Open
- Close
- Read
- Write
- Control

If the device driver does not support a particular entry point, then that entry in the Configuration Table should be NULL. RTEMS will return `RTEMS_SUCCESSFUL` as the executive's and zero (0) as the device driver's return code for these device driver entry points.

Applications can register and unregister drivers with the RTEMS I/O manager avoiding the need to have all drivers statically defined and linked into this table.

The `confdefs.h` entry `CONFIGURE_MAXIMUM_DRIVERS` configures the number of driver slots available to the application.

### 20.2.2 Major and Minor Device Numbers

Each call to the I/O manager must provide a device's major and minor numbers as arguments. The major number is the index of the requested driver's entry points in the Device Driver Table, and is used to select a specific device driver. The exact usage of the minor number is driver specific, but is commonly used to distinguish between a number of devices controlled by the same driver.

The data types `rtems_device_major_number` and `rtems_device_minor_number` are used to manipulate device major and minor numbers, respectively.

### 20.2.3 Device Names

The I/O Manager provides facilities to associate a name with a particular device. Directives are provided to register the name of a device and to look up the major/minor number pair associated with a device name.

### 20.2.4 Device Driver Environment

Application developers, as well as device driver developers, must be aware of the following regarding the RTEMS I/O Manager:

- A device driver routine executes in the context of the invoking task. Thus if the driver blocks, the invoking task blocks.
- The device driver is free to change the modes of the invoking task, although the driver should restore them to their original values.

- Device drivers may be invoked from ISRs.
- Only local device drivers are accessible through the I/O manager.
- A device driver routine may invoke all other RTEMS directives, including I/O directives, on both local and global objects.

Although the RTEMS I/O manager provides a framework for device drivers, it makes no assumptions regarding the construction or operation of a device driver.

### 20.2.5 Runtime Driver Registration

Board support package and application developers can select wether a device driver is statically entered into the default device table or registered at runtime.

Dynamic registration helps applications where:

- The BSP and kernel libraries are common to a range of applications for a specific target platform. An application may be built upon a common library with all drivers. The application selects and registers the drivers. Uniform driver name lookup protects the application.
- The type and range of drivers may vary as the application probes a bus during initialization.
- Support for hot swap bus system such as Compact PCI.
- Support for runtime loadable driver modules.

### 20.2.6 Device Driver Interface

When an application invokes an I/O manager directive, RTEMS determines which device driver entry point must be invoked. The information passed by the application to RTEMS is then passed to the correct device driver entry point. RTEMS will invoke each device driver entry point assuming it is compatible with the following prototype:

```
rtems_device_driver io_entry(
    rtems_device_major_number  major,
    rtems_device_minor_number  minor,
    void                      *argument_block
);
```

The format and contents of the parameter block are device driver and entry point dependent.

It is recommended that a device driver avoid generating error codes which conflict with those used by application components. A common technique used to generate driver specific error codes is to make the most significant part of the status indicate a driver specific code.

### 20.2.7 Device Driver Initialization

RTEMS automatically initializes all device drivers when multitasking is initiated via the `rtems_initialize_executive` directive. RTEMS initializes the device drivers by invoking each device driver initialization entry point with the following parameters:

**major**
the major device number for this device driver.

**minor**
zero.

**argument_block**
will point to the Configuration Table.

The returned status will be ignored by RTEMS. If the driver cannot successfully initialize the device, then it should invoke the fatal_error_occurred directive.

## 20.3 Operations

### 20.3.1 Register and Lookup Name

The `rtems_io_register` directive associates a name with the specified device (i.e. major/minor number pair). Device names are typically registered as part of the device driver initialization sequence. The `rtems_io_lookup` directive is used to determine the major/minor number pair associated with the specified device name. The use of these directives frees the application from being dependent on the arbitrary assignment of major numbers in a particular application. No device naming conventions are dictated by RTEMS.

### 20.3.2 Accessing an Device Driver

The I/O manager provides directives which enable the application program to utilize device drivers in a standard manner. There is a direct correlation between the RTEMS I/O manager directives `rtems_io_initialize`, `rtems_io_open`, `rtems_io_close`, `rtems_io_read`, `rtems_io_write`, and `rtems_io_control` and the underlying device driver entry points.

## 20.4 Directives

This section details the directives of the I/O Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 20.4.1 rtems_io_register_driver()

Registers and initializes the device with the specified device driver address table and device major number in the Device Driver Table.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_register_driver(
  rtems_device_major_number         major,
  const rtems_driver_address_table *driver_table,
  rtems_device_major_number        *registered_major
);
```

PARAMETERS:

**major**
This parameter is the device major number. Use a value of zero to let the system obtain a device major number automatically.

**driver_table**
This parameter is the device driver address table.

**registered_major**
This parameter is the pointer to an *rtems_device_major_number* (page 39) object. When the directive call is successful, the device major number of the registered device will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The device major number of the device was NULL.

**RTEMS_INVALID_ADDRESS**
The device driver address table was empty.

**RTEMS_INVALID_NUMBER**
The device major number of the device was out of range, see *CONFIGURE_MAXIMUM_DRIVERS* (page 611).

**RTEMS_TOO_MANY**
The system was unable to obtain a device major number.

**RTEMS_RESOURCE_IN_USE**
The device major number was already in use.

**RTEMS_CALLED_FROM_ISR**
The directive was called from interrupt context.

Other status codes may be returned by *rtems_io_initialize()* (page 477).

NOTES:

If the device major number equals zero a device major number will be obtained. The device major number of the registered driver will be returned.

After a successful registration, the *rtems_io_initialize()* (page 477) directive will be called to initialize the device.

### 20.4.2 rtems_io_unregister_driver()

Removes a device driver specified by the device major number from the Device Driver Table.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_unregister_driver(
  rtems_device_major_number major
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_UNSATISFIED**
The device major number was invalid.

**RTEMS_CALLED_FROM_ISR**
The directive was called from interrupt context.

NOTES:

Currently no specific checks are made and the driver is not closed.

### 20.4.3 rtems_io_initialize()

Initializes the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_initialize(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver initialization entry.

DESCRIPTION:

This directive calls the device driver initialization entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver initialization entry.

NOTES:

This directive is automatically invoked for each device driver defined by the application configuration during the system initialization and via the *rtems_io_register_driver()* (page 474) directive.

A device driver initialization entry is responsible for initializing all hardware and data structures associated with a device. If necessary, it can allocate memory to be used during other operations.

### 20.4.4 rtems_io_register_name()

Registers the device specified by the device major and minor numbers in the file system under the specified name.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_register_name(
  const char               *device_name,
  rtems_device_major_number major,
  rtems_device_minor_number minor
);
```

PARAMETERS:

**device_name**
This parameter is the device name in the file system.

**major**
This parameter is the device major number.

**minor**
This parameter is the device minor number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_TOO_MANY**
The name was already in use or other errors occurred.

NOTES:

The device is registered as a character device.

### 20.4.5 rtems_io_open()

Opens the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_open(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver close entry.

DESCRIPTION:

This directive calls the device driver open entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver open entry.

NOTES:

The open entry point is commonly used by device drivers to provide exclusive access to a device.

### 20.4.6 rtems_io_close()

Closes the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_close(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver close entry.

DESCRIPTION:

This directive calls the device driver close entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver close entry.

NOTES:

The close entry point is commonly used by device drivers to relinquish exclusive access to a device.

### 20.4.7 rtems_io_read()

Reads from the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_read(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver read entry.

DESCRIPTION:

This directive calls the device driver read entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver read entry.

NOTES:

Read operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be replaced with data from the device.

### 20.4.8 rtems_io_write()

Writes to the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_write(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver write entry.

DESCRIPTION:

This directive calls the device driver write entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver write entry.

NOTES:

Write operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be sent to the device.

### 20.4.9 rtems_io_control()

Controls the device specified by the device major and minor numbers.

CALLING SEQUENCE:

```
rtems_status_code rtems_io_control(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void                      *argument
);
```

PARAMETERS:

**major**
This parameter is the major number of the device.

**minor**
This parameter is the minor number of the device.

**argument**
This parameter is the argument passed to the device driver I/O control entry.

DESCRIPTION:

This directive calls the device driver I/O control entry registered in the Device Driver Table for the specified device major number.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NUMBER**
The device major number was invalid.

Other status codes may be returned by the device driver I/O control entry.

NOTES:

The exact functionality of the driver entry called by this directive is driver dependent. It should not be assumed that the control entries of two device drivers are compatible. For example, an RS-232 driver I/O control operation may change the baud of a serial line, while an I/O control operation for a floppy disk driver may cause a seek operation.

CHAPTER

**TWENTYONE**

# KERNEL CHARACTER I/O SUPPORT

## 21.1 Introduction

The kernel character input/output support is an extension of the *I/O Manager* (page 467) to output characters to the kernel character output device and receive characters from the kernel character input device using a polled and non-blocking implementation.

The directives may be used to print debug and test information. The kernel character input/output support should work even if no Console Driver is configured, see *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597). The kernel character input and output device is provided by the *BSP*. Applications may change the device. The directives provided by the Kernel Character I/O Support are:

- *rtems_putc()* (page 488) - Outputs the character to the kernel character output device.
- *rtems_put_char()* (page 489) - Puts the character using *rtems_putc()* (page 488)
- *putk()* (page 490) - Outputs the characters of the string and a newline character to the kernel character output device.
- *printk()* (page 491) - Outputs the characters defined by the format string and the arguments to the kernel character output device.
- *vprintk()* (page 492) - Outputs the characters defined by the format string and the variable argument list to the kernel character output device.
- *rtems_printk_printer()* (page 493) - Outputs the characters defined by the format string and the variable argument list to the kernel character output device.
- *getchark()* (page 494) - Tries to dequeue a character from the kernel character input device.

## 21.2 Directives

This section details the directives of the Kernel Character I/O Support. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 21.2.1 rtems_putc()

Outputs the character to the kernel character output device.

CALLING SEQUENCE:

```
void rtems_putc( char c );
```

PARAMETERS:

**c**
This parameter is the character to output.

DESCRIPTION:

The directive outputs the character specified by c to the kernel character output device using the polled character output implementation provided by BSP_output_char. The directive performs a character translation from NL to CR followed by NR.

If the kernel character output device is concurrently accessed, then interleaved output may occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 21.2.2 rtems_put_char()

Puts the character using *rtems_putc()* (page 488)

CALLING SEQUENCE:

```
void rtems_put_char( int c, void *unused );
```

PARAMETERS:

**c**
This parameter is the character to output.

**unused**
This parameter is an unused argument.

NOTES:

The directive is provided to support the RTEMS Testing Framework.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 21.2.3 putk()

Outputs the characters of the string and a newline character to the kernel character output device.

CALLING SEQUENCE:

```
int putk( const char *s );
```

PARAMETERS:

**s**
This parameter is the string to output.

RETURN VALUES:

Returns the number of characters output to the kernel character output device.

NOTES:

The directive may be used to print debug and test information. It uses *rtems_putc()* (page 488) to output the characters. This directive performs a character translation from `NL` to `CR` followed by `NR`.

If the kernel character output device is concurrently accessed, then interleaved output may occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 21.2.4 printk()

Outputs the characters defined by the format string and the arguments to the kernel character output device.

CALLING SEQUENCE:

```
int printk( const char *fmt, ... );
```

PARAMETERS:

**fmt**
This parameter is a printf()-style format string.

**...**
This parameter is a list of optional parameters required by the format string.

RETURN VALUES:

Returns the number of characters output to the kernel character output device.

NOTES:

The directive may be used to print debug and test information. It uses *rtems_putc()* (page 488) to output the characters. This directive performs a character translation from `NL` to `CR` followed by `NR`.

If the kernel character output device is concurrently accessed, then interleaved output may occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- Formatting of floating point numbers is not supported.

### 21.2.5 vprintk()

Outputs the characters defined by the format string and the variable argument list to the kernel character output device.

CALLING SEQUENCE:

```
int vprintk( const char *fmt, va_list ap );
```

PARAMETERS:

**fmt**
This parameter is a printf()-style format string.

**ap**
This parameter is the variable argument list required by the format string.

RETURN VALUES:

Returns the number of characters output to the kernel character output device.

NOTES:

The directive may be used to print debug and test information. It uses *rtems_putc()* (page 488) to output the characters. This directive performs a character translation from `NL` to `CR` followed by `NR`.

If the kernel character output device is concurrently accessed, then interleaved output may occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- Formatting of floating point numbers is not supported.

### 21.2.6 rtems_printk_printer()

Outputs the characters defined by the format string and the variable argument list to the kernel character output device.

CALLING SEQUENCE:

```
int rtems_printk_printer( void *unused, const char *fmt, va_list ap );
```

PARAMETERS:

**unused**
This parameter is an unused argument.

**fmt**
This parameter is a printf()-style format string.

**ap**
This parameter is the variable argument list required by the format string.

RETURN VALUES:

Returns the number of characters output to the kernel character output device.

NOTES:

The directive may be used to print debug and test information. It uses *rtems_putc()* (page 488) to output the characters. This directive performs a character translation from `NL` to `CR` followed by `NR`.

If the kernel character output device is concurrently accessed, then interleaved output may occur.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.
- Formatting of floating point numbers is not supported.

### 21.2.7 getchark()

Tries to dequeue a character from the kernel character input device.

CALLING SEQUENCE:

```
int getchark( void );
```

DESCRIPTION:

The directive tries to dequeue a character from the kernel character input device using the polled character input implementation referenced by BSP_poll_char if it is available.

RETURN VALUES:

**-1**
The BSP_poll_char pointer was equal to NULL.

**-1**
There was no character enqueued on the kernel character input device.

Returns the character least recently enqueued on the kernel character input device as an unsigned character value.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

CHAPTER

**TWENTYTWO**

# CACHE MANAGER

## 22.1 Introduction

The Cache Manager provides functions to perform maintenance operations for data and instruction caches.

The actual actions of the Cache Manager operations depend on the hardware and the implementation provided by the CPU architecture port or a board support package. Cache implementations tend to be highly hardware dependent. The directives provided by the Cache Manager are:

- *rtems_cache_flush_multiple_data_lines()* (page 498) - Flushes the data cache lines covering the memory area.
- *rtems_cache_invalidate_multiple_data_lines()* (page 499) - Invalidates the data cache lines covering the memory area.
- *rtems_cache_invalidate_multiple_instruction_lines()* (page 500) - Invalidates the instruction cache lines covering the memory area.
- *rtems_cache_instruction_sync_after_code_change()* (page 501) - Ensures necessary synchronization required after code changes.
- *rtems_cache_get_maximal_line_size()* (page 502) - Gets the maximal cache line size in bytes of all caches (data, instruction, or unified).
- *rtems_cache_get_data_line_size()* (page 503) - Gets the data cache line size in bytes.
- *rtems_cache_get_instruction_line_size()* (page 504) - Gets the instruction cache line size in bytes.
- *rtems_cache_get_data_cache_size()* (page 505) - Gets the data cache size in bytes for the cache level.
- *rtems_cache_get_instruction_cache_size()* (page 506) - Gets the instruction cache size in bytes for the cache level.
- *rtems_cache_flush_entire_data()* (page 507) - Flushes the entire data cache.
- *rtems_cache_invalidate_entire_data()* (page 508) - Invalidates the entire data cache.
- *rtems_cache_invalidate_entire_instruction()* (page 509) - Invalidates the entire instruction cache.
- *rtems_cache_enable_data()* (page 510) - Enables the data cache.
- *rtems_cache_disable_data()* (page 511) - Disables the data cache.
- *rtems_cache_enable_instruction()* (page 512) - Enables the instruction cache.
- *rtems_cache_disable_instruction()* (page 513) - Disables the instruction cache.
- *rtems_cache_aligned_malloc()* (page 514) - Allocates memory from the C Program Heap which begins at a cache line boundary.

## 22.2 Directives

This section details the directives of the Cache Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 22.2.1 rtems_cache_flush_multiple_data_lines()

Flushes the data cache lines covering the memory area.

CALLING SEQUENCE:

```
void rtems_cache_flush_multiple_data_lines( const void *begin, size_t size );
```

PARAMETERS:

**begin**
This parameter is the begin address of the memory area to flush.

**size**
This parameter is the size in bytes of the memory area to flush.

DESCRIPTION:

Dirty data cache lines covering the area are transfered to memory. Depending on the cache implementation this may mark the lines as invalid.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.2 rtems_cache_invalidate_multiple_data_lines()

Invalidates the data cache lines covering the memory area.

CALLING SEQUENCE:

```
void rtems_cache_invalidate_multiple_data_lines(
  const void *begin,
  size_t      size
);
```

PARAMETERS:

**begin**
This parameter is the begin address of the memory area to invalidate.

**size**
This parameter is the size in bytes of the memory area to invalidate.

DESCRIPTION:

The cache lines covering the area are marked as invalid. A later read access in the area will load the data from memory.

NOTES:

In case the area is not aligned on cache line boundaries, then this operation may destroy unrelated data.

On some systems, the cache lines may be flushed before they are invalidated.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.3 rtems_cache_invalidate_multiple_instruction_lines()

Invalidates the instruction cache lines covering the memory area.

CALLING SEQUENCE:

```
void rtems_cache_invalidate_multiple_instruction_lines(
  const void *begin,
  size_t      size
);
```

PARAMETERS:

**begin**
This parameter is the begin address of the memory area to invalidate.

**size**
This parameter is the size in bytes of the memory area to invalidate.

DESCRIPTION:

The cache lines covering the area are marked as invalid. A later instruction fetch from the area will result in a load from memory.

NOTES:

In SMP configurations, on processors without instruction cache snooping, this operation will invalidate the instruction cache lines on all processors.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.4 rtems_cache_instruction_sync_after_code_change()

Ensures necessary synchronization required after code changes.

CALLING SEQUENCE:

```
void rtems_cache_instruction_sync_after_code_change(
  const void *begin,
  size_t      size
);
```

PARAMETERS:

**begin**
This parameter is the begin address of the code area to synchronize.

**size**
This parameter is the size in bytes of the code area to synchronize.

NOTES:

When code is loaded or modified, then most systems require synchronization instructions to update the instruction caches so that the loaded or modified code is fetched. For example, systems with separate data and instruction caches or systems without instruction cache snooping. The directives should be used by run time loader for example.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.5 rtems_cache_get_maximal_line_size()

Gets the maximal cache line size in bytes of all caches (data, instruction, or unified).

CALLING SEQUENCE:

```
size_t rtems_cache_get_maximal_line_size( void );
```

RETURN VALUES:

**0**
There is no cache present.

Returns the maximal cache line size in bytes of all caches (data, instruction, or unified).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.6 rtems_cache_get_data_line_size()

Gets the data cache line size in bytes.

CALLING SEQUENCE:

```
size_t rtems_cache_get_data_line_size( void );
```

RETURN VALUES:

**0**
There is no data cache present.

Returns the data cache line size in bytes. For multi-level caches this is the maximum of the cache line sizes of all levels.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.7 rtems_cache_get_instruction_line_size()

Gets the instruction cache line size in bytes.

CALLING SEQUENCE:

```
size_t rtems_cache_get_instruction_line_size( void );
```

RETURN VALUES:

**0**
There is no instruction cache present.

Returns the instruction cache line size in bytes. For multi-level caches this is the maximum of the cache line sizes of all levels.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.8 rtems_cache_get_data_cache_size()

Gets the data cache size in bytes for the cache level.

CALLING SEQUENCE:

```
size_t rtems_cache_get_data_cache_size( uint32_t level );
```

PARAMETERS:

**level**
This parameter is the requested data cache level. The cache level zero specifies the entire data cache.

RETURN VALUES:

**0**
There is no data cache present at the requested cache level.

Returns the data cache size in bytes of the requested cache level.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.9 rtems_cache_get_instruction_cache_size()

Gets the instruction cache size in bytes for the cache level.

CALLING SEQUENCE:

```
size_t rtems_cache_get_instruction_cache_size( uint32_t level );
```

PARAMETERS:

**level**
This parameter is the requested instruction cache level. The cache level zero specifies the entire instruction cache.

RETURN VALUES:

**0**
There is no instruction cache present at the requested cache level.

Returns the instruction cache size in bytes of the requested cache level.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.10 rtems_cache_flush_entire_data()

Flushes the entire data cache.

CALLING SEQUENCE:

```
void rtems_cache_flush_entire_data( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.11 rtems_cache_invalidate_entire_data()

Invalidates the entire data cache.

CALLING SEQUENCE:

```
void rtems_cache_invalidate_entire_data( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.12 rtems_cache_invalidate_entire_instruction()

Invalidates the entire instruction cache.

CALLING SEQUENCE:

```
void rtems_cache_invalidate_entire_instruction( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.13 rtems_cache_enable_data()

Enables the data cache.

CALLING SEQUENCE:

```
void rtems_cache_enable_data( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.14 rtems_cache_disable_data()

Disables the data cache.

CALLING SEQUENCE:

```
void rtems_cache_disable_data( void );
```

NOTES:

On some *targets* or configurations, calling this directive may cause a fatal error with a fatal source of *INTERNAL_ERROR_CORE* (page 518) and fatal code of *INTERNAL_ERROR_CANNOT_DISABLE_DATA_CACHE* (page 519). The data cache may be necessary to provide *atomic operations*. In SMP configurations, the data cache may be required to ensure data coherency. See the BSP documentation in the *RTEMS User Manual* for more information.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.15 rtems_cache_enable_instruction()

Enables the instruction cache.

CALLING SEQUENCE:

```
void rtems_cache_enable_instruction( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.16 rtems_cache_disable_instruction()

Disables the instruction cache.

CALLING SEQUENCE:

```
void rtems_cache_disable_instruction( void );
```

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 22.2.17 rtems_cache_aligned_malloc()

Allocates memory from the C Program Heap which begins at a cache line boundary.

CALLING SEQUENCE:

```
void *rtems_cache_aligned_malloc( size_t size );
```

PARAMETERS:

**size**
This parameter is the size in bytes of the memory area to allocate.

RETURN VALUES:

**NULL**
There is not enough memory available to satisfy the allocation request.

Returns the begin address of the allocated memory. The begin address is on a cache line boundary.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

CHAPTER

**TWENTYTHREE**

# FATAL ERROR MANAGER

## 23.1 Introduction

The Fatal Error Manager processes all fatal or irrecoverable errors and other sources of system termination (for example after `exit()`). Fatal errors are identified by the fatal source and code pair. The directives provided by the Fatal Error Manager are:

- *rtems_fatal()* (page 526) - Invokes the fatal error handler.
- *rtems_panic()* (page 527) - Prints the message and invokes the fatal error handler.
- *rtems_shutdown_executive()* (page 528) - Invokes the fatal error handler.
- *rtems_exception_frame_print()* (page 529) - Prints the exception frame.
- *rtems_fatal_source_text()* (page 530) - Returns a descriptive text for the fatal source.
- *rtems_internal_error_text()* (page 531) - Returns a descriptive text for the internal error code.
- *rtems_fatal_error_occurred()* (page 532) - Invokes the fatal error handler.

## 23.2 Background

### 23.2.1 Overview

The fatal error manager is called upon detection of an irrecoverable error condition by either RTEMS or the application software. Fatal errors are also used in case it is difficult or impossible to return an error condition by other means, e.g. a return value of a directive call. Fatal errors can be detected from various sources, for example

- the executive (RTEMS),
- support libraries,
- user system code,
- user application code, and
- processor interrupts and exceptions (data abort, instruction prefetch errors, ECC errors, spurious interrupts, etc.).

RTEMS automatically invokes the fatal error manager upon detection of an error it considers to be fatal. Similarly, the user should invoke the fatal error manager upon detection of a fatal error.

Each *user extensions* set may include a fatal error handler. The fatal error handler in the *initial extension sets* can be used to provide access to debuggers and monitors which may be present on the target hardware. If any user-supplied fatal error handlers are installed, the fatal error manager will invoke them. Usually, the board support package provides a fatal error extension which resets the board. If no user handlers are configured or if all the user handler return control to the fatal error manager, then the CPU port provided idle loop executes.

### 23.2.2 System Termination Procedure

The `_Terminate()` handler is invoked to terminate the system. It is called by all services which determine that a system termination is required. For example, it is called by all higher level directives which announce a fatal error, see *Announcing a Fatal Error* (page 524).

The first action of the system termination handler is to disable maskable interrupts. This ensures that interrupts on this processor do not interfere with the system termination procedure. This reduces the likelihood to end up in a recursive system termination procedure.

The second action of the system termination handler is to call the fatal extensions of the *user extensions*.

The fatal extensions are called with three parameters:

- the *fatal source* (page 518),
- a legacy parameter which is always set to `false`, and
- an error code with a fatal source dependent content.

The fatal extensions of the *initial extension sets* are invoked first. For them, the following execution environment is required

- a valid stack pointer and enough stack space,
- a valid code memory, and
- valid read-only data.

In uniprocessor configurations, the read-write data (including `.bss` segment) is not required. In SMP configurations, however, the read-write data must have been initialized to determine the state of the other processors and request them to shut-down if necessary. The board support package (BSP) may install an initial extension that performs a system reset. See the BSP documentation in the *RTEMS User Manual* for more information how the system reset is done. The BSP provided fatal extension can be disabled by the *CONFIGURE_DISABLE_BSP_SETTINGS* (page 568) application configuration option. It is recommended to provide an application-specific fatal extension using the *CONFIGURE_INITIAL_EXTENSIONS* (page 573) application configuration option.

In certain error conditions, it may be unreliable to carry out the following steps of the termination procedure since the read-write data may be corrupt. One of the fatal extensions of the initial extension set should reset the system to stop the system termination procedure.

After invoking the fatal extensions of the initial extension sets, the fatal extensions of the *dynamic extension sets* are invoked. For this procedure valid read-write data is required.

The last action of the system termination handler is to execute the CPU port provided idle loop with maskable interrupts disabled. Please note, that properly configured applications should not reach this point.

### 23.2.3 Fatal Sources

The following fatal sources are defined for RTEMS via the `rtems_fatal_source` enumeration. Each symbolic name has the corresponding numeric fatal source in parenthesis.

**INTERNAL_ERROR_CORE (0)**
Errors of the core operating system. See *Internal Error Codes* (page 519).

**INTERNAL_ERROR_RTEMS_API (1)**
Errors of the Classic API.

**INTERNAL_ERROR_POSIX_API (2)**
Errors of the POSIX API.

**RTEMS_FATAL_SOURCE_BDBUF (3)**
Fatal source for the block device cache. See `rtems_bdbuf_fatal_code`.

**RTEMS_FATAL_SOURCE_APPLICATION (4)**
Fatal source for application-specific errors. The fatal code is application-specific.

**RTEMS_FATAL_SOURCE_EXIT (5)**
Fatal source of `exit()`. The fatal code is the `exit()` status code.

**RTEMS_FATAL_SOURCE_BSP (6)**
Fatal source for BSP errors. The fatal codes are defined in `<bsp/fatal.h>`. Examples are interrupt and exception initialization. See `bsp_fatal_code` and `bsp_fatal()`.

**RTEMS_FATAL_SOURCE_ASSERT (7)**
Fatal source of `assert()`. The fatal code is the pointer value of the assert context. See `rtems_assert_context`.

**RTEMS_FATAL_SOURCE_STACK_CHECKER (8)**
Fatal source of the stack checker. The fatal code is the object name of the executing task.

**RTEMS_FATAL_SOURCE_EXCEPTION (9)**
Fatal source of the exceptions. The fatal code is the pointer value of the exception frame pointer. See `rtems_exception_frame` and rtems_exception_frame_print.

**RTEMS_FATAL_SOURCE_SMP (10)**
Fatal source of SMP domain. See `SMP_Fatal_code`.

**RTEMS_FATAL_SOURCE_PANIC (11)**
Fatal source of `rtems_panic()`, see rtems_panic.

**RTEMS_FATAL_SOURCE_INVALID_HEAP_FREE (12)**
Fatal source for invalid C program heap frees via `free()`. The fatal code is the bad pointer.

**RTEMS_FATAL_SOURCE_HEAP (13)**
Fatal source for heap errors. The fatal code is the address to a heap error context. See `Heap_Error_context`.

### 23.2.4 Internal Error Codes

The following error codes are defined for the `INTERNAL_ERROR_CORE` fatal source. Each symbolic name has the corresponding numeric error code in parenthesis.

**INTERNAL_ERROR_TOO_LITTLE_WORKSPACE (2)**
There is not enough memory for the workspace. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_THREAD_EXITTED (5)**
A non-POSIX thread entry function returned. This is an API usage error.

An example code to provoke this fatal error is:

```
rtems_task task( rtems_task_argument arg )
{
  /* Classic API tasks must not return */
}

void create_bad_task( void )
{
  rtems_status_code sc;
  rtems_id          task_id;

  sc = rtems_task_create(
    rtems_build_name('T', 'A', 'S', 'K'),
    1,
    RTEMS_MINIMUM_STACK_SIZE,
    RTEMS_DEFAULT_MODES,
    RTEMS_DEFAULT_ATTRIBUTES,
    &task_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_task_start( task_id, task, 0 );
  assert( sc == RTEMS_SUCCESSFUL );
}
```

**INTERNAL_ERROR_INCONSISTENT_MP_INFORMATION (6)**
This fatal error can only occur on MPCI configurations. The MPCI nodes or global objects configuration is inconsistent. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_INVALID_NODE (7)**
This fatal error can only occur on MPCI configurations. The own MPCI node number is invalid. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_NO_MPCI (8)**
This fatal error can only occur on MPCI configurations. There is no MPCI configuration table. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_BAD_PACKET (9)**
This fatal error can only occur on MPCI configurations. The MPCI server thread received a bad packet.

**INTERNAL_ERROR_OUT_OF_PACKETS (10)**
This fatal error can only occur on MPCI configurations. The MPCI packet pool is empty. It is an application configuration error.

**INTERNAL_ERROR_OUT_OF_GLOBAL_OBJECTS (11)**
This fatal error can only occur on MPCI configurations. The MPCI global objects pool is empty. It is an application configuration error.

**INTERNAL_ERROR_OUT_OF_PROXIES (12)**
This fatal error can only occur on MPCI configurations. The MPCI thread proxy pool is empty. It is an application configuration error.

**INTERNAL_ERROR_INVALID_GLOBAL_ID (13)**
This fatal error can only occur on MPCI configurations. The system cannot find the global object for a specific object identifier. In case this happens, then this is probably an operating system bug.

**INTERNAL_ERROR_NO_MEMORY_FOR_HEAP (23)**
There is not enough memory for the C program heap. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_CPU_ISR_INSTALL_VECTOR (24)**
The use of `_CPU_ISR_install_vector()` is illegal on this system.

**INTERNAL_ERROR_RESOURCE_IN_USE (25)**
This fatal error can only occur on debug configurations. It happens in case a thread which owns mutexes is deleted. Mutexes owned by a deleted thread are in an inconsistent state.

**INTERNAL_ERROR_RTEMS_INIT_TASK_ENTRY_IS_NULL (26)**
An RTEMS initialization task entry function is NULL. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_THREAD_QUEUE_DEADLOCK (28)**
A deadlock was detected during a thread queue enqueue operation.

**INTERNAL_ERROR_THREAD_QUEUE_ENQUEUE_STICKY_FROM_BAD_STATE (29)**
This fatal error can only happen in SMP configurations. It is not allowed to obtain MrsP semaphores in a context with thread dispatching disabled, for example interrupt context.

An example code to provoke this fatal error is:

```
rtems_timer_service_routine bad( rtems_id timer_id, void *arg )
{
  rtems_id *sem_id;

```

(continues on next page)

(continued from previous page)

```
  sem_id = arg;

  rtems_semaphore_obtain( *sem_id, RTEMS_WAIT, RTEMS_NO_TIMEOUT );
  assert( 0 );
}

rtems_task fire_bad_timer( rtems_task_argument arg )
{
  rtems_status_code sc;
  rtems_id          sem_id;
  rtems_id          timer_id;

  sc = rtems_semaphore_create(
    rtems_build_name('M', 'R', 'S', 'P'),
    1,
    RTEMS_MULTIPROCESSOR_RESOURCE_SHARING
      | RTEMS_BINARY_SEMAPHORE,
    1,
    &sem_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_timer_create(
    rtems_build_name( 'E', 'V', 'I', 'L' ),
    &timer_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_semaphore_obtain( sem_id, RTEMS_WAIT, RTEMS_NO_TIMEOUT );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_timer_fire_after( timer_id, 1, bad, &sem_id );
  assert( sc == RTEMS_SUCCESSFUL );

  rtems_task_wake_after( 2 );
  assert( 0 );
}
```

**INTERNAL_ERROR_BAD_THREAD_DISPATCH_DISABLE_LEVEL (30)**

It is illegal to call blocking operating system services with thread dispatching disabled, for example in interrupt context.

An example code to provoke this fatal error is:

```
void bad( rtems_id id, void *arg )
{
  rtems_task_wake_after( RTEMS_YIELD_PROCESSOR );
  assert( 0 );
}

```

(continues on next page)

(continued from previous page)

```
void fire_bad_timer( void )
{
  rtems_status_code sc;
  rtems_id          id;

  sc = rtems_timer_create(
    rtems_build_name( 'E', 'V', 'I', 'L' ),
    &id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_timer_fire_after( id, 1, bad, NULL );
  assert( sc == RTEMS_SUCCESSFUL );

  rtems_task_wake_after( 2 );
  assert( 0 );
}
```

**INTERNAL_ERROR_BAD_THREAD_DISPATCH_ENVIRONMENT (31)**

In SMP configurations, it is a fatal error to call blocking operating system with interrupts disabled, since this prevents delivery of inter-processor interrupts. This could lead to executing threads which are not allowed to execute resulting in undefined system behaviour.

Some CPU ports, for example the ARM Cortex-M port, have a similar problem, since the interrupt state is not a part of the thread context.

This fatal error is detected in the operating system core function `_Thread_Do_dispatch()` responsible to carry out a thread dispatch.

An example code to provoke this fatal error is:

```
void bad( void )
{
  rtems_interrupt_level level;

  rtems_interrupt_local_disable( level );
  rtems_task_suspend( RTEMS_SELF );
  rtems_interrupt_local_enable( level  );
}
```

**INTERNAL_ERROR_RTEMS_INIT_TASK_CREATE_FAILED (32)**

The creation of the RTEMS initialization task failed. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_POSIX_INIT_THREAD_CREATE_FAILED (33)**

The creation of the POSIX initialization thread failed. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_LIBIO_STDOUT_FD_OPEN_FAILED (36)**

Open of the standard output file descriptor failed or resulted in an unexpected file descriptor number. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_LIBIO_STDERR_FD_OPEN_FAILED (37)**
Open of the standard error file descriptor failed or resulted in an unexpected file descriptor number. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_ILLEGAL_USE_OF_FLOATING_POINT_UNIT (38)**
The floating point unit was used illegally, for example in interrupt context on some architectures.

**INTERNAL_ERROR_ARC4RANDOM_GETENTROPY_FAIL (39)**
A `getentropy()` system call failed in one of the ARC4RANDOM(3) functions. This fatal error can only be fixed with a different implementation of `getentropy()`.

**INTERNAL_ERROR_NO_MEMORY_FOR_PER_CPU_DATA (40)**
This fatal error may happen during workspace initialization. There is not enough memory available to populate the per-CPU data areas, see <rtems/score/percpudata.h>.

**INTERNAL_ERROR_TOO_LARGE_TLS_SIZE (41)**
This fatal error may happen during system initialization. The actual thread-local storage (TLS) size of the application exceeds the configured maximum, see *CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE* (page 578). You can get the thread-local storage size of an application using the RTEMS tool `rtems-execinfo`.

**INTERNAL_ERROR_RTEMS_INIT_TASK_CONSTRUCT_FAILED (42)**
The construction of the RTEMS initialization task failed. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_IDLE_THREAD_CREATE_FAILED (43)**
The creation of an IDLE task failed. This fatal error may occur during system initialization. It happens if a task create extension fails for an IDLE task.

**INTERNAL_ERROR_NO_MEMORY_FOR_IDLE_TASK_STORAGE (44)**
There was not enough memory available to allocate an IDLE task stack. This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_IDLE_THREAD_STACK_TOO_SMALL (45)**
The task stack size of an IDLE task would have been less than the configured stack size for IDLE tasks, see *CONFIGURE_IDLE_TASK_STACK_SIZE* (page 708). This fatal error may occur during system initialization. It is an application configuration error.

**INTERNAL_ERROR_CANNOT_DISABLE_DATA_CACHE (46)**
This fatal error may be caused by *rtems_cache_disable_data()* (page 511) if the data cache cannot be disabled for a particular *target* or configuration. The data cache may be necessary to provide *atomic operations*. In SMP configurations, the data cache may be required to ensure data coherency. See the BSP documentation in the *RTEMS User Manual* for more information.

## 23.3 Operations

### 23.3.1 Announcing a Fatal Error

A fatal error can be announced by calling for example *rtems_fatal()* (page 526), *rtems_shutdown_executive()* (page 528), *rtems_panic()* (page 527), *rtems_fatal_error_occurred()* (page 532), or `exit()`.

## 23.4 Directives

This section details the directives of the Fatal Error Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 23.4.1 rtems_fatal()

Invokes the fatal error handler.

CALLING SEQUENCE:

```
void rtems_fatal(
  rtems_fatal_source fatal_source,
  rtems_fatal_code   fatal_code
);
```

PARAMETERS:

**fatal_source**
This parameter is the fatal source.

**fatal_code**
This parameter is the fatal code.

DESCRIPTION:

This directive processes fatal errors. The fatal source is set to the value of the `fatal_source` parameter. The fatal code is set to the value of the `fatal_code` parameter.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not return to the caller.
- The directive invokes the fatal error extensions in *extension forward order*.
- The directive does not invoke handlers registered by `atexit()` or `on_exit()`.
- The directive may terminate the system.

### 23.4.2 rtems_panic()

Prints the message and invokes the fatal error handler.

CALLING SEQUENCE:

```
void rtems_panic( const char *fmt, ... );
```

PARAMETERS:

**fmt**
This parameter is the message format.

**...**
This parameter is a list of optional parameters required by the message format.

DESCRIPTION:

This directive prints a message via *printk()* (page 491) specified by the `fmt` parameter and optional parameters and then invokes the fatal error handler. The fatal source is set to `RTEMS_FATAL_SOURCE_PANIC`. The fatal code is set to the value of the `fmt` parameter value.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not return to the caller.
- The directive invokes the fatal error extensions in *extension forward order*.
- The directive does not invoke handlers registered by `atexit()` or `on_exit()`.
- The directive may terminate the system.

### 23.4.3 rtems_shutdown_executive()

Invokes the fatal error handler.

CALLING SEQUENCE:

```
void rtems_shutdown_executive( uint32_t fatal_code );
```

PARAMETERS:

**fatal_code**
This parameter is the fatal code.

DESCRIPTION:

This directive processes fatal errors. The fatal source is set to `RTEMS_FATAL_SOURCE_EXIT`. The fatal code is set to the value of the `fatal_code` parameter.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not return to the caller.
- The directive invokes the fatal error extensions in *extension forward order*.
- The directive does not invoke handlers registered by `atexit()` or `on_exit()`.
- The directive may terminate the system.

### 23.4.4 rtems_exception_frame_print()

Prints the exception frame.

CALLING SEQUENCE:

```
void rtems_exception_frame_print( const rtems_exception_frame *frame );
```

PARAMETERS:

**frame**
This parameter is the reference to the exception frame to print.

DESCRIPTION:

The exception frame is printed in an architecture-dependent format using *printk()* (page 491).

### 23.4.5 rtems_fatal_source_text()

Returns a descriptive text for the fatal source.

CALLING SEQUENCE:

```
const char *rtems_fatal_source_text( rtems_fatal_source fatal_source );
```

PARAMETERS:

**fatal_source**
This parameter is the fatal source.

RETURN VALUES:

**"?"**
The `fatal_source` parameter value was not a fatal source.

Returns a descriptive text for the fatal source. The text for the fatal source is the enumerator constant name.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.

### 23.4.6 rtems_internal_error_text()

Returns a descriptive text for the internal error code.

CALLING SEQUENCE:

```
const char *rtems_internal_error_text( rtems_fatal_code internal_error_code );
```

PARAMETERS:

**internal_error_code**
This parameter is the internal error code.

RETURN VALUES:

**"?"**
The `internal_error_code` parameter value was not an internal error code.

Returns a descriptive text for the internal error code. The text for the internal error code is the enumerator constant name.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.

### 23.4.7 rtems_fatal_error_occurred()

Invokes the fatal error handler.

CALLING SEQUENCE:

```
void rtems_fatal_error_occurred( uint32_t fatal_code );
```

PARAMETERS:

**fatal_code**
This parameter is the fatal code.

DESCRIPTION:

This directive processes fatal errors. The fatal source is set to `INTERNAL_ERROR_RTEMS_API`. The fatal code is set to the value of the `fatal_code` parameter.

NOTES:

This directive is deprecated and should not be used in new code. It is recommended to not use this directive since error locations cannot be uniquely identified. A recommended alternative directive is *rtems_fatal()* (page 526).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not return to the caller.
- The directive invokes the fatal error extensions in *extension forward order*.
- The directive does not invoke handlers registered by `atexit()` or `on_exit()`.
- The directive may terminate the system.

CHAPTER

**TWENTYFOUR**

# BOARD SUPPORT PACKAGES

## 24.1 Introduction

A board support package (BSP) is a collection of user-provided facilities which interface RTEMS and an application with a specific hardware platform. These facilities may include hardware initialization, device drivers, user extensions, and a Multiprocessor Communications Interface (MPCI). However, a minimal BSP need only support processor reset and initialization and, if needed, a clock tick.

## 24.2 Reset and Initialization

An RTEMS based application is initiated or re-initiated when the processor is reset. This initialization code is responsible for preparing the target platform for the RTEMS application. Although the exact actions performed by the initialization code are highly processor and target dependent, the logical functionality of these actions are similar across a variety of processors and target platforms.

Normally, the BSP and some of the application initialization is intertwined in the RTEMS initialization sequence controlled by the shared function `boot_card()`.

The reset application initialization code is executed first when the processor is reset. All of the hardware must be initialized to a quiescent state by this software before initializing RTEMS. When in quiescent state, devices do not generate any interrupts or require any servicing by the application. Some of the hardware components may be initialized in this code as well as any application initialization that does not involve calls to RTEMS directives.

The processor's Interrupt Vector Table which will be used by the application may need to be set to the required value by the reset application initialization code. Because interrupts are enabled automatically by RTEMS as part of the context switch to the first task, the Interrupt Vector Table MUST be set before this directive is invoked to ensure correct interrupt vectoring. The processor's Interrupt Vector Table must be accessible by RTEMS as it will be modified by the when installing user Interrupt Service Routines (ISRs) On some CPUs, RTEMS installs it's own Interrupt Vector Table as part of initialization and thus these requirements are met automatically. The reset code which is executed before the call to any RTEMS initialization routines has the following requirements:

- Must not make any blocking RTEMS directive calls.
- If the processor supports multiple privilege levels, must leave the processor in the most privileged, or supervisory, state.
- Must allocate a stack of sufficient size to execute the initialization and shutdown of the system. This stack area will NOT be used by any task once the system is initialized. This stack is often reserved via the linker script or in the assembly language start up file.
- Must initialize the stack pointer for the initialization process to that allocated.
- Must initialize the processor's Interrupt Vector Table.
- Must disable all maskable interrupts.
- If the processor supports a separate interrupt stack, must allocate the interrupt stack and initialize the interrupt stack pointer.

At the end of the initialization sequence, RTEMS does not return to the BSP initialization code, but instead context switches to the highest priority task to begin application execution. This task is typically a User Initialization Task which is responsible for performing both local and global application initialization which is dependent on RTEMS facilities. It is also responsible for initializing any higher level RTEMS services the application uses such as networking and blocking device drivers.

### 24.2.1 Interrupt Stack Requirements

The worst-case stack usage by interrupt service routines must be taken into account when designing an application. If the processor supports interrupt nesting, the stack usage must include the deepest nest level. The worst-case stack usage must account for the following requirements:

- Processor's interrupt stack frame
- Processor's subroutine call stack frame
- RTEMS system calls
- Registers saved on stack
- Application subroutine calls

The size of the interrupt stack must be greater than or equal to the confugured minimum stack size.

### 24.2.2 Processors with a Separate Interrupt Stack

Some processors support a separate stack for interrupts. When an interrupt is vectored and the interrupt is not nested, the processor will automatically switch from the current stack to the interrupt stack. The size of this stack is based solely on the worst-case stack usage by interrupt service routines.

The dedicated interrupt stack for the entire application on some architectures is supplied and initialized by the reset and initialization code of the user's Board Support Package. Whether allocated and initialized by the BSP or RTEMS, since all ISRs use this stack, the stack size must take into account the worst case stack usage by any combination of nested ISRs.

### 24.2.3 Processors Without a Separate Interrupt Stack

Some processors do not support a separate stack for interrupts. In this case, without special assistance every task's stack must include enough space to handle the task's worst-case stack usage as well as the worst-case interrupt stack usage. This is necessary because the worst-case interrupt nesting could occur while any task is executing.

On many processors without dedicated hardware managed interrupt stacks, RTEMS manages a dedicated interrupt stack in software. If this capability is supported on a CPU, then it is logically equivalent to the processor supporting a separate interrupt stack in hardware.

## 24.3 Device Drivers

Device drivers consist of control software for special peripheral devices and provide a logical interface for the application developer. The RTEMS I/O manager provides directives which allow applications to access these device drivers in a consistent fashion. A Board Support Package may include device drivers to access the hardware on the target platform. These devices typically include serial and parallel ports, counter/timer peripherals, real-time clocks, disk interfaces, and network controllers.

For more information on device drivers, refer to the I/O Manager chapter.

### 24.3.1 Clock Tick Device Driver

Most RTEMS applications will include a clock tick device driver which invokes a clock tick directive at regular intervals. The clock tick is necessary if the application is to utilize timeslicing, the clock manager, the timer manager, the rate monotonic manager, or the timeout option on blocking directives.

The clock tick is usually provided as an interrupt from a counter/timer or a real-time clock device. When a counter/timer is used to provide the clock tick, the device is typically programmed to operate in continuous mode. This mode selection causes the device to automatically reload the initial count and continue the countdown without programmer intervention. This reduces the overhead required to manipulate the counter/timer in the clock tick ISR and increases the accuracy of tick occurrences. The initial count can be based on the microseconds_per_tick field in the RTEMS Configuration Table. An alternate approach is to set the initial count for a fixed time period (such as one millisecond) and have the ISR invoke a clock tick directive on the configured `microseconds_per_tick` boundaries. Obviously, this can induce some error if the configured `microseconds_per_tick` is not evenly divisible by the chosen clock interrupt quantum.

It is important to note that the interval between clock ticks directly impacts the granularity of RTEMS timing operations. In addition, the frequency of clock ticks is an important factor in the overall level of system overhead. A high clock tick frequency results in less processor time being available for task execution due to the increased number of clock tick ISRs.

## 24.4 User Extensions

RTEMS allows the application developer to augment selected features by invoking user-supplied extension routines when the following system events occur:

- Task creation
- Task initiation
- Task reinitiation
- Task deletion
- Task context switch
- Post task context switch
- Task begin
- Task exits
- Fatal error detection

User extensions can be used to implement a wide variety of functions including execution profiling, non-standard coprocessor support, debug support, and error detection and recovery. For example, the context of a non-standard numeric coprocessor may be maintained via the user extensions. In this example, the task creation and deletion extensions are responsible for allocating and deallocating the context area, the task initiation and reinitiation extensions would be responsible for priming the context area, and the task context switch extension would save and restore the context of the device.

For more information on user extensions, refer to *User Extensions Manager* (page 541).

## 24.5 Multiprocessor Communications Interface (MPCI)

RTEMS requires that an MPCI layer be provided when a multiple node application is developed. This MPCI layer must provide an efficient and reliable communications mechanism between the multiple nodes. Tasks on different nodes communicate and synchronize with one another via the MPCI. Each MPCI layer must be tailored to support the architecture of the target platform.

For more information on the MPCI, refer to the Multiprocessing Manager chapter.

### 24.5.1 Tightly-Coupled Systems

A tightly-coupled system is a multiprocessor configuration in which the processors communicate solely via shared global memory. The MPCI can simply place the RTEMS packets in the shared memory space. The two primary considerations when designing an MPCI for a tightly-coupled system are data consistency and informing another node of a packet.

The data consistency problem may be solved using atomic "test and set" operations to provide a "lock" in the shared memory. It is important to minimize the length of time any particular processor locks a shared data structure.

The problem of informing another node of a packet can be addressed using one of two techniques. The first technique is to use an interprocessor interrupt capability to cause an interrupt on the receiving node. This technique requires that special support hardware be provided by either the processor itself or the target platform. The second technique is to have a node poll for arrival of packets. The drawback to this technique is the overhead associated with polling.

### 24.5.2 Loosely-Coupled Systems

A loosely-coupled system is a multiprocessor configuration in which the processors communicate via some type of communications link which is not shared global memory. The MPCI sends the RTEMS packets across the communications link to the destination node. The characteristics of the communications link vary widely and have a significant impact on the MPCI layer. For example, the bandwidth of the communications link has an obvious impact on the maximum MPCI throughput.

The characteristics of a shared network, such as Ethernet, lend themselves to supporting an MPCI layer. These networks provide both the point-to-point and broadcast capabilities which are expected by RTEMS.

### 24.5.3 Systems with Mixed Coupling

A mixed-coupling system is a multiprocessor configuration in which the processors communicate via both shared memory and communications links. A unique characteristic of mixed-coupling systems is that a node may not have access to all communication methods. There may be multiple shared memory areas and communication links. Therefore, one of the primary functions of the MPCI layer is to efficiently route RTEMS packets between nodes. This routing may be based on numerous algorithms. In addition, the router may provide alternate communications paths in the event of an overload or a partial failure.

### 24.5.4 Heterogeneous Systems

Designing an MPCI layer for a heterogeneous system requires special considerations by the developer. RTEMS is designed to eliminate many of the problems associated with sharing data in a heterogeneous environment. The MPCI layer need only address the representation of thirty-two (32) bit unsigned quantities.

For more information on supporting a heterogeneous system, refer the Supporting Heterogeneous Environments in the Multiprocessing Manager chapter.

CHAPTER

**TWENTYFIVE**

# USER EXTENSIONS MANAGER

## 25.1 Introduction

The User Extensions Manager allows the application developer to augment the executive by allowing them to supply extension routines which are invoked at critical system events. The directives provided by the User Extensions Manager are:

- *rtems_extension_create()* (page 550) - Creates an extension set.
- *rtems_extension_delete()* (page 552) - Deletes the extension set.
- *rtems_extension_ident()* (page 553) - Identifies an extension set by the object name.

## 25.2 Background

User extensions (call-back functions) are invoked by the system when the following events occur

- thread creation,
- thread start,
- thread restart,
- thread switch,
- thread begin,
- thread exitted (return from thread entry function),
- thread termination,
- thread deletion, and
- fatal error detection (system termination).

The user extensions have event-specific arguments, invocation orders and execution contexts. Extension sets can be installed at run-time via rtems_extension_create() (dynamic extension sets) or at link-time via the application configuration option *CONFIGURE_INITIAL_EXTENSIONS* (page 573) (initial extension sets).

The execution context of user extensions varies. Some user extensions are invoked with ownership of the allocator mutex. The allocator mutex protects dynamic memory allocations and object creation/deletion. Some user extensions are invoked with thread dispatching disabled. The fatal error extension is invoked in an arbitrary context.

### 25.2.1 Extension Sets

User extensions are maintained as a set. All user extensions are optional and may be *NULL*. Together a set of these user extensions typically performs a specific functionality such as performance monitoring or debugger support. The extension set is defined via the following structure.

```
typedef struct {
  rtems_task_create_extension    thread_create;
  rtems_task_start_extension     thread_start;
  rtems_task_restart_extension   thread_restart;
  rtems_task_delete_extension    thread_delete;
  rtems_task_switch_extension    thread_switch;
  rtems_task_begin_extension     thread_begin;
  rtems_task_exitted_extension   thread_exitted;
  rtems_fatal_extension          fatal;
  rtems_task_terminate_extension thread_terminate;
} rtems_extensions_table;
```

### 25.2.2 TCB Extension Area

There is no system-provided storage for the initial extension sets.

The task control block (TCB) contains a pointer for each dynamic extension set. The pointer is initialized to *NULL* during thread initialization before the thread create extension is invoked. The pointer may be used by the dynamic extension set to maintain thread-specific data.

The TCB extension is an array of pointers in the TCB. The index into the table can be obtained from the extension identifier returned when the extension object is created:

```
index = rtems_object_id_get_index( extension_id );
```

The number of pointers in the area is the same as the number of dynamic user extension sets configured. This allows an application to augment the TCB with user-defined information. For example, an application could implement task profiling by storing timing statistics in the TCB's extended memory area. When a task context switch is being executed, the thread switch extension could read a real-time clock to calculate how long the task being swapped out has run as well as timestamp the starting time for the task being swapped in.

If used, the extended memory area for the TCB should be allocated and the TCB extension pointer should be set at the time the task is created or started by either the thread create or thread start extension. The application is responsible for managing this extended memory area for the TCBs. The memory may be reinitialized by the thread restart extension and should be deallocated by the thread delete extension when the task is deleted. Since the TCB extension buffers would most likely be of a fixed size, the RTEMS partition manager could be used to manage the application's extended memory area. The application could create a partition of fixed size TCB extension buffers and use the partition manager's allocation and deallocation directives to obtain and release the extension buffers.

### 25.2.3 Order of Invocation

The user extensions are invoked in either *extension forward order* or *extension reverse order*. By invoking the user extensions in these orders, extensions can be built upon one another. At the following system events, the user extensions are invoked in *forward* order

- thread creation,
- thread start,
- thread restart,
- thread switch,
- thread begin,
- thread exitted (return from thread entry function), and
- fatal error detection.

At the following system events, the user extensions are invoked in *reverse* order:

- thread termination, and
- thread deletion.

At these system events, the user extensions are invoked in reverse order to insure that if an extension set is built upon another, the more complicated user extension is invoked before the user extension it is built upon. An example is use of the thread delete extension by the Standard C Library. Extension sets which are installed after the Standard C Library will operate correctly even if they utilize the C Library because the C Library's thread delete extension is invoked after that of the other thread delete extensions.

### 25.2.4 Thread Create Extension

The thread create extension is invoked during thread creation, for example via rtems_task_create() or `pthread_create()`. The thread create extension is defined as follows.

```
typedef bool ( *rtems_task_create_extension )(
  rtems_tcb *executing,
  rtems_tcb *created
);
```

The `executing` is a pointer to the TCB of the currently executing thread. The `created` is a pointer to the TCB of the created thread. The created thread is completely initialized with respect to the operating system.

The executing thread is the owner of the allocator mutex except during creation of the idle threads. Since the allocator mutex allows nesting the normal memory allocation routines can be used.

A thread create extension will frequently attempt to allocate resources. If this allocation fails, then the thread create extension must return `false` and the entire thread create operation will fail, otherwise it must return `true`.

The thread create extension is invoked in forward order with thread dispatching enabled (except during system initialization).

### 25.2.5 Thread Start Extension

The thread start extension is invoked during a thread start, for example via rtems_task_start() or `pthread_create()`. The thread start extension is defined as follows.

```
typedef void ( *rtems_task_start_extension )(
  rtems_tcb *executing,
  rtems_tcb *started
);
```

The `executing` is a pointer to the TCB of the currently executing thread. The `started` is a pointer to the TCB of the started thread. It is invoked after the environment of the started thread has been loaded and the started thread has been made ready. So, in SMP configurations, the thread may already run on another processor before the thread start extension is actually invoked. Thread switch and thread begin extensions may run before or in parallel with the thread start extension in SMP configurations.

The thread start extension is invoked in forward order with thread dispatching disabled.

### 25.2.6 Thread Restart Extension

The thread restart extension is invoked during a thread restart, for example via rtems_task_restart(). The thread restart extension is defined as follows.

```
typedef void ( *rtems_task_restart_extension )(
  rtems_tcb *executing,
  rtems_tcb *restarted
);
```

Both `executing` and `restarted` are pointers the TCB of the currently executing thread. It is invoked in the context of the executing thread right before the execution context is reloaded. The thread stack reflects the previous execution context.

The thread restart extension is invoked in forward order with thread dispatching enabled (except during system initialization). The thread life is protected. Thread restart and delete requests issued by thread restart extensions lead to recursion. The POSIX cleanup handlers, POSIX key destructors and thread-local object destructors run in this context.

### 25.2.7 Thread Switch Extension

The thread switch extension is defined as follows.

```
typedef void ( *rtems_task_switch_extension )(
  rtems_tcb *executing,
  rtems_tcb *heir
);
```

The invocation conditions of the thread switch extension depend on whether RTEMS was configured for uniprocessor or SMP systems. A user must pay attention to the differences to correctly implement a thread switch extension.

In uniprocessor configurations, the thread switch extension is invoked before the context switch from the currently executing thread to the heir thread. The `executing` is a pointer to the TCB of the currently executing thread. The `heir` is a pointer to the TCB of the heir thread. The context switch initiated through the multitasking start is not covered by the thread switch extension.

In SMP configurations, the thread switch extension is invoked after the context switch to the new executing thread (previous heir thread). The `executing` is a pointer to the TCB of the previously executing thread. Despite the name, this is not the currently executing thread. The `heir` is a pointer to the TCB of the newly executing thread. This is the currently executing thread. The context switches initiated through the multitasking start are covered by the thread switch extension. The reason for the differences to uniprocessor configurations is that the context switch may update the heir thread of the processor, see *Thread Dispatch Details* (page 867). The thread switch extensions are invoked with disabled interrupts and with ownership of a per-processor SMP lock. Thread switch extensions may run in parallel on multiple processors. It is recommended to use thread-local or per-processor data structures for thread switch extensions. A global SMP lock should be avoided for performance reasons.

The thread switch extension is invoked in forward order with thread dispatching disabled.

### 25.2.8 Thread Begin Extension

The thread begin extension is invoked during a thread begin before the thread entry function is called. The thread begin extension is defined as follows.

```
typedef void ( *rtems_task_begin_extension )(
  rtems_tcb *executing
);
```

The `executing` is a pointer to the TCB of the currently executing thread. The thread begin extension executes in a normal thread context and may allocate resources for the executing thread. In particular, it has access to thread-local storage of the executing thread.

The thread begin extension is invoked in forward order with thread dispatching enabled. The thread switch extension may be called multiple times for this thread before or during the thread begin extension is invoked.

### 25.2.9 Thread Exitted Extension

The thread exitted extension is invoked once the thread entry function returns. The thread exitted extension is defined as follows.

```
typedef void ( *rtems_task_exitted_extension )(
  rtems_tcb *executing
);
```

The `executing` is a pointer to the TCB of the currently executing thread.

This extension is invoked in forward order with thread dispatching enabled.

### 25.2.10 Thread Termination Extension

The thread termination extension is invoked in case a termination request is recognized by the currently executing thread. Termination requests may result due to calls of rtems_task_delete(), `pthread_exit()`, or `pthread_cancel()`. The thread termination extension is defined as follows.

```
typedef void ( *rtems_task_terminate_extension )(
  rtems_tcb *executing
);
```

The `executing` is a pointer to the TCB of the currently executing thread.

It is invoked in the context of the terminated thread right before the thread dispatch to the heir thread. The POSIX cleanup handlers, POSIX key destructors and thread-local object destructors run in this context. Depending on the order, the thread termination extension has access to thread-local storage and thread-specific data of POSIX keys.

The thread terminate extension is invoked in reverse order with thread dispatching enabled. The thread life is protected. Thread restart and delete requests issued by thread terminate extensions lead to recursion.

### 25.2.11 Thread Delete Extension

The thread delete extension is invoked in case a zombie thread is killed. A thread becomes a zombie thread after it terminated. The thread delete extension is defined as follows.

```
typedef void ( *rtems_task_delete_extension )(
  rtems_tcb *executing,
  rtems_tcb *deleted
);
```

The `executing` is a pointer to the TCB of the currently executing thread. The `deleted` is a pointer to the TCB of the deleted thread. The `executing` and `deleted` pointers are never equal.

The executing thread is the owner of the allocator mutex. Since the allocator mutex allows nesting the normal memory allocation routines can be used.

The thread delete extension is invoked in reverse order with thread dispatching enabled.

Please note that a thread delete extension is not immediately invoked with a call to rtems_task_delete() or similar. The thread must first terminate and this may take some time. The thread delete extension is invoked by rtems_task_create() or similar as a result of a lazy garbage collection of zombie threads.

### 25.2.12 Fatal Error Extension

The fatal error extension is invoked during *system termination* (page 517). The fatal error extension is defined as follows.

```
typedef void( *rtems_fatal_extension )(
  rtems_fatal_source source,
  bool               always_set_to_false,
  rtems_fatal_code   code
);
```

The `source` parameter is the fatal source indicating the subsystem the fatal condition originated in. The `always_set_to_false` parameter is always set to `false` and provided only for backward compatibility reasons. The `code` parameter is the fatal error code. This value must be interpreted with respect to the source.

The fatal error extension is invoked in forward order.

It is strongly advised to use initial extension sets to install a fatal error extension. Usually, the initial extension set of board support package provides a fatal error extension which resets the board. In this case, the dynamic fatal error extensions are not invoked.

## 25.3 Directives

This section details the directives of the User Extensions Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 25.3.1 rtems_extension_create()

Creates an extension set.

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_create(
  rtems_name                    name,
  const rtems_extensions_table *extension_table,
  rtems_id                     *id
);
```

PARAMETERS:

**name**
This parameter is the object name of the extension set.

**extension_table**
This parameter is the table with the extensions to be used by the extension set.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the identifier of the created extension set will be stored in this object.

DESCRIPTION:

This directive creates an extension set which resides on the local node. The extension set has the user-defined object name specified in `name`. The assigned object identifier is returned in `id`. This identifier is used to access the extension set with other extension set related directives.

The extension set is initialized using the extension table specified in `extension_table`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_NAME**
The `name` parameter was invalid.

**RTEMS_INVALID_ADDRESS**
The `extension_table` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_TOO_MANY**
There was no inactive object available to create an extension set. The number of extension sets available to the application is configured through the *CONFIGURE_MAXIMUM_USER_EXTENSIONS* (page 623) application configuration option.

NOTES:

The user-provided extension table is not used after the return of the directive.

Each extension of the extension table is optional and may be NULL. All extensions except the task switch extension of the extension table are atomically and immediately installed. A task switch extension is separately installed after the other extensions. The extensions of the extension table are invoked upon the next system event supporting an extension.

An alternative to dynamically created extension sets are initial extensions, see *CONFIGURE_INITIAL_EXTENSIONS* (page 573). Initial extensions are recommended for extension sets which provide a fatal error extension.

For control and maintenance of the extension set, RTEMS allocates a *ESCB* from the local ESCB free pool and initializes it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of extension sets available to the application is configured through the *CONFIGURE_MAXIMUM_USER_EXTENSIONS* (page 623) application configuration option.

### 25.3.2 rtems_extension_delete()

Deletes the extension set.

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_delete( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the extension set identifier.

DESCRIPTION:

This directive deletes the extension set specified by `id`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ID**
There was no extension set associated with the identifier specified by `id`.

NOTES:

The *ESCB* for the deleted extension set is reclaimed by RTEMS.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.

### 25.3.3 rtems_extension_ident()

Identifies an extension set by the object name.

CALLING SEQUENCE:

```
rtems_status_code rtems_extension_ident( rtems_name name, rtems_id *id );
```

PARAMETERS:

**name**
This parameter is the object name to look up.

**id**
This parameter is the pointer to an *rtems_id* (page 41) object. When the directive call is successful, the object identifier of an object with the specified name will be stored in this object.

DESCRIPTION:

This directive obtains an extension set identifier associated with the extension set name specified in `name`.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `id` parameter was NULL.

**RTEMS_INVALID_NAME**
The `name` parameter was 0.

**RTEMS_INVALID_NAME**
There was no object with the specified name on the local node.

NOTES:

If the extension set name is not unique, then the extension set identifier will match the first extension set with that name in the search order. However, this extension set identifier is not guaranteed to correspond to the desired extension set.

The objects are searched from lowest to the highest index. Only the local node is searched.

The extension set identifier is used with other extension related directives to access the extension set.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive will not cause the calling task to be preempted.

CHAPTER

**TWENTYSIX**

# CONFIGURING A SYSTEM

## 26.1 Introduction

The application configuration information group provides an API to get the configuration of an application.

RTEMS must be configured for an application. This configuration encompasses a variety of information including the length of each clock tick, the maximum number of each information RTEMS object that can be created, the application initialization tasks, the task scheduling algorithm to be used, and the device drivers in the application.

Although this information is contained in data structures that are used by RTEMS at system initialization time, the data structures themselves must not be generated by hand. RTEMS provides a set of macros system which provides a simple standard mechanism to automate the generation of these structures.

The RTEMS header file `<rtems/confdefs.h>` is at the core of the automatic generation of system configuration. It is based on the idea of setting macros which define configuration parameters of interest to the application and defaulting or calculating all others. This variety of macros can automatically produce all of the configuration data required for an RTEMS application. The term `confdefs` is shorthand for a *Configuration Defaults*.

As a general rule, application developers only specify values for the configuration parameters of interest to them. They define what resources or features they require. In most cases, when a parameter is not specified, it defaults to zero (0) instances, a standards compliant value, or disabled as appropriate. For example, by default there will be 256 task priority levels but this can be lowered by the application. This number of priority levels is required to be compliant with the RTEID/ORKID standards upon which the Classic API is based. There are similar cases where the default is selected to be compliant with the POSIX standard.

For each configuration parameter in the configuration tables, the macro corresponding to that field is discussed. The RTEMS Maintainers expect that all systems can be easily configured using the `<rtems/confdefs.h>` mechanism and that using this mechanism will avoid internal RTEMS configuration changes impacting applications.

Some application configuration settings and other system parameters can be queried by the application. The directives provided by the Application Configuration Information are:

- *rtems_get_build_label()* (page 743) - Gets the RTEMS build label.
- *rtems_get_copyright_notice()* (page 744) - Gets the RTEMS copyright notice.
- *rtems_get_target_hash()* (page 745) - Gets the RTEMS target hash.
- *rtems_get_version_string()* (page 746) - Gets the RTEMS version string.
- *rtems_configuration_get_do_zero_of_workspace()* (page 747) - Indicates if the RTEMS Workspace is configured to be zeroed during system initialization for this application.
- *rtems_configuration_get_idle_task_stack_size()* (page 748) - Gets the IDLE task stack size in bytes of this application.
- *rtems_configuration_get_idle_task()* (page 749) - Gets the IDLE task body of this application.
- *rtems_configuration_get_interrupt_stack_size()* (page 750) - Gets the interrupt stack size in bytes of this application.
- *rtems_configuration_get_maximum_barriers()* (page 751) - Gets the resource number of *Barrier Manager* (page 355) objects configured for this application.

- *rtems_configuration_get_maximum_extensions()* (page 752) - Gets the resource number of *User Extensions Manager* (page 541) objects configured for this application.
- *rtems_configuration_get_maximum_message_queues()* (page 753) - Gets the resource number of *Message Manager* (page 365) objects configured for this application.
- *rtems_configuration_get_maximum_partitions()* (page 754) - Gets the resource number of *Partition Manager* (page 415) objects configured for this application.
- *rtems_configuration_get_maximum_periods()* (page 755) - Gets the resource number of *Rate Monotonic Manager* (page 299) objects configured for this application.
- *rtems_configuration_get_maximum_ports()* (page 756) - Gets the resource number of *Dual-Ported Memory Manager* (page 455) objects configured for this application.
- *rtems_configuration_get_maximum_processors()* (page 757) - Gets the maximum number of processors configured for this application.
- *rtems_configuration_get_maximum_regions()* (page 758) - Gets the resource number of *Region Manager* (page 431) objects configured for this application.
- *rtems_configuration_get_maximum_semaphores()* (page 759) - Gets the resource number of *Semaphore Manager* (page 327) objects configured for this application.
- *rtems_configuration_get_maximum_tasks()* (page 760) - Gets the resource number of *Task Manager* (page 97) objects configured for this application.
- *rtems_configuration_get_maximum_timers()* (page 761) - Gets the resource number of *Timer Manager* (page 275) objects configured for this application.
- *rtems_configuration_get_microseconds_per_tick()* (page 762) - Gets the number of microseconds per clock tick configured for this application.
- *rtems_configuration_get_milliseconds_per_tick()* (page 763) - Gets the number of milliseconds per clock tick configured for this application.
- *rtems_configuration_get_nanoseconds_per_tick()* (page 764) - Gets the number of microseconds per clock tick configured for this application.
- *rtems_configuration_get_number_of_initial_extensions()* (page 765) - Gets the number of initial extensions configured for this application.
- *rtems_configuration_get_stack_allocate_for_idle_hook()* (page 766) - Gets the task stack allocator allocate hook used to allocate the stack of each *IDLE task* configured for this application.
- *rtems_configuration_get_stack_allocate_hook()* (page 767) - Gets the task stack allocator allocate hook configured for this application.
- *rtems_configuration_get_stack_allocate_init_hook()* (page 768) - Gets the task stack allocator initialization hook configured for this application.
- *rtems_configuration_get_stack_allocator_avoids_work_space()* (page 769) - Indicates if the task stack allocator is configured to avoid the RTEMS Workspace for this application.
- *rtems_configuration_get_stack_free_hook()* (page 770) - Gets the task stack allocator free hook configured for this application.
- *rtems_configuration_get_stack_space_size()* (page 771) - Gets the configured size in bytes of the memory space used to allocate thread stacks for this application.

- *rtems_configuration_get_ticks_per_timeslice()* (page 772) - Gets the clock ticks per timeslice configured for this application.
- *rtems_configuration_get_unified_work_area()* (page 773) - Indicates if the RTEMS Workspace and C Program Heap are configured to be unified for this application.
- *rtems_configuration_get_user_extension_table()* (page 774) - Gets the initial extensions table configured for this application.
- *rtems_configuration_get_user_multiprocessing_table()* (page 775) - Gets the MPCI configuration table configured for this application.
- *rtems_configuration_get_work_space_size()* (page 776) - Gets the RTEMS Workspace size in bytes configured for this application.
- *rtems_configuration_get_rtems_api_configuration()* (page 777) - Gets the Classic API Configuration Table of this application.
- *rtems_resource_is_unlimited()* (page 778) - Indicates if the resource is unlimited.
- *rtems_resource_maximum_per_allocation()* (page 779) - Gets the maximum number per allocation of a resource number.
- *rtems_resource_unlimited()* (page 780) - Augments the resource number so that it indicates an unlimited resource.

## 26.2 Default Value Selection Philosophy

The user should be aware that the defaults are intentionally set as low as possible. By default, no application resources are configured. The `<rtems/confdefs.h>` file ensures that at least one application task or thread is configured and that at least one of the initialization task/thread tables is configured.

## 26.3 Sizing the RTEMS Workspace

The RTEMS Workspace is a user-specified block of memory reserved for use by RTEMS. The application should NOT modify this memory. This area consists primarily of the RTEMS data structures whose exact size depends upon the values specified in the Configuration Table. In addition, task stacks and floating point context areas are dynamically allocated from the RTEMS Workspace.

The `<rtems/confdefs.h>` mechanism calculates the size of the RTEMS Workspace automatically. It assumes that all tasks are floating point and that all will be allocated the minimum stack space. This calculation includes the amount of memory that will be allocated for internal use by RTEMS. The automatic calculation may underestimate the workspace size truly needed by the application, in which case one can use the *CONFIGURE_MEMORY_OVERHEAD* (page 580) macro to add a value to the estimate. See Specify Memory Overhead for more details.

The memory area for the RTEMS Workspace is determined by the BSP. In case the RTEMS Workspace is too large for the available memory, then a fatal run-time error occurs and the system terminates.

The file `<rtems/confdefs.h>` will calculate the value of the `work_space_size` parameter of the Configuration Table. There are many parameters the application developer can specify to help `<rtems/confdefs.h>` in its calculations. Correctly specifying the application requirements via parameters such as *CONFIGURE_EXTRA_TASK_STACKS* (page 571) and *CONFIGURE_MAXIMUM_TASKS* (page 621) is critical for production software.

For each class of objects, the allocation can operate in one of two ways. The default way has an ceiling on the maximum number of object instances which can concurrently exist in the system. Memory for all instances of that object class is reserved at system initialization. The second way allocates memory for an initial number of objects and increases the current allocation by a fixed increment when required. Both ways allocate space from inside the RTEMS Workspace.

See *Unlimited Objects* (page 564) for more details about the second way, which allows for dynamic allocation of objects and therefore does not provide determinism. This mode is useful mostly for when the number of objects cannot be determined ahead of time or when porting software for which you do not know the object requirements.

The space needed for stacks and for RTEMS objects will vary from one version of RTEMS and from one target processor to another. Therefore it is safest to use `<rtems/confdefs.h>` and specify your application's requirements in terms of the numbers of objects and multiples of `RTEMS_MINIMUM_STACK_SIZE`, as far as is possible. The automatic estimates of space required will in general change when:

- a configuration parameter is changed,
- task or interrupt stack sizes change,
- the floating point attribute of a task changes,
- task floating point attribute is altered,
- RTEMS is upgraded, or
- the target processor is changed.

Failure to provide enough space in the RTEMS Workspace may result in fatal run-time errors terminating the system.

## 26.4 Potential Issues with RTEMS Workspace Size Estimation

The `<rtems/confdefs.h>` file estimates the amount of memory required for the RTEMS Workspace. This estimate is only as accurate as the information given to `<rtems/confdefs.h>` and may be either too high or too low for a variety of reasons. Some of the reasons that `<rtems/confdefs.h>` may reserve too much memory for RTEMS are:

- All tasks/threads are assumed to be floating point.

Conversely, there are many more reasons that the resource estimate could be too low:

- Task/thread stacks greater than minimum size must be accounted for explicitly by developer.
- Memory for messages is not included.
- Device driver requirements are not included.
- Network stack requirements are not included.
- Requirements for add-on libraries are not included.

In general, `<rtems/confdefs.h>` is very accurate when given enough information. However, it is quite easy to use a library and forget to account for its resources.

## 26.5 Configuration Example

In the following example, the configuration information for a system with a single message queue, four (4) tasks, and a timeslice of fifty (50) milliseconds is as follows:

```
#include <bsp.h>
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MICROSECONDS_PER_TICK   1000 /* 1 millisecond */
#define CONFIGURE_TICKS_PER_TIMESLICE       50 /* 50 milliseconds */
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 4
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES 1
#define CONFIGURE_MESSAGE_BUFFER_MEMORY \
           CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE(20, sizeof(struct USER_MESSAGE))
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

In this example, only a few configuration parameters are specified. The impact of these are as follows:

- The example specified *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634) but did not specify any additional parameters. This results in a configuration of an application which will begin execution of a single initialization task named `Init` which is non-preemptible and at priority one (1).
- By specifying *CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER* (page 596), this application is configured to have a clock tick device driver. Without a clock tick device driver, RTEMS has no way to know that time is passing and will be unable to support delays and wall time. Further configuration details about time are provided. Per *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) and *CONFIGURE_TICKS_PER_TIMESLICE* (page 586), the user specified they wanted a clock tick to occur each millisecond, and that the length of a timeslice would be fifty (50) milliseconds.
- By specifying *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597), the application will include a console device driver. Although the console device driver may support a combination of multiple serial ports and display and keyboard combinations, it is only required to provide a single device named `/dev/console`. This device will be used for Standard Input, Output and Error I/O Streams. Thus when *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597) is specified, implicitly three (3) file descriptors are reserved for the Standard I/O Streams and those file descriptors are associated with `/dev/console` during initialization. All console devices are expected to support the POSIX*termios* interface.
- The example above specifies via *CONFIGURE_MAXIMUM_TASKS* (page 621) that the application requires a maximum of four (4) simultaneously existing Classic API tasks. Similarly, by specifying *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615), there may be a maximum of only one (1) concurrently existent Classic API message queues.
- The most surprising configuration parameter in this example is the use of *CONFIGURE_MESSAGE_BUFFER_MEMORY* (page 581). Message buffer memory is allocated from the RTEMS Workspace and must be accounted for. In this example, the single message queue will have up to twenty (20) messages of type `struct USER_MESSAGE`.
- The *CONFIGURE_INIT* (page 572) constant must be defined in order to make `<rtems/`

confdefs.h> instantiate the configuration data structures. This can only be defined in one source file per application that includes <rtems/confdefs.h> or the symbol table will be instantiated multiple times and linking errors produced.

This example illustrates that parameters have default values. Among other things, the application implicitly used the following defaults:

- All unspecified types of communications and synchronization objects in the Classic and POSIX Threads API have maximums of zero (0).
- The filesystem will be the default filesystem which is the In-Memory File System (IMFS).
- The application will have the default number of priority levels.
- The minimum task stack size will be that recommended by RTEMS for the target architecture.

## 26.6 Unlimited Objects

In real-time embedded systems the RAM is normally a limited, critical resource and dynamic allocation is avoided as much as possible to ensure predictable, deterministic execution times. For such cases, see *Sizing the RTEMS Workspace* (page 560) for an overview of how to tune the size of the workspace. Frequently when users are porting software to RTEMS the precise resource requirements of the software is unknown. In these situations users do not need to control the size of the workspace very tightly because they just want to get the new software to run; later they can tune the workspace size as needed.

The following object classes in the Classic API can be configured in unlimited mode:

- Barriers
- Message Queues
- Partitions
- Periods
- Ports
- Regions
- Semaphores
- Tasks
- Timers

Additionally, the following object classes from the POSIX API can be configured in unlimited mode:

- Keys – `pthread_key_create()`
- Key Value Pairs – `pthread_setspecific()`
- Message Queues – `mq_open()`
- Named Semaphores – `sem_open()`
- Shared Memory – `shm_open()`
- Threads – `pthread_create()`
- Timers – `timer_create()`

**Warning**

The following object classes can *not* be configured in unlimited mode:

- Drivers
- File Descriptors
- POSIX Queued Signals
- User Extensions

Due to the memory requirements of unlimited objects it is strongly recommended to use them only in combination with the unified work areas. See Separate or Unified Work Areas for more information on unified work areas.

The following example demonstrates how the two simple configuration defines for unlimited objects and unified works areas can replace many seperate configuration defines for supported object classes:

```
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_UNIFIED_WORK_AREAS
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

Users are cautioned that using unlimited objects is not recommended for production software unless the dynamic growth is absolutely required. It is generally considered a safer embedded systems programming practice to know the system limits rather than experience an out of memory error at an arbitrary and largely unpredictable time in the field.

### 26.6.1 Unlimited Objects by Class

When the number of objects is not known ahead of time, RTEMS provides an auto-extending mode that can be enabled individually for each object type by using the macro *rtems_resource_unlimited()* (page 780). This takes a value as a parameter, and is used to set the object maximum number field in an API Configuration table. The value is an allocation unit size. When RTEMS is required to grow the object table it is grown by this size. The kernel will return the object memory back to the RTEMS Workspace when an object is destroyed. The kernel will only return an allocated block of objects to the RTEMS Workspace if at least half the allocation size of free objects remain allocated. RTEMS always keeps one allocation block of objects allocated. Here is an example of using `rtems_resource_unlimited()`:

```
#define CONFIGURE_MAXIMUM_TASKS rtems_resource_unlimited( 5 )
```

Object maximum specifications can be evaluated with the *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779) macros.

### 26.6.2 Unlimited Objects by Default

To ease the burden of developers who are porting new software RTEMS also provides the capability to make all object classes listed above operate in unlimited mode in a simple manner. The application developer is only responsible for enabling unlimited objects (*CONFIGURE_UNLIMITED_OBJECTS* (page 589)) and specifying the allocation size (*CONFIGURE_UNLIMITED_ALLOCATION_SIZE* (page 588)).

```
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNLIMITED_ALLOCATION_SIZE 5
```

## 26.7 General System Configuration

This section describes general system configuration options.

### 26.7.1 CONFIGURE_DIRTY_MEMORY

CONSTANT:

`CONFIGURE_DIRTY_MEMORY`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the memory areas used for the RTEMS Workspace and the C Program Heap are dirtied with a `0xCF` byte pattern during system initialization.

NOTES:

Dirtying memory can add significantly to system initialization time. It may assist in finding code that incorrectly assumes the contents of free memory areas is cleared to zero during system initialization. In case *CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY* (page 591) is also defined, then the memory is first dirtied and then zeroed.

See also *CONFIGURE_MALLOC_DIRTY* (page 575).

### 26.7.2 CONFIGURE_DISABLE_BSP_SETTINGS

CONSTANT:

`CONFIGURE_DISABLE_BSP_SETTINGS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the optional BSP provided settings listed below are disabled.

The optional BSP provided default values for the following application configuration options are disabled:

- *CONFIGURE_IDLE_TASK_BODY* (page 706)
- *CONFIGURE_IDLE_TASK_STACK_SIZE* (page 708)
- *CONFIGURE_INTERRUPT_STACK_SIZE* (page 574)

The optional BSP provided initial extension set is disabled (see *initial extension sets*). The optional BSP provided prerequisite IO device drivers are disabled (see Device Driver Configuration). The optional BSP provided support for `sbrk()` is disabled.

This configuration option provides an all or nothing choice with respect to the optional BSP provided settings.

### 26.7.3 CONFIGURE_DISABLE_NEWLIB_REENTRANCY

CONSTANT:

`CONFIGURE_DISABLE_NEWLIB_REENTRANCY`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Newlib reentrancy support per thread is disabled and a global reentrancy structure is used.

NOTES:

You can enable this option to reduce the size of the *TCB*. Use this option with care, since it can lead to race conditions and undefined system behaviour. For example, `errno` is no longer a thread-local variable if this option is enabled.

### 26.7.4 CONFIGURE_EXECUTIVE_RAM_SIZE

CONSTANT:

`CONFIGURE_EXECUTIVE_RAM_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

If this configuration option is undefined, then the RTEMS Workspace and task stack space size is calculated by `<rtems/confdefs.h>` based on the values configuration options.

DESCRIPTION:

The value of this configuration option defines the RTEMS Workspace size in bytes.

NOTES:

This is an advanced configuration option. Use it only if you know exactly what you are doing.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINTPTR_MAX.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

### 26.7.5 CONFIGURE_EXTRA_TASK_STACKS

CONSTANT:

`CONFIGURE_EXTRA_TASK_STACKS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the number of bytes the applications wishes to add to the task stack requirements calculated by `<rtems/confdefs.h>`.

NOTES:

This parameter is very important. If the application creates tasks with stacks larger then the minimum, then that memory is **not** accounted for by `<rtems/confdefs.h>`.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.7.6 CONFIGURE_INIT

CONSTANT:

`CONFIGURE_INIT`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

There is no default configuration associated with this configuration option. If `<rtems/confdefs.h>` is included and this configuration option is not defined, then only white space is included.

DESCRIPTION:

While this configuration option is defined, when the `<rtems/confdefs.h>` is included, the system settings defined by present application configuration options are statically allocated and initialized. All user provided application configuration options defined before the include of `<rtems/confdefs.h>` are evaluated. They define the actual system settings.

### 26.7.7 CONFIGURE_INITIAL_EXTENSIONS

CONSTANT:

`CONFIGURE_INITIAL_EXTENSIONS`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is the empty list.

DESCRIPTION:

The value of this configuration option is used to initialize the table of initial user extensions.

NOTES:

The value of this configuration option is placed before the entries of `BSP_INITIAL_EXTENSION` and after the entries of all other initial user extensions.

It is recommended that applications provide a fatal extension to customize the *System Termination Procedure* (page 517).

CONSTRAINTS:

The value of the configuration option shall be a list of initializers for structures of type *rtems_extensions_table* (page 40).

### 26.7.8 CONFIGURE_INTERRUPT_STACK_SIZE

CONSTANT:

`CONFIGURE_INTERRUPT_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

If the *CONFIGURE_DISABLE_BSP_SETTINGS* (page 568) configuration option is not defined and `BSP_INTERRUPT_STACK_SIZE` is provided by the *BSP*, then the default value is defined by `BSP_INTERRUPT_STACK_SIZE`, otherwise the default value is `CPU_STACK_MINIMUM_SIZE`.

DESCRIPTION:

The value of this configuration option defines the size of an interrupt stack in bytes.

NOTES:

There is one interrupt stack available for each configured processor (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)). The interrupt stack areas are statically allocated in a special linker section (`.rtemsstack.interrupt`). The placement of this linker section is BSP-specific.

Some BSPs use the interrupt stack as the initialization stack which is used to perform the sequential system initialization before the multithreading is started.

The interrupt stacks are covered by the stack checker, see *CONFIGURE_STACK_CHECKER_ENABLED* (page 585). However, using a too small interrupt stack size may still result in undefined behaviour.

The interrupt stack size may depend on the interrupt priority settings, see also *rtems_interrupt_set_priority()* (page 202).

In releases before RTEMS 5.1 the default value was *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584) instead of `CPU_STACK_MINIMUM_SIZE`.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to a BSP-specific and application-specific minimum value.
- The value of the configuration option shall be small enough so that the interrupt stack area calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type size_t.
- The value of the configuration option shall be aligned according to `CPU_INTERRUPT_STACK_ALIGNMENT`.

### 26.7.9 CONFIGURE_MALLOC_DIRTY

CONSTANT:

`CONFIGURE_MALLOC_DIRTY`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then each memory area returned by C Program Heap allocator functions such as `malloc()` is dirtied with a `0xCF` byte pattern before it is handed over to the application.

NOTES:

The dirtying performed by this option is carried out for each successful memory allocation from the C Program Heap in contrast to *CONFIGURE_DIRTY_MEMORY* (page 567) which dirties the memory only once during the system initialization.

### 26.7.10 CONFIGURE_MAXIMUM_FILE_DESCRIPTORS

CONSTANT:

`CONFIGURE_MAXIMUM_FILE_DESCRIPTORS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 3.

DESCRIPTION:

The value of this configuration option defines the maximum number of file like objects that can be concurrently open.

NOTES:

The default value of three file descriptors allows RTEMS to support standard input, output, and error I/O streams on `/dev/console`.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

### 26.7.11 CONFIGURE_MAXIMUM_PROCESSORS

CONSTANT:

`CONFIGURE_MAXIMUM_PROCESSORS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 1.

DESCRIPTION:

The value of this configuration option defines the maximum number of processors an application intends to use. The number of actually available processors depends on the hardware and may be less. It is recommended to use the smallest value suitable for the application in order to save memory. Each processor needs an IDLE task stack and interrupt stack for example.

NOTES:

If there are more processors available than configured, the rest will be ignored.

This configuration option is only evaluated in SMP configurations of RTEMS (e.g. RTEMS was built with the SMP build configuration option enabled). In all other configurations it has no effect.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to one.
- The value of the configuration option shall be less than or equal to `CPU_MAXIMUM_PROCESSORS`.

### 26.7.12 CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE

CONSTANT:

`CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

If the value of this configuration option is greater than zero, then it defines the maximum thread-local storage size, otherwise the thread-local storage size is defined by the linker depending on the thread-local storage objects used by the application in the statically-linked executable.

NOTES:

This configuration option can be used to reserve space for the dynamic linking of modules with thread-local storage objects.

If the thread-local storage size defined by the thread-local storage objects used by the application in the statically-linked executable is greater than a non-zero value of this configuration option, then a fatal error will occur during system initialization.

Use `RTEMS_ALIGN_UP()` and `RTEMS_TASK_STORAGE_ALIGNMENT` to adjust the size to meet the minimum alignment requirement of a thread-local storage area.

The actual thread-local storage size is determined when the application executable is linked. The `rtems-exeinfo` command line tool included in the RTEMS Tools can be used to obtain the thread-local storage size and alignment of an application executable.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be an integral multiple of `RTEMS_TASK_STORAGE_ALIGNMENT`.

### 26.7.13 CONFIGURE_MAXIMUM_THREAD_NAME_SIZE

CONSTANT:

`CONFIGURE_MAXIMUM_THREAD_NAME_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 16.

DESCRIPTION:

The value of this configuration option defines the maximum thread name size including the terminating `NUL` character.

NOTES:

The default value was chosen for Linux compatibility, see pthread_setname_np().

The size of the thread control block is increased by the maximum thread name size.

This configuration option is available since RTEMS 5.1.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

### 26.7.14 CONFIGURE_MEMORY_OVERHEAD

CONSTANT:

`CONFIGURE_MEMORY_OVERHEAD`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the number of kilobytes the application wishes to add to the RTEMS Workspace size calculated by `<rtems/confdefs.h>`.

NOTES:

This configuration option should only be used when it is suspected that a bug in `<rtems/confdefs.h>` has resulted in an underestimation. Typically the memory allocation will be too low when an application does not account for all message queue buffers or task stacks, see *CONFIGURE_MESSAGE_BUFFER_MEMORY* (page 581).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.7.15 CONFIGURE_MESSAGE_BUFFER_MEMORY

CONSTANT:

`CONFIGURE_MESSAGE_BUFFER_MEMORY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the number of bytes reserved for message queue buffers in the RTEMS Workspace.

NOTES:

The configuration options *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615) and *CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES* (page 638) define only how many message queues can be created by the application. The memory for the message buffers is configured by this option. For each message queue you have to reserve some memory for the message buffers. The size depends on the maximum number of pending messages and the maximum size of the messages of a message queue. Use the `CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE()` macro to specify the message buffer memory for each message queue and sum them up to define the value for `CONFIGURE_MAXIMUM_MESSAGE_QUEUES`.

The interface for the `CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE()` help macro is as follows:

```
CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE( max_messages, max_msg_size )
```

Where `max_messages` is the maximum number of pending messages and `max_msg_size` is the maximum size in bytes of the messages of the corresponding message queue. Both parameters shall be compile time constants. Not using this help macro (e.g. just using `max_messages * max_msg_size`) may result in an underestimate of the RTEMS Workspace size.

The following example illustrates how the `CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE()` help macro can be used to assist in calculating the message buffer memory required. In this example, there are two message queues used in this application. The first message queue has a maximum of 24 pending messages with the message structure defined by the type `one_message_type`. The other message queue has a maximum of 500 pending messages with the message structure defined by the type `other_message_type`.

```
#define CONFIGURE_MESSAGE_BUFFER_MEMORY ( \
    CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE( \
      24, \
      sizeof( one_message_type ) \
    ) \
    + CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE( \
      500, \
      sizeof( other_message_type ) \
```

(continues on next page)

(continued from previous page)

```
    ) \
  )
```

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.7.16 CONFIGURE_MICROSECONDS_PER_TICK

CONSTANT:

`CONFIGURE_MICROSECONDS_PER_TICK`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 10000.

DESCRIPTION:

The value of this configuration option defines the length of time in microseconds between clock ticks (clock tick quantum).

When the clock tick quantum value is too low, the system will spend so much time processing clock ticks that it does not have processing time available to perform application work. In this case, the system will become unresponsive.

The lowest practical time quantum varies widely based upon the speed of the target hardware and the architectural overhead associated with interrupts. In general terms, you do not want to configure it lower than is needed for the application.

The clock tick quantum should be selected such that it all blocking and delay times in the application are evenly divisible by it. Otherwise, rounding errors will be introduced which may negatively impact the application.

NOTES:

This configuration option has no impact if the Clock Driver is not configured, see *CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER* (page 593).

There may be Clock Driver specific limits on the resolution or maximum value of a clock tick quantum.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to a value defined by the *Clock Driver*.
- The value of the configuration option shall be less than or equal to a value defined by the *Clock Driver*.
- The resulting clock ticks per second should be an integer.

### 26.7.17 CONFIGURE_MINIMUM_TASK_STACK_SIZE

CONSTANT:

`CONFIGURE_MINIMUM_TASK_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is `CPU_STACK_MINIMUM_SIZE`.

DESCRIPTION:

The value of this configuration option defines the minimum stack size in bytes for every user task or thread in the system.

NOTES:

Adjusting this parameter should be done with caution. Examining the actual stack usage using the stack checker usage reporting facility is recommended (see also *CONFIGURE_STACK_CHECKER_ENABLED* (page 585)).

This parameter can be used to lower the minimum from that recommended. This can be used in low memory systems to reduce memory consumption for stacks. However, this shall be done with caution as it could increase the possibility of a blown task stack.

This parameter can be used to increase the minimum from that recommended. This can be used in higher memory systems to reduce the risk of stack overflow without performing analysis on actual consumption.

By default, this configuration parameter defines also the minimum stack size of POSIX threads. This can be changed with the *CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE* (page 644) configuration option.

In releases before RTEMS 5.1 the `CONFIGURE_MINIMUM_TASK_STACK_SIZE` was used to define the default value of *CONFIGURE_INTERRUPT_STACK_SIZE* (page 574).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option shall be greater than or equal to a BSP-specific and application-specific minimum value.

### 26.7.18 CONFIGURE_STACK_CHECKER_ENABLED

CONSTANT:

`CONFIGURE_STACK_CHECKER_ENABLED`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the stack checker is enabled.

NOTES:

The stack checker performs run-time stack bounds checking. This increases the time required to create tasks as well as adding overhead to each context switch.

In 4.9 and older, this configuration option was named `STACK_CHECKER_ON`.

### 26.7.19 CONFIGURE_TICKS_PER_TIMESLICE

CONSTANT:

`CONFIGURE_TICKS_PER_TIMESLICE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 50.

DESCRIPTION:

The value of this configuration option defines the length of the timeslice quantum in ticks for each task.

NOTES:

This configuration option has no impact if the Clock Driver is not configured, see *CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER* (page 593).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to one.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.7.20 CONFIGURE_UNIFIED_WORK_AREAS

CONSTANT:

`CONFIGURE_UNIFIED_WORK_AREAS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then there will be separate memory pools for the RTEMS Workspace and C Program Heap.

DESCRIPTION:

In case this configuration option is defined, then the RTEMS Workspace and the C Program Heap will be one pool of memory.

NOTES:

Having separate pools does have some advantages in the event a task blows a stack or writes outside its memory area. However, in low memory systems the overhead of the two pools plus the potential for unused memory in either pool is very undesirable.

In high memory environments, this is desirable when you want to use the *Unlimited Objects* (page 564) option. You will be able to create objects until you run out of all available memory rather then just until you run out of RTEMS Workspace.

### 26.7.21 CONFIGURE_UNLIMITED_ALLOCATION_SIZE

CONSTANT:

`CONFIGURE_UNLIMITED_ALLOCATION_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 8.

DESCRIPTION:

If *CONFIGURE_UNLIMITED_OBJECTS* (page 589) is defined, then the value of this configuration option defines the default objects maximum of all object classes supporting *Unlimited Objects* (page 564) to `rtems_resource_unlimited( CONFIGURE_UNLIMITED_ALLOCATION_SIZE )`.

NOTES:

By allowing users to declare all resources as being unlimited the user can avoid identifying and limiting the resources used.

The object maximum of each class can be configured also individually using the *rtems_resource_unlimited()* (page 780) macro.

CONSTRAINTS:

The value of the configuration option shall meet the constraints of all object classes to which it is applied.

### 26.7.22 CONFIGURE_UNLIMITED_OBJECTS

CONSTANT:

`CONFIGURE_UNLIMITED_OBJECTS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then unlimited objects are used by default.

NOTES:

When using unlimited objects, it is common practice to also specify *CONFIGURE_UNIFIED_WORK_AREAS* (page 587) so the system operates with a single pool of memory for both RTEMS Workspace and C Program Heap.

This option does not override an explicit configuration for a particular object class by the user.

See also *CONFIGURE_UNLIMITED_ALLOCATION_SIZE* (page 588).

### 26.7.23 CONFIGURE_VERBOSE_SYSTEM_INITIALIZATION

CONSTANT:

`CONFIGURE_VERBOSE_SYSTEM_INITIALIZATION`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the system initialization is verbose.

NOTES:

You may use this feature to debug system initialization issues. The *printk()* (page 491) function is used to print the information.

### 26.7.24 CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY

CONSTANT:

`CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the memory areas used for the RTEMS Workspace and the C Program Heap are zeroed with a `0x00` byte pattern during system initialization.

NOTES:

Zeroing memory can add significantly to the system initialization time. It is not necessary for RTEMS but is often assumed by support libraries. In case *CONFIGURE_DIRTY_MEMORY* (page 567) is also defined, then the memory is first dirtied and then zeroed.

## 26.8 Device Driver Configuration

This section describes configuration options related to the device drivers. Note that network device drivers are not covered by the following options.

### 26.8.1 CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then a Clock Driver may be initialized during system initialization.

DESCRIPTION:

In case this configuration option is defined, then **no** Clock Driver is initialized during system initialization.

NOTES:

This configuration parameter is intended to prevent the common user error of using the Hello World example as the baseline for an application and leaving out a clock tick source.

The application shall define exactly one of the following configuration options

- *CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER* (page 596),
- `CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER`, or
- *CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER* (page 605),

otherwise a compile time error in the configuration file will occur.

### 26.8.2 CONFIGURE_APPLICATION_EXTRA_DRIVERS

CONSTANT:

`CONFIGURE_APPLICATION_EXTRA_DRIVERS`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is the empty list.

DESCRIPTION:

The value of this configuration option is used to initialize the Device Driver Table.

NOTES:

The value of this configuration option is placed after the entries of other device driver configuration options.

See *CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS* (page 608) for an alternative placement of application device driver initializers.

CONSTRAINTS:

The value of the configuration option shall be a list of initializers for structures of type *rtems_driver_address_table* (page 40).

### 26.8.3 CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the ATA Driver is initialized during system initialization.

NOTES:

Most BSPs do not include support for an ATA Driver.

If this option is defined and the BSP does not have this device driver, then the user will get a link time error for an undefined symbol.

### 26.8.4 CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Clock Driver is initialized during system initialization.

NOTES:

The Clock Driver is responsible for providing a regular interrupt which invokes a clock tick directive.

The application shall define exactly one of the following configuration options

- `CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`,
- *CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER* (page 593), or
- *CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER* (page 605),

otherwise a compile time error in the configuration file will occur.

### 26.8.5 CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Console Driver is initialized during system initialization.

NOTES:

The Console Driver is responsible for providing the `/dev/console` device file. This device is used to initialize the standard input, output, and error file descriptors.

BSPs should be constructed in a manner that allows *printk()* (page 491) to work properly without the need for the Console Driver to be configured.

The

- `CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER`,
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER* (page 602), and
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER* (page 603)

configuration options are mutually exclusive.

### 26.8.6 CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Frame Buffer Driver is initialized during system initialization.

NOTES:

Most BSPs do not include support for a Frame Buffer Driver. This is because many boards do not include the required hardware.

If this option is defined and the BSP does not have this device driver, then the user will get a link time error for an undefined symbol.

### 26.8.7 CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the IDE Driver is initialized during system initialization.

NOTES:

Most BSPs do not include support for an IDE Driver.

If this option is defined and the BSP does not have this device driver, then the user will get a link time error for an undefined symbol.

### 26.8.8 CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the `/dev/null` Driver is initialized during system initialization.

NOTES:

This device driver is supported by all BSPs.

### 26.8.9 CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Real-Time Clock Driver is initialized during system initialization.

NOTES:

Most BSPs do not include support for a real-time clock (RTC). This is because many boards do not include the required hardware.

If this is defined and the BSP does not have this device driver, then the user will get a link time error for an undefined symbol.

### 26.8.10 CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Simple Console Driver is initialized during system initialization.

NOTES:

This device driver is responsible for providing the `/dev/console` device file. This device is used to initialize the standard input, output, and error file descriptors.

This device driver reads via *getchark()* (page 494).

This device driver writes via *rtems_putc()* (page 488).

The Termios framework is not used. There is no support to change device settings, e.g. baud, stop bits, parity, etc.

The

- *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597),
- `CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER`, and
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER* (page 603)

configuration options are mutually exclusive.

### 26.8.11 CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Simple Task Console Driver is initialized during system initialization.

NOTES:

This device driver is responsible for providing the `/dev/console` device file. This device is used to initialize the standard input, output, and error file descriptors.

This device driver reads via *getchark()* (page 494).

This device driver writes into a write buffer. The count of characters written into the write buffer is returned. It might be less than the requested count, in case the write buffer is full. The write is non-blocking and may be called from interrupt context. A dedicated task reads from the write buffer and outputs the characters via *rtems_putc()* (page 488). This task runs with the least important priority. The write buffer size is 2047 characters and it is not configurable.

Use `fsync( STDOUT_FILENO )` or `fdatasync( STDOUT_FILENO )` to drain the write buffer.

The Termios framework is not used. There is no support to change device settings, e.g. baud, stop bits, parity, etc.

The

- *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597),
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER* (page 602), and
- `CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER`

configuration options are mutually exclusive.

### 26.8.12 CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Stub Driver is initialized during system initialization.

NOTES:

This device driver simply provides entry points that return successful and is primarily a test fixture. It is supported by all BSPs.

### 26.8.13 CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Benchmark Timer Driver is initialized during system initialization.

NOTES:

The Benchmark Timer Driver is intended for the benchmark tests of the RTEMS Testsuite. Applications should not use this driver.

The application shall define exactly one of the following configuration options

- *CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER* (page 596),
- *CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER* (page 593), or
- `CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER`,

otherwise a compile time error will occur.

### 26.8.14 CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Watchdog Driver is initialized during system initialization.

NOTES:

Most BSPs do not include support for a watchdog device driver. This is because many boards do not include the required hardware.

If this is defined and the BSP does not have this device driver, then the user will get a link time error for an undefined symbol.

### 26.8.15 CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the `/dev/zero` Driver is initialized during system initialization.

NOTES:

This device driver is supported by all BSPs.

### 26.8.16 CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS

CONSTANT:

`CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is the empty list.

DESCRIPTION:

The value of this configuration option is used to initialize the Device Driver Table.

NOTES:

The value of this configuration option is placed after the entries defined by `CONFIGURE_BSP_PREREQUISITE_DRIVERS` and before all other device driver configuration options.

See *CONFIGURE_APPLICATION_EXTRA_DRIVERS* (page 594) for an alternative placement of application device driver initializers.

CONSTRAINTS:

The value of the configuration option shall be a list of initializers for structures of type *rtems_driver_address_table* (page 40).

### 26.8.17 CONFIGURE_ATA_DRIVER_TASK_PRIORITY

CONSTANT:

`CONFIGURE_ATA_DRIVER_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 140.

DESCRIPTION:

The value of this configuration option defines the ATA task priority.

NOTES:

This configuration option is only evaluated if the configuration option *CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER* (page 595) is defined.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

### 26.8.18 CONFIGURE_EXCEPTION_TO_SIGNAL_MAPPING

CONSTANT:

`CONFIGURE_EXCEPTION_TO_SIGNAL_MAPPING`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the machine exception to POSIX signal mapping is configured during system initialization.

NOTES:

This device driver is responsible for setting up a mapping from machine exceptions to POSIX signals so that applications may consume them and alter task execution as necessary.

This is especially useful for applications written in Ada or C++.

### 26.8.19 CONFIGURE_MAXIMUM_DRIVERS

CONSTANT:

`CONFIGURE_MAXIMUM_DRIVERS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

This is computed by default, and is set to the number of statically configured device drivers configured using the following configuration options:

- *CONFIGURE_APPLICATION_EXTRA_DRIVERS* (page 594)
- *CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER* (page 595)
- *CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER* (page 596)
- *CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER* (page 597)
- *CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER* (page 598)
- *CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER* (page 599)
- *CONFIGURE_APPLICATION_NEEDS_LIBBLOCK* (page 685)
- *CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER* (page 600)
- *CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER* (page 601)
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER* (page 602)
- *CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER* (page 603)
- *CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER* (page 604)
- *CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER* (page 605)
- *CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER* (page 606)
- *CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER* (page 607)
- *CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS* (page 608)

If the *CONFIGURE_DISABLE_BSP_SETTINGS* (page 568) configuration option is not defined and the *BSP* provides `CONFIGURE_BSP_PREREQUISITE_DRIVERS`, then the BSP-provided prerequisite device drivers are also taken into account.

DESCRIPTION:

The value of this configuration option defines the number of device drivers.

NOTES:

If the application will dynamically install device drivers, then the configuration option value shall be larger than the number of statically configured device drivers.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be greater than or equal than the number of statically configured device drivers.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

## 26.9 Classic API Configuration

This section describes configuration options related to the Classic API.

### 26.9.1 CONFIGURE_MAXIMUM_BARRIERS

CONSTANT:

`CONFIGURE_MAXIMUM_BARRIERS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Barriers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.2 CONFIGURE_MAXIMUM_MESSAGE_QUEUES

CONSTANT:

`CONFIGURE_MAXIMUM_MESSAGE_QUEUES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Message Queues that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564). You have to account for the memory used to store the messages of each message queue, see *CONFIGURE_MESSAGE_BUFFER_MEMORY* (page 581).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.3 CONFIGURE_MAXIMUM_PARTITIONS

CONSTANT:

`CONFIGURE_MAXIMUM_PARTITIONS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Partitions that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.4 CONFIGURE_MAXIMUM_PERIODS

CONSTANT:

`CONFIGURE_MAXIMUM_PERIODS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Periods that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.5 CONFIGURE_MAXIMUM_PORTS

CONSTANT:

`CONFIGURE_MAXIMUM_PORTS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Ports that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.6 CONFIGURE_MAXIMUM_REGIONS

CONSTANT:

`CONFIGURE_MAXIMUM_REGIONS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Regions that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.7 CONFIGURE_MAXIMUM_SEMAPHORES

CONSTANT:

`CONFIGURE_MAXIMUM_SEMAPHORES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Semaphore that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

In SMP configurations, the size of a Semaphore Control Block depends on the scheduler count (see *Configuration Step 3 - Scheduler Table* (page 727)). The semaphores using the *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30) need a ceiling priority per scheduler.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.8 CONFIGURE_MAXIMUM_TASKS

CONSTANT:

`CONFIGURE_MAXIMUM_TASKS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Tasks that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

The calculations for the required memory in the RTEMS Workspace for tasks assume that each task has a minimum stack size and has floating point support enabled. The configuration option *CONFIGURE_EXTRA_TASK_STACKS* (page 571) is used to specify task stack requirements *above* the minimum size required.

The maximum number of POSIX threads is specified by *CONFIGURE_MAXIMUM_POSIX_THREADS* (page 642).

A future enhancement to `<rtems/confdefs.h>` could be to eliminate the assumption that all tasks have floating point enabled. This would require the addition of a new configuration parameter to specify the number of tasks which enable floating point support.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.9 CONFIGURE_MAXIMUM_TIMERS

CONSTANT:

`CONFIGURE_MAXIMUM_TIMERS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API Timers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.9.10 CONFIGURE_MAXIMUM_USER_EXTENSIONS

CONSTANT:

`CONFIGURE_MAXIMUM_USER_EXTENSIONS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of Classic API User Extensions that can be concurrently active.

NOTES:

This object class cannot be configured in unlimited allocation mode.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

### 26.9.11 CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE

CONSTANT:

`CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the minimum count of Classic API Tasks which are constructed by *rtems_task_construct()* (page 113).

NOTES:

By default, the calculation for the required memory in the RTEMS Workspace for tasks assumes that all Classic API Tasks are created by *rtems_task_create()* (page 109). This configuration option can be used to reduce the required memory for the system-provided task storage areas since tasks constructed by *rtems_task_construct()* (page 113) use a user-provided task storage area.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to *CONFIGURE_MAXIMUM_TASKS* (page 621).

## 26.10 Classic API Initialization Task Configuration

This section describes configuration options related to the Classic API initialization task.

### 26.10.1 CONFIGURE_INIT_TASK_ARGUMENTS

CONSTANT:

`CONFIGURE_INIT_TASK_ARGUMENTS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines task argument of the Classic API initialization task.

CONSTRAINTS:

The value of the configuration option shall be convertible to an integer of type *rtems_task_argument* (page 54).

### 26.10.2 CONFIGURE_INIT_TASK_ATTRIBUTES

CONSTANT:

`CONFIGURE_INIT_TASK_ATTRIBUTES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is `RTEMS_DEFAULT_ATTRIBUTES`.

DESCRIPTION:

The value of this configuration option defines the task attributes of the Classic API initialization task.

CONSTRAINTS:

The value of the configuration option shall be a valid task attribute set.

### 26.10.3 CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE

CONSTANT:

`CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

This configuration option has no default value. If it is not specified, then the Classic API initialization task will be created with the stack size defined by the *CONFIGURE_INIT_TASK_STACK_SIZE* (page 633) configuration option.

DESCRIPTION:

The value of this configuration option defines the task storage size of the Classic API initialization task.

NOTES:

If this configuration option is specified, then

- a task storage area of the specified size is statically allocated by `<rtems/confdefs.h>` for the Classic API initialization task,
- the Classic API initialization task is constructed by *rtems_task_construct()* (page 113) instead of using *rtems_task_create()* (page 109),
- the maximum thread-local storage size defined by *CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE* (page 578) is used for the Classic API initialization task,
- the Classic API initialization task should be accounted for in *CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE* (page 624), and
- the task storage area used for the Classic API initialization task is not reclaimed by the system if the task is deleted.

The

- *CONFIGURE_INIT_TASK_STACK_SIZE* (page 633) and
- `CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE`

configuration options are mutually exclusive.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).
- The value of the configuration option shall be defined using *RTEMS_TASK_STORAGE_SIZE()* (page 144).

### 26.10.4 CONFIGURE_INIT_TASK_ENTRY_POINT

CONSTANT:

`CONFIGURE_INIT_TASK_ENTRY_POINT`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is `Init`.

DESCRIPTION:

The value of this configuration option initializes the entry point of the Classic API initialization task.

NOTES:

The application shall provide the function referenced by this configuration option.

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void ( *entry_point )( rtems_task_argument )`.

### 26.10.5 CONFIGURE_INIT_TASK_INITIAL_MODES

CONSTANT:

`CONFIGURE_INIT_TASK_INITIAL_MODES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

In SMP configurations, the default value is `RTEMS_DEFAULT_MODES` otherwise the default value is `RTEMS_NO_PREEMPT`.

DESCRIPTION:

The value of this configuration option defines the initial execution mode of the Classic API initialization task.

CONSTRAINTS:

The value of the configuration option shall be a valid task mode set.

### 26.10.6 CONFIGURE_INIT_TASK_NAME

CONSTANT:

`CONFIGURE_INIT_TASK_NAME`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is `rtems_build_name( 'U', 'I', '1', ' ' )`.

DESCRIPTION:

The value of this configuration option defines the name of the Classic API initialization task.

NOTES:

Use *rtems_build_name()* (page 902) to define the task name.

CONSTRAINTS:

The value of the configuration option shall be convertible to an integer of type `rtems_name`.

### 26.10.7 CONFIGURE_INIT_TASK_PRIORITY

CONSTANT:

`CONFIGURE_INIT_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 1.

DESCRIPTION:

The value of this configuration option defines the initial priority of the Classic API initialization task.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

### 26.10.8 CONFIGURE_INIT_TASK_STACK_SIZE

CONSTANT:

`CONFIGURE_INIT_TASK_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).

DESCRIPTION:

The value of this configuration option defines the task stack size of the Classic API initialization task.

NOTES:

The

- `CONFIGURE_INIT_TASK_STACK_SIZE` and
- *CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE* (page 628)

configuration options are mutually exclusive.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.10.9 CONFIGURE_RTEMS_INIT_TASKS_TABLE

CONSTANT:

`CONFIGURE_RTEMS_INIT_TASKS_TABLE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then exactly one Classic API initialization task is configured.

NOTES:

The application shall define at least one of the following configuration options

- `CONFIGURE_RTEMS_INIT_TASKS_TABLE`,
- *CONFIGURE_POSIX_INIT_THREAD_TABLE* (page 648), or
- *CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION* (page 707)

otherwise a compile time error in the configuration file will occur.

The Classic API initialization task performs the *Global Construction* (page 92).

## 26.11 POSIX API Configuration

This section describes configuration options related to the POSIX API. Most POSIX API objects are available by default since RTEMS 5.1. The queued signals and timers are only available if RTEMS was built with the enable POSIX build configuration option.

### 26.11.1 CONFIGURE_MAXIMUM_POSIX_KEYS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_KEYS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Keys that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.11.2 CONFIGURE_MAXIMUM_POSIX_KEY_VALUE_PAIRS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_KEY_VALUE_PAIRS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is *CONFIGURE_MAXIMUM_POSIX_KEYS* (page 636) * ( *CONFIGURE_MAXIMUM_TASKS* (page 621) + *CONFIGURE_MAXIMUM_POSIX_THREADS* (page 642) ).

DESCRIPTION:

The value of this configuration option defines the maximum number of key value pairs used by POSIX API Keys that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

A key value pair is created by `pthread_setspecific()` if the value is not NULL, otherwise it is deleted.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.11.3 CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Message Queues that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564). You have to account for the memory used to store the messages of each message queue, see *CONFIGURE_MESSAGE_BUFFER_MEMORY* (page 581).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.11.4 CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Queued Signals that can be concurrently active.

NOTES:

Unlimited objects are not available for queued signals.

Queued signals are only available if RTEMS was built with the POSIX API build configuration option enabled.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option shall be zero if the POSIX API is not enabled (e.g. RTEMS was built without the `RTEMS_POSIX_API = True` build configuration option). Otherwise a compile time error in the configuration file will occur.

### 26.11.5 CONFIGURE_MAXIMUM_POSIX_SEMAPHORES

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_SEMAPHORES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Named Semaphores that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

Named semaphores are created with `sem_open()`. Semaphores initialized with `sem_init()` are not affected by this configuration option since the storage space for these semaphores is user-provided.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.11.6 CONFIGURE_MAXIMUM_POSIX_SHMS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_SHMS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Shared Memory objects that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the RTEMS Workspace size calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.

### 26.11.7 CONFIGURE_MAXIMUM_POSIX_THREADS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_THREADS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Threads that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

This calculations for the required memory in the RTEMS Workspace for threads assume that each thread has a minimum stack size and has floating point support enabled. The configuration option *CONFIGURE_EXTRA_TASK_STACKS* (page 571) is used to specify thread stack requirements **above** the minimum size required.

The maximum number of Classic API Tasks is specified by *CONFIGURE_MAXIMUM_TASKS* (page 621).

All POSIX threads have floating point enabled.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.11.8 CONFIGURE_MAXIMUM_POSIX_TIMERS

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_TIMERS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum number of POSIX API Timers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode, see *Unlimited Objects* (page 564).

Timers are only available if RTEMS was built with the POSIX API build configuration option enabled.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to 65535.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option may be defined through *rtems_resource_unlimited()* (page 780) the enable unlimited objects for the object class, if the value passed to *rtems_resource_unlimited()* (page 780) satisfies all other constraints of the configuration option.
- The value of the configuration option shall be zero if the POSIX API is not enabled (e.g. RTEMS was built without the `RTEMS_POSIX_API = True` build configuration option). Otherwise a compile time error in the configuration file will occur.

### 26.11.9 CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE

CONSTANT:

`CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is two times the value of *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).

DESCRIPTION:

The value of this configuration option defines the minimum stack size in bytes for every POSIX thread in the system.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.
- The value of the configuration option shall be greater than or equal to a BSP-specific and application-specific minimum value.

## 26.12 POSIX Initialization Thread Configuration

This section describes configuration options related to the POSIX initialization thread.

### 26.12.1 CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is `POSIX_Init`.

DESCRIPTION:

The value of this configuration option initializes the entry point of the POSIX API initialization thread.

NOTES:

The application shall provide the function referenced by this configuration option.

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void *( *entry_point )( void * )`.

### 26.12.2 CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is *CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE* (page 644).

DESCRIPTION:

The value of this configuration option defines the thread stack size of the POSIX API initialization thread.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.12.3 CONFIGURE_POSIX_INIT_THREAD_TABLE

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_TABLE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then exactly one POSIX initialization thread is configured.

NOTES:

The application shall define at least one of the following configuration options

- *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634),
- `CONFIGURE_POSIX_INIT_THREAD_TABLE`, or
- *CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION* (page 707)

otherwise a compile time error in the configuration file will occur.

If no Classic API initialization task is configured, then the POSIX API initialization thread performs the *Global Construction* (page 92).

## 26.13 Event Recording Configuration

This section describes configuration options related to the event recording.

### 26.13.1 CONFIGURE_RECORD_EXTENSIONS_ENABLED

CONSTANT:

`CONFIGURE_RECORD_EXTENSIONS_ENABLED`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case

- this configuration option is defined
- and *CONFIGURE_RECORD_PER_PROCESSOR_ITEMS* (page 654) is properly defined,

then the event record extensions are enabled.

NOTES:

The record extensions capture thread create, start, restart, delete, switch, begin, exitted and terminate events.

### 26.13.2 CONFIGURE_RECORD_FATAL_DUMP_BASE64

CONSTANT:

`CONFIGURE_RECORD_FATAL_DUMP_BASE64`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case

- this configuration option is defined
- and *CONFIGURE_RECORD_PER_PROCESSOR_ITEMS* (page 654) is properly defined,
- and *CONFIGURE_RECORD_FATAL_DUMP_BASE64_ZLIB* (page 652) is undefined,

then the event records are dumped in Base64 encoding in a fatal error extension (see *System Termination Procedure* (page 517)).

NOTES:

This extension can be used to produce crash dumps.

### 26.13.3 CONFIGURE_RECORD_FATAL_DUMP_BASE64_ZLIB

CONSTANT:

`CONFIGURE_RECORD_FATAL_DUMP_BASE64_ZLIB`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case

- this configuration option is defined
- and *CONFIGURE_RECORD_PER_PROCESSOR_ITEMS* (page 654) is properly defined,

then the event records are compressed by zlib and dumped in Base64 encoding in a fatal error extension (see *System Termination Procedure* (page 517)).

NOTES:

The zlib compression needs about 512KiB of RAM. This extension can be used to produce crash dumps.

### 26.13.4 CONFIGURE_RECORD_INTERRUPTS_ENABLED

CONSTANT:

`CONFIGURE_RECORD_INTERRUPTS_ENABLED`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case

- this configuration option is defined
- and *CONFIGURE_RECORD_PER_PROCESSOR_ITEMS* (page 654) is properly defined,

then the interrupt event recording is enabled.

NOTES:

The interrupt event recording generates interrupt entry and exit events when interrupt entries are dispatched.

### 26.13.5 CONFIGURE_RECORD_PER_PROCESSOR_ITEMS

CONSTANT:

`CONFIGURE_RECORD_PER_PROCESSOR_ITEMS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the event record item count per processor.

NOTES:

The event record buffers are statically allocated for each configured processor (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)). If the value of this configuration option is zero, then nothing is allocated.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to 16.
- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be a power of two.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

## 26.14 Filesystem Configuration

This section describes configuration options related to filesytems. By default, the In-Memory Filesystem (IMFS) is used as the base filesystem (also known as root filesystem). In order to save some memory for your application, you can disable the filesystem support with the *CONFIGURE_APPLICATION_DISABLE_FILESYSTEM* (page 656) configuration option. Alternatively, you can strip down the features of the base filesystem with the *CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM* (page 683) and *CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM* (page 682) configuration options. These three configuration options are mutually exclusive. They are intended for an advanced application configuration.

Features of the IMFS can be disabled and enabled with the following configuration options:

- *CONFIGURE_IMFS_DISABLE_CHMOD* (page 665)
- *CONFIGURE_IMFS_DISABLE_CHOWN* (page 666)
- *CONFIGURE_IMFS_DISABLE_LINK* (page 667)
- *CONFIGURE_IMFS_DISABLE_MKNOD* (page 668)
- *CONFIGURE_IMFS_DISABLE_MKNOD_FILE* (page 670)
- *CONFIGURE_IMFS_DISABLE_MOUNT* (page 671)
- *CONFIGURE_IMFS_DISABLE_READDIR* (page 672)
- *CONFIGURE_IMFS_DISABLE_READLINK* (page 673)
- *CONFIGURE_IMFS_DISABLE_RENAME* (page 674)
- *CONFIGURE_IMFS_DISABLE_RMNOD* (page 675)
- *CONFIGURE_IMFS_DISABLE_SYMLINK* (page 676)
- *CONFIGURE_IMFS_DISABLE_UNMOUNT* (page 677)
- *CONFIGURE_IMFS_DISABLE_UTIME* (page 678)
- *CONFIGURE_IMFS_ENABLE_MKFIFO* (page 679)

### 26.14.1 CONFIGURE_APPLICATION_DISABLE_FILESYSTEM

CONSTANT:

`CONFIGURE_APPLICATION_DISABLE_FILESYSTEM`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then a base filesystem and the configured filesystems are initialized during system initialization.

DESCRIPTION:

In case this configuration option is defined, then **no base filesystem** is initialized during system initialization and **no filesystems** are configured.

NOTES:

Filesystems shall be initialized to support file descriptor based device drivers and basic input/output functions such as `printf()`. Filesystems can be disabled to reduce the memory footprint of an application.

### 26.14.2 CONFIGURE_FILESYSTEM_ALL

CONSTANT:

`CONFIGURE_FILESYSTEM_ALL`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the following configuration options will be defined as well

- *CONFIGURE_FILESYSTEM_DOSFS* (page 658),
- *CONFIGURE_FILESYSTEM_FTPFS* (page 659),
- *CONFIGURE_FILESYSTEM_IMFS* (page 660),
- *CONFIGURE_FILESYSTEM_JFFS2* (page 661),
- *CONFIGURE_FILESYSTEM_NFS* (page 662),
- *CONFIGURE_FILESYSTEM_RFS* (page 663), and
- *CONFIGURE_FILESYSTEM_TFTPFS* (page 664).

### 26.14.3 CONFIGURE_FILESYSTEM_DOSFS

CONSTANT:

`CONFIGURE_FILESYSTEM_DOSFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the DOS (FAT) filesystem is registered, so that instances of this filesystem can be mounted by the application.

NOTES:

This filesystem requires a Block Device Cache configuration, see *CONFIGURE_APPLICATION_NEEDS_LIBBLOCK* (page 685).

### 26.14.4 CONFIGURE_FILESYSTEM_FTPFS

CONSTANT:

`CONFIGURE_FILESYSTEM_FTPFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the FTP filesystem (FTP client) is registered, so that instances of this filesystem can be mounted by the application.

### 26.14.5 CONFIGURE_FILESYSTEM_IMFS

CONSTANT:

`CONFIGURE_FILESYSTEM_IMFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the In-Memory Filesystem (IMFS) is registered, so that instances of this filesystem can be mounted by the application.

NOTES:

Applications will rarely need this configuration option. This configuration option is intended for test programs. You do not need to define this configuration option for the base filesystem (also known as root filesystem).

### 26.14.6 CONFIGURE_FILESYSTEM_JFFS2

CONSTANT:

`CONFIGURE_FILESYSTEM_JFFS2`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the JFFS2 filesystem is registered, so that instances of this filesystem can be mounted by the application.

### 26.14.7 CONFIGURE_FILESYSTEM_NFS

CONSTANT:

`CONFIGURE_FILESYSTEM_NFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Network Filesystem (NFS) client is registered, so that instances of this filesystem can be mounted by the application.

### 26.14.8 CONFIGURE_FILESYSTEM_RFS

CONSTANT:

`CONFIGURE_FILESYSTEM_RFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the RTEMS Filesystem (RFS) is registered, so that instances of this filesystem can be mounted by the application.

NOTES:

This filesystem requires a Block Device Cache configuration, see *CONFIGURE_APPLICATION_NEEDS_LIBBLOCK* (page 685).

### 26.14.9 CONFIGURE_FILESYSTEM_TFTPFS

CONSTANT:

`CONFIGURE_FILESYSTEM_TFTPFS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the TFTP filesystem (TFTP client) is registered, so that instances of this filesystem can be mounted by the application.

### 26.14.10 CONFIGURE_IMFS_DISABLE_CHMOD

CONSTANT:

`CONFIGURE_IMFS_DISABLE_CHMOD`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports changing the mode of files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support changing the mode of files (no support for `chmod()`).

### 26.14.11 CONFIGURE_IMFS_DISABLE_CHOWN

CONSTANT:

`CONFIGURE_IMFS_DISABLE_CHOWN`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports changing the ownership of files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support changing the ownership of files (no support for `chown()`).

### 26.14.12 CONFIGURE_IMFS_DISABLE_LINK

CONSTANT:

`CONFIGURE_IMFS_DISABLE_LINK`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports hard links.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support hard links (no support for `link()`).

### 26.14.13 CONFIGURE_IMFS_DISABLE_MKNOD

CONSTANT:

`CONFIGURE_IMFS_DISABLE_MKNOD`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports making files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support making files (no support for `mknod()`).

### 26.14.14 CONFIGURE_IMFS_DISABLE_MKNOD_DEVICE

CONSTANT:

`CONFIGURE_IMFS_DISABLE_MKNOD_DEVICE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports making device files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support making device files.

### 26.14.15 CONFIGURE_IMFS_DISABLE_MKNOD_FILE

CONSTANT:

`CONFIGURE_IMFS_DISABLE_MKNOD_FILE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports making regular files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support making regular files.

### 26.14.16 CONFIGURE_IMFS_DISABLE_MOUNT

CONSTANT:

`CONFIGURE_IMFS_DISABLE_MOUNT`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports mounting other filesystems.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support mounting other filesystems (no support for `mount()`).

### 26.14.17 CONFIGURE_IMFS_DISABLE_READDIR

CONSTANT:

`CONFIGURE_IMFS_DISABLE_READDIR`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports reading directories.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support reading directories (no support for `readdir()`). It is still possible to open files in a directory.

### 26.14.18 CONFIGURE_IMFS_DISABLE_READLINK

CONSTANT:

`CONFIGURE_IMFS_DISABLE_READLINK`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports reading symbolic links.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support reading symbolic links (no support for `readlink()`).

### 26.14.19 CONFIGURE_IMFS_DISABLE_RENAME

CONSTANT:

`CONFIGURE_IMFS_DISABLE_RENAME`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports renaming files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support renaming files (no support for `rename()`).

### 26.14.20 CONFIGURE_IMFS_DISABLE_RMNOD

CONSTANT:

`CONFIGURE_IMFS_DISABLE_RMNOD`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports removing files.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support removing files (no support for `rmnod()`).

### 26.14.21 CONFIGURE_IMFS_DISABLE_SYMLINK

CONSTANT:

`CONFIGURE_IMFS_DISABLE_SYMLINK`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports creating symbolic links.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support creating symbolic links (no support for `symlink()`).

### 26.14.22 CONFIGURE_IMFS_DISABLE_UNMOUNT

CONSTANT:

`CONFIGURE_IMFS_DISABLE_UNMOUNT`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports unmounting other filesystems.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support unmounting other filesystems (no support for `unmount()`).

### 26.14.23 CONFIGURE_IMFS_DISABLE_UTIME

CONSTANT:

`CONFIGURE_IMFS_DISABLE_UTIME`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS supports changing file times.

DESCRIPTION:

In case this configuration option is defined, then the root IMFS does not support changing file times (no support for `utime()`).

### 26.14.24 CONFIGURE_IMFS_ENABLE_MKFIFO

CONSTANT:

`CONFIGURE_IMFS_ENABLE_MKFIFO`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the root IMFS does not support making FIFOs (no support for `mkfifo()`).

DESCRIPTION:

In case this configuration option is defined, then the root IMFS supports making FIFOs.

### 26.14.25 CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK

CONSTANT:

`CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 128.

DESCRIPTION:

The value of this configuration option defines the block size for in-memory files managed by the IMFS.

NOTES:

The configured block size has two impacts. The first is the average amount of unused memory in the last block of each file. For example, when the block size is 512, on average one-half of the last block of each file will remain unused and the memory is wasted. In contrast, when the block size is 16, the average unused memory per file is only 8 bytes. However, it requires more allocations for the same size file and thus more overhead per block for the dynamic memory management.

Second, the block size has an impact on the maximum size file that can be stored in the IMFS. With smaller block size, the maximum file size is correspondingly smaller. The following shows the maximum file size possible based on the configured block size:

- when the block size is 16 bytes, the maximum file size is 1,328 bytes.
- when the block size is 32 bytes, the maximum file size is 18,656 bytes.
- when the block size is 64 bytes, the maximum file size is 279,488 bytes.
- when the block size is 128 bytes, the maximum file size is 4,329,344 bytes.
- when the block size is 256 bytes, the maximum file size is 68,173,568 bytes.
- when the block size is 512 bytes, the maximum file size is 1,082,195,456 bytes.

CONSTRAINTS:

The value of the configuration option shall be equal to 16, 32, 64, 128, 256, or 512.

### 26.14.26 CONFIGURE_JFFS2_DELAYED_WRITE_TASK_PRIORITY

CONSTANT:

`CONFIGURE_JFFS2_DELAYED_WRITE_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

The value of this configuration option defines the JFFS2 delayed write task priority.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

### 26.14.27 CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM

CONSTANT:

`CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then an IMFS with a reduced feature set will be the base filesystem (also known as root filesystem).

NOTES:

In case this configuration option is defined, then the following configuration options will be defined as well

- *CONFIGURE_IMFS_DISABLE_CHMOD* (page 665),
- *CONFIGURE_IMFS_DISABLE_CHOWN* (page 666),
- *CONFIGURE_IMFS_DISABLE_LINK* (page 667),
- *CONFIGURE_IMFS_DISABLE_MKNOD_FILE* (page 670),
- *CONFIGURE_IMFS_DISABLE_MOUNT* (page 671),
- *CONFIGURE_IMFS_DISABLE_READDIR* (page 672),
- *CONFIGURE_IMFS_DISABLE_READLINK* (page 673),
- *CONFIGURE_IMFS_DISABLE_RENAME* (page 674),
- *CONFIGURE_IMFS_DISABLE_RMNOD* (page 675),
- *CONFIGURE_IMFS_DISABLE_SYMLINK* (page 676),
- *CONFIGURE_IMFS_DISABLE_UTIME* (page 678), and
- *CONFIGURE_IMFS_DISABLE_UNMOUNT* (page 677).

In addition, a simplified path evaluation is enabled. It allows only a look up of absolute paths.

This configuration of the IMFS is basically a device-only filesystem. It is comparable in functionality to the pseudo-filesystem name space provided before RTEMS release 4.5.0.

### 26.14.28 CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM

CONSTANT:

`CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then an IMFS with a reduced feature set will be the base filesystem (also known as root filesystem).

NOTES:

In case this configuration option is defined, then the following configuration options will be defined as well

- *CONFIGURE_IMFS_DISABLE_CHMOD* (page 665),
- *CONFIGURE_IMFS_DISABLE_CHOWN* (page 666),
- *CONFIGURE_IMFS_DISABLE_LINK* (page 667),
- *CONFIGURE_IMFS_DISABLE_READLINK* (page 673),
- *CONFIGURE_IMFS_DISABLE_RENAME* (page 674),
- *CONFIGURE_IMFS_DISABLE_SYMLINK* (page 676),
- *CONFIGURE_IMFS_DISABLE_UTIME* (page 678), and
- *CONFIGURE_IMFS_DISABLE_UNMOUNT* (page 677).

## 26.15 Block Device Cache Configuration

This section describes configuration options related to the Block Device Cache (bdbuf).

### 26.15.1 CONFIGURE_APPLICATION_NEEDS_LIBBLOCK

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_LIBBLOCK`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Block Device Cache is initialized during system initialization.

NOTES:

Each option of the Block Device Cache (bdbuf) configuration can be explicitly set by the user with the configuration options below. The Block Device Cache is used for example by the RFS and DOSFS filesystems.

### 26.15.2 CONFIGURE_BDBUF_BUFFER_MAX_SIZE

CONSTANT:

`CONFIGURE_BDBUF_BUFFER_MAX_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 4096.

DESCRIPTION:

The value of this configuration option defines the maximum size of a buffer in bytes.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be an integral multiple of *CONFIGURE_BDBUF_BUFFER_MIN_SIZE* (page 687).

### 26.15.3 CONFIGURE_BDBUF_BUFFER_MIN_SIZE

CONSTANT:

`CONFIGURE_BDBUF_BUFFER_MIN_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 512.

DESCRIPTION:

The value of this configuration option defines the minimum size of a buffer in bytes.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.4 CONFIGURE_BDBUF_CACHE_MEMORY_SIZE

CONSTANT:

`CONFIGURE_BDBUF_CACHE_MEMORY_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 32768.

DESCRIPTION:

The value of this configuration option defines the size of the cache memory in bytes.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to SIZE_MAX.

### 26.15.5 CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS

CONSTANT:

`CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the maximum blocks per read-ahead request.

NOTES:

A value of 0 disables the read-ahead task (default). The read-ahead task will issue speculative read transfers if a sequential access pattern is detected. This can improve the performance on some systems.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.6 CONFIGURE_BDBUF_MAX_WRITE_BLOCKS

CONSTANT:

`CONFIGURE_BDBUF_MAX_WRITE_BLOCKS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 16.

DESCRIPTION:

The value of this configuration option defines the maximum blocks per write request.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.7 CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY

CONSTANT:

`CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

The value of this configuration option defines the read-ahead task priority.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

### 26.15.8 CONFIGURE_BDBUF_TASK_STACK_SIZE

CONSTANT:

`CONFIGURE_BDBUF_TASK_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is `RTEMS_MINIMUM_STACK_SIZE`.

DESCRIPTION:

The value of this configuration option defines the task stack size of the Block Device Cache tasks in bytes.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.
- The value of the configuration option shall be small enough so that the task stack space calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type uintptr_t.

### 26.15.9 CONFIGURE_SWAPOUT_BLOCK_HOLD

CONSTANT:

`CONFIGURE_SWAPOUT_BLOCK_HOLD`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 1000.

DESCRIPTION:

The value of this configuration option defines the swapout task maximum block hold time in milliseconds.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.10 CONFIGURE_SWAPOUT_SWAP_PERIOD

CONSTANT:

`CONFIGURE_SWAPOUT_SWAP_PERIOD`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 250.

DESCRIPTION:

The value of this configuration option defines the swapout task swap period in milliseconds.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.11 CONFIGURE_SWAPOUT_TASK_PRIORITY

CONSTANT:

`CONFIGURE_SWAPOUT_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

The value of this configuration option defines the swapout task priority.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

### 26.15.12 CONFIGURE_SWAPOUT_WORKER_TASKS

CONSTANT:

`CONFIGURE_SWAPOUT_WORKER_TASKS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the swapout worker task count.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.15.13 CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY

CONSTANT:

`CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

The value of this configuration option defines the swapout worker task priority.

CONSTRAINTS:

The value of the configuration option shall be a valid Classic API task priority. The set of valid task priorities depends on the scheduler configuration.

## 26.16 Task Stack Allocator Configuration

This section describes configuration options related to the task stack allocator. RTEMS allows the application or BSP to define its own allocation and deallocation methods for task stacks. This can be used to place task stacks in special areas of memory or to utilize a Memory Management Unit so that stack overflows are detected in hardware.

### 26.16.1 CONFIGURE_TASK_STACK_ALLOCATOR

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is `_Workspace_Allocate`, which indicates that task stacks will be allocated from the RTEMS Workspace.

DESCRIPTION:

The value of this configuration option initializes the stack allocator allocate handler.

NOTES:

A correctly configured system shall configure the following to be consistent:

- *CONFIGURE_TASK_STACK_ALLOCATOR_INIT* (page 702)
- `CONFIGURE_TASK_STACK_ALLOCATOR`
- *CONFIGURE_TASK_STACK_DEALLOCATOR* (page 703)

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void *( *allocate )( size_t )`.

### 26.16.2 CONFIGURE_TASK_STACK_ALLOCATOR_AVOIDS_WORK_SPACE

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR_AVOIDS_WORK_SPACE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the system is informed that the task stack allocator does not use the RTEMS Workspace.

NOTES:

This configuration option may be used if a custom task stack allocator is configured, see *CONFIGURE_TASK_STACK_ALLOCATOR* (page 699).

### 26.16.3 CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

By default, the IDLE task storage area will be allocated from the RTEMS Workspace.

DESCRIPTION:

The value of this configuration option is the address for the stack allocator allocate handler used to allocate the task storage area of each *IDLE task*.

NOTES:

This configuration option is independent of the other thread stack allocator configuration options. It is assumed that any memory allocated for the task storage area of an *IDLE task* will not be from the RTEMS Workspace.

The IDLE task stack allocator may increase the size of the allocated memory area to account for the actually allocated memory area.

The

- *CONFIGURE_IDLE_TASK_STORAGE_SIZE* (page 709), and
- `CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE`

configuration options are mutually exclusive.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be defined to a valid function pointer of the type `void *( *allocate )( uint32_t, size_t * )`.
- The IDLE task stack allocator shall return a pointer to the allocated memory area or terminate the system with a fatal error if the allocation request cannot be satisfied.
- The IDLE task stack allocator may increase the size of the allocated memory area.

### 26.16.4 CONFIGURE_TASK_STACK_ALLOCATOR_INIT

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR_INIT`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is NULL.

DESCRIPTION:

The value of this configuration option initializes the stack allocator initialization handler.

NOTES:

A correctly configured system shall configure the following to be consistent:

- `CONFIGURE_TASK_STACK_ALLOCATOR_INIT`
- *CONFIGURE_TASK_STACK_ALLOCATOR* (page 699)
- *CONFIGURE_TASK_STACK_DEALLOCATOR* (page 703)

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void ( *initialize )( size_t )` or to NULL.

### 26.16.5 CONFIGURE_TASK_STACK_DEALLOCATOR

CONSTANT:

`CONFIGURE_TASK_STACK_DEALLOCATOR`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is `_Workspace_Free`, which indicates that task stacks will be allocated from the RTEMS Workspace.

DESCRIPTION:

The value of this configuration option initializes the stack allocator deallocate handler.

NOTES:

A correctly configured system shall configure the following to be consistent:

- *CONFIGURE_TASK_STACK_ALLOCATOR_INIT* (page 702)
- *CONFIGURE_TASK_STACK_ALLOCATOR* (page 699)
- `CONFIGURE_TASK_STACK_DEALLOCATOR`

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void ( *deallocate )( void * )`.

### 26.16.6 CONFIGURE_TASK_STACK_FROM_ALLOCATOR

CONSTANT:

`CONFIGURE_TASK_STACK_FROM_ALLOCATOR`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is a macro which supports the system heap allocator.

DESCRIPTION:

The value of this configuration option is used to calculate the task stack space size.

NOTES:

This configuration option may be used if a custom task stack allocator is configured, see *CONFIGURE_TASK_STACK_ALLOCATOR* (page 699).

CONSTRAINTS:

The value of the configuration option shall be defined to a macro which accepts exactly one parameter and returns an unsigned integer. The parameter will be an allocation size and the macro shall return this size plus the overhead of the allocator to manage an allocation request for this size.

## 26.17 Idle Task Configuration

This section describes configuration options related to the idle tasks.

### 26.17.1 CONFIGURE_IDLE_TASK_BODY

CONSTANT:

`CONFIGURE_IDLE_TASK_BODY`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

If the *CONFIGURE_DISABLE_BSP_SETTINGS* (page 568) configuration option is not defined and `BSP_IDLE_TASK_BODY` is provided by the *BSP*, then the default value is defined by `BSP_IDLE_TASK_BODY`, otherwise the default value is `_CPU_Thread_Idle_body`.

DESCRIPTION:

The value of this configuration option initializes the IDLE thread body.

NOTES:

IDLE threads shall not block. A blocking IDLE thread results in undefined system behaviour because the scheduler assume that at least one ready thread exists.

IDLE threads can be used to initialize the application, see configuration option *CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION* (page 707).

The BSP may have knowledge of the specific CPU model, system controller logic, and peripheral buses, so a BSP-specific IDLE task may be capable of turning components off to save power during extended periods of no task activity.

CONSTRAINTS:

The value of the configuration option shall be defined to a valid function pointer of the type `void *( *idle_body )( uintptr_t )`.

### 26.17.2 CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION

CONSTANT:

`CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the user is assumed to provide one or more initialization tasks.

DESCRIPTION:

This configuration option is defined to indicate that the user has configured **no** user initialization tasks or threads and that the user provided IDLE task will perform application initialization and then transform itself into an IDLE task.

NOTES:

If you use this option be careful, the user IDLE task **cannot** block at all during the initialization sequence. Further, once application initialization is complete, it shall make itself preemptible and enter an idle body loop.

The IDLE task shall run at the lowest priority of all tasks in the system.

If this configuration option is defined, then it is mandatory to configure a user IDLE task with the *CONFIGURE_IDLE_TASK_BODY* (page 706) configuration option, otherwise a compile time error in the configuration file will occur.

The application shall define at least one of the following configuration options

- *CONFIGURE_RTEMS_INIT_TASKS_TABLE* (page 634),
- *CONFIGURE_POSIX_INIT_THREAD_TABLE* (page 648), or
- `CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION`

otherwise a compile time error in the configuration file will occur.

If no Classic API initialization task and no POSIX API initialization thread is configured, then no *Global Construction* (page 92) is performed.

### 26.17.3 CONFIGURE_IDLE_TASK_STACK_SIZE

CONSTANT:

`CONFIGURE_IDLE_TASK_STACK_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

If the *CONFIGURE_DISABLE_BSP_SETTINGS* (page 568) configuration option is not defined and `BSP_IDLE_TASK_STACK_SIZE` is provided by the *BSP*, then the default value is defined by `BSP_IDLE_TASK_STACK_SIZE`, otherwise the default value is defined by the *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584) configuration option.

DESCRIPTION:

The value of this configuration option defines the task stack size for an IDLE task.

NOTES:

In SMP configurations, there is one IDLE task per configured processor, see *CONFIGURE_MAXIMUM_PROCESSORS* (page 577).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to a BSP-specific and application-specific minimum value.
- The value of the configuration option shall be small enough so that the IDLE task stack area calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type size_t.

### 26.17.4 CONFIGURE_IDLE_TASK_STORAGE_SIZE

CONSTANT:

`CONFIGURE_IDLE_TASK_STORAGE_SIZE`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

This configuration option has no default value. If it is not specified, then the task storage area for each *IDLE task* will allocated from the RTEMS Workspace or through a custom IDLE task stack allocator.

DESCRIPTION:

If this configuration option is specified, then the task storage areas for the *IDLE tasks* are statically allocated by `<rtems/confdefs.h>`. The value of this configuration option defines the size in bytes of the task storage area of each IDLE task in the system.

NOTES:

By default, the IDLE task storage areas are allocated from the RTEMS Workspace. Applications which do not want to use a heap allocator can use this configuration option to use statically allocated memory for the IDLE task storage areas. The task storage area contains the task stack, the thread-local storage, and the floating-point context on architectures with a separate floating-point context. The size of the thread-local storage area is defined at link time or by the *CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE* (page 578) configuration option. You have to estimate the actual thread-local storage size if you want to use this configuration option. If the IDLE task stack size would be less than the value defined by the *CONFIGURE_IDLE_TASK_STACK_SIZE* (page 708) configuration option, for example because the thread-local storage size is larger than expected, then the system terminates with the *INTERNAL_ERROR_CORE* (page 518) fatal source and the *INTERNAL_ERROR_IDLE_THREAD_STACK_TOO_SMALL* (page 519) fatal code during system initialization.

The value of this configuration option is passed to *RTEMS_TASK_STORAGE_SIZE()* (page 144) by `<rtems/confdefs.h>` to determine the actual size of the statically allocated area to take architecture-specific overheads into account.

The

- `CONFIGURE_IDLE_TASK_STORAGE_SIZE`, and
- *CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE* (page 701)

configuration options are mutually exclusive.

CONSTRAINTS:

The value of the configuration option shall be greater than or equal to *CONFIGURE_IDLE_TASK_STACK_SIZE* (page 708).

## 26.18 General Scheduler Configuration

This section describes configuration options related to selecting a scheduling algorithm for an application. A scheduler configuration is optional and only necessary in very specific circumstances. A normal application configuration does not need any of the configuration options described in this section.

By default, the *Deterministic Priority Scheduler* (page 68) algorithm is used in uniprocessor configurations. In case SMP is enabled and the configured maximum processors (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)) is greater than one, then the *Earliest Deadline First SMP Scheduler* (page 70) is selected as the default scheduler algorithm.

For the schedulers provided by RTEMS (see *Scheduling Concepts* (page 61)), the configuration is straightforward. All that is required is to define the configuration option which specifies which scheduler you want for in your application.

The pluggable scheduler interface also enables the user to provide their own scheduling algorithm. If you choose to do this, you must define multiple configuration option.

### 26.18.1 CONFIGURE_CBS_MAXIMUM_SERVERS

CONSTANT:

`CONFIGURE_CBS_MAXIMUM_SERVERS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is *CONFIGURE_MAXIMUM_TASKS* (page 621).

DESCRIPTION:

The value of this configuration option defines the maximum number Constant Bandwidth Servers that can be concurrently active.

NOTES:

This configuration option is only evaluated if the configuration option *CONFIGURE_SCHEDULER_CBS* (page 714) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to SIZE_MAX.
- The value of the configuration option shall be less than or equal to a BSP-specific and application-specific value which depends on the size of the memory available to the application.

### 26.18.2 CONFIGURE_MAXIMUM_PRIORITY

CONSTANT:

`CONFIGURE_MAXIMUM_PRIORITY`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 255.

DESCRIPTION:

For the following schedulers

- *Deterministic Priority Scheduler* (page 68), which is the default in uniprocessor configurations and can be configured through the *CONFIGURE_SCHEDULER_PRIORITY* (page 718) configuration option,
- *Deterministic Priority SMP Scheduler* (page 70) which can be configured through the *CONFIGURE_SCHEDULER_PRIORITY_SMP* (page 720) configuration option, and
- *Arbitrary Processor Affinity Priority SMP Scheduler* (page 70) which can be configured through the *CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP* (page 719) configuration option

this configuration option specifies the maximum numeric priority of any task for these schedulers and one less that the number of priority levels for these schedulers. For all other schedulers provided by RTEMS, this configuration option has no effect.

NOTES:

The numerically greatest priority is the logically lowest priority in the system and will thus be used by the IDLE task.

Priority zero is reserved for internal use by RTEMS and is not available to applications.

Reducing the number of priorities through this configuration option reduces the amount of memory allocated by the schedulers listed above. These schedulers use a chain control structure per priority and this structure consists of three pointers. On a 32-bit architecture, the allocated memory is 12 bytes * (`CONFIGURE_MAXIMUM_PRIORITY + 1`), e.g. 3072 bytes for 256 priority levels (default), 48 bytes for 4 priority levels (`CONFIGURE_MAXIMUM_PRIORITY == 3`).

The default value is 255, because RTEMS shall support 256 priority levels to be compliant with various standards. These priorities range from 0 to 255.

CONSTRAINTS:

The value of the configuration option shall be equal to 3, 7, 31, 63, 127, or 255.

### 26.18.3 CONFIGURE_SCHEDULER_ASSIGNMENTS

CONSTANT:

`CONFIGURE_SCHEDULER_ASSIGNMENTS`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value of this configuration option is computed so that the default scheduler is assigned to each configured processor (up to 32).

DESCRIPTION:

The value of this configuration option is used to initialize the initial scheduler to processor assignments.

NOTES:

Where the system was built with SMP support enabled, this configuration option is evaluated, otherwise it is ignored.

This is an advanced configuration option, see *Clustered Scheduler Configuration* (page 726).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be a list of the following macros:
  - `RTEMS_SCHEDULER_ASSIGN( scheduler_index, attributes )`
  - `RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER`

  The `scheduler_index` macro parameter shall be a valid index of the scheduler table defined by the *CONFIGURE_SCHEDULER_TABLE_ENTRIES* (page 724) configuration option.

  The `attributes` macro parameter shall be set to exactly one of the following constants:
  - `RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY`
  - `RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL`
- The value of the configuration option shall be a list of exactly *CONFIGURE_MAXIMUM_PROCESSORS* (page 577) elements.

### 26.18.4 CONFIGURE_SCHEDULER_CBS

CONSTANT:

`CONFIGURE_SCHEDULER_CBS`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Constant Bandwidth Server Scheduling (CBS)* (page 69) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for exactly one processor.

### 26.18.5 CONFIGURE_SCHEDULER_EDF

CONSTANT:

`CONFIGURE_SCHEDULER_EDF`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Earliest Deadline First Scheduler* (page 68) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for exactly one processor.

### 26.18.6 CONFIGURE_SCHEDULER_EDF_SMP

CONSTANT:

`CONFIGURE_SCHEDULER_EDF_SMP`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Earliest Deadline First SMP Scheduler* (page 70) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

This scheduler algorithm is only available when RTEMS is built with SMP support enabled.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for up to 32 processors.

This scheduler algorithm is the default in SMP configurations if *CONFIGURE_MAXIMUM_PROCESSORS* (page 577) is greater than one.

### 26.18.7 CONFIGURE_SCHEDULER_NAME

CONSTANT:

`CONFIGURE_SCHEDULER_NAME`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is

- `"MEDF"` for the *Earliest Deadline First SMP Scheduler* (page 70),
- `"MPA "` for the *Arbitrary Processor Affinity Priority SMP Scheduler* (page 70),
- `"MPD "` for the *Deterministic Priority SMP Scheduler* (page 70),
- `"MPS "` for the *Simple Priority SMP Scheduler* (page 70),
- `"UCBS"` for the *Constant Bandwidth Server Scheduling (CBS)* (page 69),
- `"UEDF"` for the *Earliest Deadline First Scheduler* (page 68),
- `"UPD "` for the *Deterministic Priority Scheduler* (page 68), and
- `"UPS "` for the *Simple Priority Scheduler* (page 68).

DESCRIPTION:

The value of this configuration option defines the name of the default scheduler.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

Schedulers can be identified via *rtems_scheduler_ident()* (page 72).

Use *rtems_build_name()* (page 902) to define the scheduler name.

CONSTRAINTS:

The value of the configuration option shall be convertible to an integer of type `rtems_name`.

### 26.18.8 CONFIGURE_SCHEDULER_PRIORITY

CONSTANT:

`CONFIGURE_SCHEDULER_PRIORITY`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Deterministic Priority Scheduler* (page 68) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for exactly one processor.

This scheduler algorithm is the default when *CONFIGURE_MAXIMUM_PROCESSORS* (page 577) is exactly one.

The memory allocated for this scheduler depends on the *CONFIGURE_MAXIMUM_PRIORITY* (page 712) configuration option.

### 26.18.9 CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP

CONSTANT:

`CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Arbitrary Processor Affinity Priority SMP Scheduler* (page 70) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

This scheduler algorithm is only available when RTEMS is built with SMP support enabled.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for up to 32 processors.

The memory allocated for this scheduler depends on the *CONFIGURE_MAXIMUM_PRIORITY* (page 712) configuration option.

### 26.18.10 CONFIGURE_SCHEDULER_PRIORITY_SMP

CONSTANT:

`CONFIGURE_SCHEDULER_PRIORITY_SMP`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Deterministic Priority SMP Scheduler* (page 70) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

This scheduler algorithm is only available when RTEMS is built with SMP support enabled.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for up to 32 processors.

The memory allocated for this scheduler depends on the *CONFIGURE_MAXIMUM_PRIORITY* (page 712) configuration option.

### 26.18.11 CONFIGURE_SCHEDULER_SIMPLE

CONSTANT:

`CONFIGURE_SCHEDULER_SIMPLE`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Simple Priority Scheduler* (page 68) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for exactly one processor.

### 26.18.12 CONFIGURE_SCHEDULER_SIMPLE_SMP

CONSTANT:

`CONFIGURE_SCHEDULER_SIMPLE_SMP`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the *Simple Priority SMP Scheduler* (page 70) algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

This scheduler algorithm is only available when RTEMS is built with SMP support enabled.

In case no explicit *Clustered Scheduler Configuration* (page 726) is present, then it is used as the scheduler for up to 32 processors.

### 26.18.13 CONFIGURE_SCHEDULER_STRONG_APA

CONSTANT:

`CONFIGURE_SCHEDULER_STRONG_APA`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the Strong APA algorithm is made available to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

This scheduler algorithm is only available when RTEMS is built with SMP support enabled.

This scheduler algorithm is not correctly implemented. Do not use it.

### 26.18.14 CONFIGURE_SCHEDULER_TABLE_ENTRIES

CONSTANT:

`CONFIGURE_SCHEDULER_TABLE_ENTRIES`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value of this configuration option is the definition of exactly one table entry for the configured scheduler.

DESCRIPTION:

The value of this configuration option is used to initialize the table of configured schedulers.

NOTES:

Schedulers registered in the scheduler table by this configuration option are available to the application. The scheduler table entry index defines the index of the scheduler.

This is an advanced configuration option, see *Clustered Scheduler Configuration* (page 726).

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be a list of the following macros:
  - `RTEMS_SCHEDULER_TABLE_CBS( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_EDF( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_EDF_SMP( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_PRIORITY_AFFINITY_SMP( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_PRIORITY( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_PRIORITY_SMP( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_SIMPLE( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_SIMPLE_SMP( name, obj_name )`
  - `RTEMS_SCHEDULER_TABLE_STRONG_APA( name, obj_name )`

  The `name` macro parameter shall be the name associated with the scheduler data structures, see *Clustered Scheduler Configuration* (page 726).

  The `obj_name` macro parameter shall be the scheduler object name. It is recommended to define the scheduler object name through *rtems_build_name()* (page 902).
- Where the system was build with SMP support enabled, the table shall have one or more entries, otherwise it shall have exactly one entry.

### 26.18.15 CONFIGURE_SCHEDULER_USER

CONSTANT:

`CONFIGURE_SCHEDULER_USER`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

In case this configuration option is defined, then the user shall provide a scheduler algorithm to the application.

NOTES:

This scheduler configuration option is an advanced configuration option. Think twice before you use it.

RTEMS allows the application to provide its own task/thread scheduling algorithm. In order to do this, one shall define `CONFIGURE_SCHEDULER_USER` to indicate the application provides its own scheduling algorithm. If `CONFIGURE_SCHEDULER_USER` is defined then the following additional macros shall be defined:

- `CONFIGURE_SCHEDULER` shall be defined to a static definition of the scheduler data structures of the user scheduler.
- `CONFIGURE_SCHEDULER_TABLE_ENTRIES` shall be defined to a scheduler table entry initializer for the user scheduler.
- `CONFIGURE_SCHEDULER_USER_PER_THREAD` shall be defined to the type of the per-thread information of the user scheduler.

At this time, the mechanics and requirements for writing a new scheduler are evolving and not fully documented. It is recommended that you look at the existing Deterministic Priority Scheduler in `cpukit/score/src/schedulerpriority*.c` for guidance. For guidance on the configuration macros, please examine `cpukit/include/rtems/confdefs/scheduler.h` for how these are defined for the Deterministic Priority Scheduler.

## 26.19 Clustered Scheduler Configuration

This section describes configuration options related to clustered scheduling. A clustered scheduler configuration is optional. It is an advanced configuration area and only necessary in specific circumstances.

Clustered scheduling helps to control the worst-case latencies in a multiprocessor system (SMP). The goal is to reduce the amount of shared state in the system and thus prevention of lock contention. Modern multiprocessor systems tend to have several layers of data and instruction caches. With clustered scheduling it is possible to honour the cache topology of a system and thus avoid expensive cache synchronization traffic.

We have clustered scheduling in case the set of processors of a system is partitioned into non-empty pairwise-disjoint subsets. These subsets are called clusters. Clusters with a cardinality of one are partitions. Each cluster is owned by exactly one scheduler.

In order to use clustered scheduling the application designer has to answer two questions.

1. How is the set of processors partitioned into clusters?
2. Which scheduler algorithm is used for which cluster?

The schedulers are statically configured.

### 26.19.1 Configuration Step 1 - Scheduler Algorithms

Firstly, the application must select which scheduling algorithms are available with the following defines

- *CONFIGURE_SCHEDULER_EDF_SMP* (page 716),
- *CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP* (page 719),
- *CONFIGURE_SCHEDULER_PRIORITY_SMP* (page 720), and
- *CONFIGURE_SCHEDULER_SIMPLE_SMP* (page 722).

This is necessary to calculate the per-thread overhead introduced by the scheduler algorithms. After these definitions the configuration file must `#include <rtems/scheduler.h>` to have access to scheduler-specific configuration macros.

It is possible to make more than one scheduler algorithm available to the application. For example a *Simple Priority SMP Scheduler* (page 70) could be used in a partition for low latency tasks in addition to an *EDF SMP Scheduler* (page 70) for a general-purpose cluster. Since the per-thread overhead depends on the scheduler algorithm only the scheduler algorithms used by the application should be configured.

### 26.19.2 Configuration Step 2 - Schedulers

Each scheduler needs some data structures. Use the following macros to create the scheduler data structures for a particular scheduler identified in the configuration by `name`.

- `RTEMS_SCHEDULER_EDF_SMP(name)`,
- `RTEMS_SCHEDULER_PRIORITY_AFFINITY_SMP(name, prio_count)`,
- `RTEMS_SCHEDULER_PRIORITY_SMP(name, prio_count)`, and
- `RTEMS_SCHEDULER_SIMPLE_SMP(name)`.

The name parameter is used as part of a designator for scheduler-specific data structures, so the usual C/C++ designator rules apply. This name is not the scheduler object name. Additional parameters are scheduler-specific.

### 26.19.3 Configuration Step 3 - Scheduler Table

The schedulers are registered in the system via the scheduler table. To populate the scheduler table define CONFIGURE_SCHEDULER_TABLE_ENTRIES to a list of the following scheduler table entry initializers

- RTEMS_SCHEDULER_TABLE_EDF_SMP(name, obj_name),
- RTEMS_SCHEDULER_TABLE_PRIORITY_AFFINITY_SMP(name, obj_name),
- RTEMS_SCHEDULER_TABLE_PRIORITY_SMP(name, obj_name), and
- RTEMS_SCHEDULER_TABLE_SIMPLE_SMP(name, obj_name).

The name parameter must correspond to the parameter defining the scheduler data structures of configuration step 2. The obj_name determines the scheduler object name and can be used in rtems_scheduler_ident() to get the scheduler object identifier. The scheduler index is defined by the index of the scheduler table. It is a configuration error to add a scheduler multiple times to the scheduler table.

### 26.19.4 Configuration Step 4 - Processor to Scheduler Assignment

The last step is to define which processor uses which scheduler. For this purpose a scheduler assignment table must be defined. The entry count of this table must be equal to the configured maximum processors (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)). A processor assignment to a scheduler can be optional or mandatory. The boot processor must have a scheduler assigned. In case the system needs more mandatory processors than available then a fatal run-time error will occur. To specify the scheduler assignments define CONFIGURE_SCHEDULER_ASSIGNMENTS to a list of

- RTEMS_SCHEDULER_ASSIGN(scheduler_index, attr) and
- RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER

macros. The scheduler_index parameter must be a valid index into the scheduler table defined by configuration step 3. The attr parameter defines the scheduler assignment attributes. By default, a scheduler assignment to a processor is optional. For the scheduler assignment attribute use one of the mutually exclusive variants

- RTEMS_SCHEDULER_ASSIGN_DEFAULT,
- RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY, and
- RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL.

It is possible to add/remove processors to/from schedulers at run-time, see rtems_scheduler_add_processor() and rtems_scheduler_remove_processor().

### 26.19.5 Configuration Example

The following example shows a scheduler configuration for a hypothetical product using two chip variants. One variant has four processors which is used for the normal product line and another provides eight processors for the high-performance product line. The first processor performs hard-real time control of actuators and sensors. The second processor is not used by

RTEMS at all and runs a Linux instance to provide a graphical user interface. The additional processors are used for a worker thread pool to perform data processing operations.

The processors managed by RTEMS use two Deterministic Priority SMP schedulers capable of dealing with 256 priority levels. The scheduler with index zero has the name "IO ". The scheduler with index one has the name "WORK". The scheduler assignments of the first, third and fourth processor are mandatory, so the system must have at least four processors, otherwise a fatal run-time error will occur during system startup. The processor assignments for the fifth up to the eighth processor are optional so that the same application can be used for the normal and high-performance product lines. The second processor has no scheduler assigned and runs Linux. A hypervisor will ensure that the two systems cannot interfere in an undesirable way.

```
#define CONFIGURE_MAXIMUM_PROCESSORS 8
#define CONFIGURE_MAXIMUM_PRIORITY 255

/* Configuration Step 1 - Scheduler Algorithms */
#define CONFIGURE_SCHEDULER_PRIORITY_SMP
#include <rtems/scheduler.h>

/* Configuration Step 2 - Schedulers */
RTEMS_SCHEDULER_PRIORITY_SMP(io, CONFIGURE_MAXIMUM_PRIORITY + 1);
RTEMS_SCHEDULER_PRIORITY_SMP(work, CONFIGURE_MAXIMUM_PRIORITY + 1);

/* Configuration Step 3 - Scheduler Table */
#define CONFIGURE_SCHEDULER_TABLE_ENTRIES \
  RTEMS_SCHEDULER_TABLE_PRIORITY_SMP( \
    io, \
     rtems_build_name('I', 'O', ' ', ' ') \
  ), \
  RTEMS_SCHEDULER_TABLE_PRIORITY_SMP( \
    work, \
    rtems_build_name('W', 'O', 'R', 'K') \
  )

/* Configuration Step 4 - Processor to Scheduler Assignment */
#define CONFIGURE_SCHEDULER_ASSIGNMENTS \
  RTEMS_SCHEDULER_ASSIGN(0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
  RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER, \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
  RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL)
```

### 26.19.6 Configuration Errors

In case one of the scheduler indices in CONFIGURE_SCHEDULER_ASSIGNMENTS is invalid a link-time error will occur with an undefined reference to RTEMS_SCHEDULER_INVALID_INDEX.

Some fatal errors may occur in case of scheduler configuration inconsistencies or a lack of processors on the system. The fatal source is RTEMS_FATAL_SOURCE_SMP.

- `SMP_FATAL_BOOT_PROCESSOR_NOT_ASSIGNED_TO_SCHEDULER` - the boot processor must have a scheduler assigned.
- `SMP_FATAL_MANDATORY_PROCESSOR_NOT_PRESENT` - there exists a mandatory processor beyond the range of physically or virtually available processors. The processor demand must be reduced for this system.
- `SMP_FATAL_START_OF_MANDATORY_PROCESSOR_FAILED` - the start of a mandatory processor failed during system initialization. The system may not have this processor at all or it could be a problem with a boot loader for example. Check the `CONFIGURE_SCHEDULER_ASSIGNMENTS` definition.
- `SMP_FATAL_MULTITASKING_START_ON_UNASSIGNED_PROCESSOR` - it is not allowed to start multitasking on a processor with no scheduler assigned.

## 26.20 FACE Technical Standard Related Configuration

This section describes configuration options related to adapting RTEMS behavior to be aligned with the FACE Technical Standard. The FACE Technical Standard is a product of the FACE Consortium which operates under the Open Group. The FACE Consortium was founded by avionics organizations to improve the portability of cockpit software across various platforms. It addresses technical and business concerns.

Most important from an RTEMS perspective, the FACE Technical Standard defines four POSIX profiles: Security, Safety Base, Safety Extended, and the General Purpose Profile. Each has an increasingly larger subset of POSIX APIs. In the Security and Safety profiles, ARINC 653 is required. It is optional in the General Purpose Profile.

The RTEMS Project has been tracking alignment with the FACE POSIX profiles and they are included in the "RTEMS POSIX 1003.1 Compliance Guide."

### 26.20.1 CONFIGURE_POSIX_TIMERS_FACE_BEHAVIOR

CONSTANT:

`CONFIGURE_POSIX_TIMERS_FACE_BEHAVIOR`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the described feature is not enabled.

DESCRIPTION:

If this configuration option is defined, then POSIX timers may not be created to use the *CLOCK_REALTIME*. Per POSIX, this is allowed behavior but per the FACE Technical Standard, it is not. Using POSIX timers based on CLOCK_REALTIME (e.g., time of day) is unsafe for real-time safety systems as setting CLOCK_REALTIME will perturb any active timers.

If this option is not defined, POSIX timers may be created to use the CLOCK_REALTIME in compliance with the POSIX specification.

## 26.21 Multiprocessing Configuration

This section describes multiprocessing related configuration options. The options are only used if RTEMS was built when the multiprocessing build configuration option is enabled. The multiprocessing configuration is distinct from the SMP configuration. Additionally, this class of configuration options are only applicable if the configuration option *CONFIGURE_MP_APPLICATION* (page 734) is defined. The multiprocessing (MPCI) support must not be confused with the SMP support.

### 26.21.1 CONFIGURE_EXTRA_MPCI_RECEIVE_SERVER_STACK

CONSTANT:

`CONFIGURE_EXTRA_MPCI_RECEIVE_SERVER_STACK`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

The value of this configuration option defines the number of bytes the applications wishes to add to the MPCI task stack on top of *CONFIGURE_MINIMUM_TASK_STACK_SIZE* (page 584).

NOTES:

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.
- The value of the configuration option shall be small enough so that the MPCI receive server stack area calculation carried out by `<rtems/confdefs.h>` does not overflow an integer of type size_t.

### 26.21.2 CONFIGURE_MP_APPLICATION

CONSTANT:

`CONFIGURE_MP_APPLICATION`

OPTION TYPE:

This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:

If this configuration option is undefined, then the multiprocessing services are not initialized.

DESCRIPTION:

This configuration option is defined to indicate that the application intends to be part of a multiprocessing configuration. Additional configuration options are assumed to be provided.

NOTES:

This configuration option shall be undefined if the multiprocessing support is not enabled (e.g. RTEMS was built without the multiprocessing build configuration option enabled). Otherwise a compile time error in the configuration file will occur.

### 26.21.3 CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS

CONSTANT:

`CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 32.

DESCRIPTION:

The value of this configuration option defines the maximum number of concurrently active global objects in a multiprocessor system.

NOTES:

This value corresponds to the total number of objects which can be created with the `RTEMS_GLOBAL` attribute.

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.21.4 CONFIGURE_MP_MAXIMUM_NODES

CONSTANT:

`CONFIGURE_MP_MAXIMUM_NODES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 2.

DESCRIPTION:

The value of this configuration option defines the maximum number of nodes in a multiprocessor system.

NOTES:

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.21.5 CONFIGURE_MP_MAXIMUM_PROXIES

CONSTANT:

`CONFIGURE_MP_MAXIMUM_PROXIES`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is 32.

DESCRIPTION:

The value of this configuration option defines the maximum number of concurrently active thread/task proxies on this node in a multiprocessor system.

NOTES:

Since a proxy is used to represent a remote task/thread which is blocking on this node. This configuration parameter reflects the maximum number of remote tasks/threads which can be blocked on objects on this node, see *Proxies* (page 847).

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

### 26.21.6 CONFIGURE_MP_MPCI_TABLE_POINTER

CONSTANT:

`CONFIGURE_MP_MPCI_TABLE_POINTER`

OPTION TYPE:

This configuration option is an initializer define.

DEFAULT VALUE:

The default value is `&MPCI_table`.

DESCRIPTION:

The value of this configuration option initializes the MPCI Configuration Table.

NOTES:

RTEMS provides a Shared Memory MPCI Device Driver which can be used on any Multiprocessor System assuming the BSP provides the proper set of supporting methods.

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The value of the configuration option shall be a pointer to `rtems_mpci_table`.

### 26.21.7 CONFIGURE_MP_NODE_NUMBER

CONSTANT:

`CONFIGURE_MP_NODE_NUMBER`

OPTION TYPE:

This configuration option is an integer define.

DEFAULT VALUE:

The default value is `NODE_NUMBER`.

DESCRIPTION:

The value of this configuration option defines the node number of this node in a multiprocessor system.

NOTES:

In the RTEMS Multiprocessing Test Suite, the node number is derived from the Makefile variable `NODE_NUMBER`. The same code is compiled with the `NODE_NUMBER` set to different values. The test programs behave differently based upon their node number.

This configuration option is only evaluated if *CONFIGURE_MP_APPLICATION* (page 734) is defined.

CONSTRAINTS:

The following constraints apply to this configuration option:

- The value of the configuration option shall be greater than or equal to zero.
- The value of the configuration option shall be less than or equal to UINT32_MAX.

## 26.22 PCI Library Configuration

This section defines the system configuration parameters supported by `rtems/confdefs.h` related to configuring the PCI Library for RTEMS.

The PCI Library startup behaviour can be configured in four different ways depending on how `CONFIGURE_PCI_CONFIG_LIB` is defined:

**PCI_LIB_AUTO**
Used to enable the PCI auto configuration software. PCI will be automatically probed, PCI buses enumerated, all devices and bridges will be initialized using Plug & Play software routines. The PCI device tree will be populated based on the PCI devices found in the system, PCI devices will be configured by allocating address region resources automatically in PCI space according to the BSP or host bridge driver set up.

**PCI_LIB_READ**
Used to enable the PCI read configuration software. The current PCI configuration is read to create the RAM representation (the PCI device tree) of the PCI devices present. PCI devices are assumed to already have been initialized and PCI buses enumerated, it is therefore required that a BIOS or a boot loader has set up configuration space prior to booting into RTEMS.

**PCI_LIB_STATIC**
Used to enable the PCI static configuration software. The user provides a PCI tree with information how all PCI devices are to be configured at compile time by linking in a custom `struct pci_bus pci_hb` tree. The static PCI library will not probe PCI for devices, instead it will assume that all devices defined by the user are present, it will enumerate the PCI buses and configure all PCI devices in static configuration accordingly. Since probe and allocation software is not needed the startup is faster, has smaller footprint and does not require dynamic memory allocation.

**PCI_LIB_PERIPHERAL**
Used to enable the PCI peripheral configuration. It is similar to `PCI_LIB_STATIC`, but it will never write the configuration to the PCI devices since PCI peripherals are not allowed to access PCI configuration space.

Note that selecting `PCI_LIB_STATIC` or `PCI_LIB_PERIPHERAL` but not defining `pci_hb` will reuslt in link errors. Note also that in these modes Plug & Play is not performed.

## 26.23 Ada Configuration

The GNU Ada runtime library (libgnarl) uses threads, mutexes, condition variables, and signals from the pthreads API. It uses also thread-local storage for the Ada Task Control Block (ATCB). From these resources only the threads need to be accounted for in the configuration. You should include the Ada tasks in your setting of the *CONFIGURE_MAXIMUM_POSIX_THREADS* (page 642) configuration option.

## 26.24 Directives

This section details the directives of the Application Configuration Information. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 26.24.1 rtems_get_build_label()

Gets the RTEMS build label.

CALLING SEQUENCE:

```
const char *rtems_get_build_label( void );
```

DESCRIPTION:

The build label is a user-provided string defined by the build configuration through the `RTEMS_BUILD_LABEL` build option. The format of the string is completely user-defined.

RETURN VALUES:

Returns a pointer to the RTEMS build label.

NOTES:

The build label can be used to distinguish test suite results obtained from different build configurations. A use case is to record test results with performance data to track performance regressions. For this a database of performance limits is required. The build label and the target hash obtained from *rtems_get_target_hash()* (page 745) can be used as a key to obtain performance limits.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.2 rtems_get_copyright_notice()

Gets the RTEMS copyright notice.

CALLING SEQUENCE:

```
const char *rtems_get_copyright_notice( void );
```

RETURN VALUES:

Returns a pointer to the RTEMS copyright notice.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.3 rtems_get_target_hash()

Gets the RTEMS target hash.

CALLING SEQUENCE:

```
const char *rtems_get_target_hash( void );
```

DESCRIPTION:

The target hash is calculated from BSP-specific values which characterize a target system. The target hash is encoded as a base64url string. The target hash algorithm is unspecified.

RETURN VALUES:

Returns a pointer to the RTEMS target hash.

NOTES:

For example, the device tree, settings of the memory controller, processor and bus frequencies, a serial number of a chip may be used to calculate the target hash.

The target hash can be used to distinguish test suite results obtained from different target systems. See also *rtems_get_build_label()* (page 743).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.4 rtems_get_version_string()

Gets the RTEMS version string.

CALLING SEQUENCE:

```
const char *rtems_get_version_string( void );
```

RETURN VALUES:

Returns a pointer to the RTEMS version string.

NOTES:

The version string has no particular format. Parsing the string may break across RTEMS releases.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.5 rtems_configuration_get_do_zero_of_workspace()

Indicates if the RTEMS Workspace is configured to be zeroed during system initialization for this application.

CALLING SEQUENCE:

```
bool rtems_configuration_get_do_zero_of_workspace( void );
```

RETURN VALUES:

Returns true, if the RTEMS Workspace is configured to be zeroed during system initialization for this application, otherwise false.

NOTES:

The setting is defined by the *CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY* (page 591) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.6 rtems_configuration_get_idle_task_stack_size()

Gets the IDLE task stack size in bytes of this application.

CALLING SEQUENCE:

```
size_t rtems_configuration_get_idle_task_stack_size( void );
```

RETURN VALUES:

Returns the IDLE task stack size in bytes of this application.

NOTES:

The IDLE task stack size is defined by the *CONFIGURE_IDLE_TASK_STACK_SIZE* (page 708) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.7 rtems_configuration_get_idle_task()

Gets the IDLE task body of this application.

CALLING SEQUENCE:

```
void *( * )( uintptr_t ) rtems_configuration_get_idle_task( void );
```

RETURN VALUES:

Returns the IDLE task body of this application.

NOTES:

The IDLE task body is defined by the *CONFIGURE_IDLE_TASK_BODY* (page 706) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.8 rtems_configuration_get_interrupt_stack_size()

Gets the interrupt stack size in bytes of this application.

CALLING SEQUENCE:

```
size_t rtems_configuration_get_interrupt_stack_size( void );
```

RETURN VALUES:

Returns the interrupt stack size in bytes of this application.

NOTES:

The interrupt stack size is defined by the *CONFIGURE_INTERRUPT_STACK_SIZE* (page 574) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.9 rtems_configuration_get_maximum_barriers()

Gets the resource number of *Barrier Manager* (page 355) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_barriers( void );
```

RETURN VALUES:

Returns the resource number of *Barrier Manager* (page 355) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_BARRIERS* (page 614) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.10 rtems_configuration_get_maximum_extensions()

Gets the resource number of *User Extensions Manager* (page 541) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_extensions( void );
```

RETURN VALUES:

Returns the resource number of *User Extensions Manager* (page 541) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_USER_EXTENSIONS* (page 623) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.11 rtems_configuration_get_maximum_message_queues()

Gets the resource number of *Message Manager* (page 365) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_message_queues( void );
```

RETURN VALUES:

Returns the resource number of *Message Manager* (page 365) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_MESSAGE_QUEUES* (page 615) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.12 rtems_configuration_get_maximum_partitions()

Gets the resource number of *Partition Manager* (page 415) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_partitions( void );
```

RETURN VALUES:

Returns the resource number of *Partition Manager* (page 415) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_PARTITIONS* (page 616) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.13 rtems_configuration_get_maximum_periods()

Gets the resource number of *Rate Monotonic Manager* (page 299) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_periods( void );
```

RETURN VALUES:

Returns the resource number of *Rate Monotonic Manager* (page 299) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_PERIODS* (page 617) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.14 rtems_configuration_get_maximum_ports()

Gets the resource number of *Dual-Ported Memory Manager* (page 455) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_ports( void );
```

RETURN VALUES:

Returns the resource number of *Dual-Ported Memory Manager* (page 455) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_PORTS* (page 618) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.15 rtems_configuration_get_maximum_processors()

Gets the maximum number of processors configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_processors( void );
```

RETURN VALUES:

Returns the maximum number of processors configured for this application.

NOTES:

The actual number of processors available to the application is returned by *rtems_scheduler_get_processor_maximum()* (page 80) which less than or equal to the configured maximum number of processors (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)).

In uniprocessor configurations, this macro is a compile time constant which evaluates to one.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.16 rtems_configuration_get_maximum_regions()

Gets the resource number of *Region Manager* (page 431) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_regions( void );
```

RETURN VALUES:

Returns the resource number of *Region Manager* (page 431) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_REGIONS* (page 619) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.17 rtems_configuration_get_maximum_semaphores()

Gets the resource number of *Semaphore Manager* (page 327) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_semaphores( void );
```

RETURN VALUES:

Returns the resource number of *Semaphore Manager* (page 327) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_SEMAPHORES* (page 620) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.18 rtems_configuration_get_maximum_tasks()

Gets the resource number of *Task Manager* (page 97) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_tasks( void );
```

RETURN VALUES:

Returns the resource number of *Task Manager* (page 97) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_TASKS* (page 621) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.19 rtems_configuration_get_maximum_timers()

Gets the resource number of *Timer Manager* (page 275) objects configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_maximum_timers( void );
```

RETURN VALUES:

Returns the resource number of *Timer Manager* (page 275) objects configured for this application.

NOTES:

The resource number is defined by the *CONFIGURE_MAXIMUM_TIMERS* (page 622) application configuration option. See also *rtems_resource_is_unlimited()* (page 778) and *rtems_resource_maximum_per_allocation()* (page 779).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.20 rtems_configuration_get_microseconds_per_tick()

Gets the number of microseconds per clock tick configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_microseconds_per_tick( void );
```

RETURN VALUES:

Returns the number of microseconds per clock tick configured for this application.

NOTES:

The number of microseconds per *clock tick* is defined by the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.21 rtems_configuration_get_milliseconds_per_tick()

Gets the number of milliseconds per clock tick configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_milliseconds_per_tick( void );
```

RETURN VALUES:

Returns the number of milliseconds per clock tick configured for this application.

NOTES:

The number of milliseconds per *clock tick* is defined by the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.22 rtems_configuration_get_nanoseconds_per_tick()

Gets the number of microseconds per clock tick configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_nanoseconds_per_tick( void );
```

RETURN VALUES:

Returns the number of microseconds per clock tick configured for this application.

NOTES:

The number of nanoseconds per *clock tick* is defined by the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.23 rtems_configuration_get_number_of_initial_extensions()

Gets the number of initial extensions configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_number_of_initial_extensions( void );
```

RETURN VALUES:

Returns the number of initial extensions configured for this application.

NOTES:

The number of initial extensions is defined by the *CONFIGURE_INITIAL_EXTENSIONS* (page 573) application configuration option and related options.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.24 rtems_configuration_get_stack_allocate_for_idle_hook()

Gets the task stack allocator allocate hook used to allocate the stack of each *IDLE task* configured for this application.

CALLING SEQUENCE:

```
void *( * )( uint32_t, size_t * )
rtems_configuration_get_stack_allocate_for_idle_hook( void );
```

RETURN VALUES:

Returns the task stack allocator allocate hook used to allocate the stack of each *IDLE task* configured for this application.

NOTES:

The task stack allocator allocate hook for idle tasks is defined by the *CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE* (page 701) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.25 rtems_configuration_get_stack_allocate_hook()

Gets the task stack allocator allocate hook configured for this application.

CALLING SEQUENCE:

```
void *( * )( size_t ) rtems_configuration_get_stack_allocate_hook( void );
```

RETURN VALUES:

Returns the task stack allocator allocate hook configured for this application.

NOTES:

The task stack allocator allocate hook is defined by the *CONFIGURE_TASK_STACK_ALLOCATOR* (page 699) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.26 rtems_configuration_get_stack_allocate_init_hook()

Gets the task stack allocator initialization hook configured for this application.

CALLING SEQUENCE:

```
void ( * )( size_t ) rtems_configuration_get_stack_allocate_init_hook( void );
```

RETURN VALUES:

Returns the task stack allocator initialization hook configured for this application.

NOTES:

The task stack allocator initialization hook is defined by the *CONFIGURE_TASK_STACK_ALLOCATOR_INIT* (page 702) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.27 rtems_configuration_get_stack_allocator_avoids_work_space()

Indicates if the task stack allocator is configured to avoid the RTEMS Workspace for this application.

CALLING SEQUENCE:

```
bool rtems_configuration_get_stack_allocator_avoids_work_space( void );
```

RETURN VALUES:

Returns true, if the task stack allocator is configured to avoid the RTEMS Workspace for this application, otherwise false.

NOTES:

The setting is defined by the *CONFIGURE_TASK_STACK_ALLOCATOR_AVOIDS_WORK_SPACE* (page 700) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.28 rtems_configuration_get_stack_free_hook()

Gets the task stack allocator free hook configured for this application.

CALLING SEQUENCE:

```
void ( * )( void * ) rtems_configuration_get_stack_free_hook( void );
```

RETURN VALUES:

Returns the task stack allocator free hook configured for this application.

NOTES:

The task stack allocator free hook is defined by the *CONFIGURE_TASK_STACK_DEALLOCATOR* (page 703) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.29 rtems_configuration_get_stack_space_size()

Gets the configured size in bytes of the memory space used to allocate thread stacks for this application.

CALLING SEQUENCE:

```
uintptr_t rtems_configuration_get_stack_space_size( void );
```

RETURN VALUES:

Returns the configured size in bytes of the memory space used to allocate thread stacks for this application.

NOTES:

The size takes only threads and tasks into account with are known at the application configuration time.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.30 rtems_configuration_get_ticks_per_timeslice()

Gets the clock ticks per timeslice configured for this application.

CALLING SEQUENCE:

```
uint32_t rtems_configuration_get_ticks_per_timeslice( void );
```

RETURN VALUES:

Returns the clock ticks per timeslice configured for this application.

NOTES:

The *clock ticks* per timeslice is defined by the *CONFIGURE_TICKS_PER_TIMESLICE* (page 586) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.31 rtems_configuration_get_unified_work_area()

Indicates if the RTEMS Workspace and C Program Heap are configured to be unified for this application.

CALLING SEQUENCE:

```
bool rtems_configuration_get_unified_work_area( void );
```

RETURN VALUES:

Returns true, if the RTEMS Workspace and C Program Heap are configured to be unified for this application, otherwise false.

NOTES:

The setting is defined by the *CONFIGURE_UNIFIED_WORK_AREAS* (page 587) application configuration option.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.32 rtems_configuration_get_user_extension_table()

Gets the initial extensions table configured for this application.

CALLING SEQUENCE:

```
const rtems_extensions_table *rtems_configuration_get_user_extension_table(
  void
);
```

RETURN VALUES:

Returns a pointer to the initial extensions table configured for this application.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.33 rtems_configuration_get_user_multiprocessing_table()

Gets the MPCI configuration table configured for this application.

CALLING SEQUENCE:

```
const MPCI_Configuration *rtems_configuration_get_user_multiprocessing_table(
  void
);
```

RETURN VALUES:

Returns a pointer to the MPCI configuration table configured for this application.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.34 rtems_configuration_get_work_space_size()

Gets the RTEMS Workspace size in bytes configured for this application.

CALLING SEQUENCE:

```
uintptr_t rtems_configuration_get_work_space_size( void );
```

RETURN VALUES:

Returns the RTEMS Workspace size in bytes configured for this application.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.35 rtems_configuration_get_rtems_api_configuration()

Gets the Classic API Configuration Table of this application.

CALLING SEQUENCE:

```
const rtems_api_configuration_table *
rtems_configuration_get_rtems_api_configuration( void );
```

RETURN VALUES:

Returns a pointer to the Classic API Configuration Table of this application.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 26.24.36 rtems_resource_is_unlimited()

Indicates if the resource is unlimited.

CALLING SEQUENCE:

```
bool rtems_resource_is_unlimited( uint32_t resource );
```

PARAMETERS:

**resource**
This parameter is the resource number.

RETURN VALUES:

Returns true, if the resource is unlimited, otherwise false.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 26.24.37 rtems_resource_maximum_per_allocation()

Gets the maximum number per allocation of a resource number.

CALLING SEQUENCE:

```
uint32_t rtems_resource_maximum_per_allocation( uint32_t resource );
```

PARAMETERS:

**resource**
This parameter is the resource number.

RETURN VALUES:

Returns the maximum number per allocation of a resource number.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 26.24.38 rtems_resource_unlimited()

Augments the resource number so that it indicates an unlimited resource.

CALLING SEQUENCE:

```
uint32_t rtems_resource_unlimited( uint32_t resource );
```

PARAMETERS:

**resource**
This parameter is the resource number to augment.

RETURN VALUES:

Returns the resource number augmented to indicate an unlimited resource.

NOTES:

This directive should be used to configure unlimited objects, see *Unlimited Objects* (page 564).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

## 26.25 Obsolete Configuration Options

### 26.25.1 CONFIGURE_BDBUF_BUFFER_COUNT

This configuration option was introduced in RTEMS 4.7.0 and is obsolete since RTEMS 4.10.0.

### 26.25.2 CONFIGURE_BDBUF_BUFFER_SIZE

This configuration option was introduced in RTEMS 4.7.0 and is obsolete since RTEMS 4.10.0.

### 26.25.3 CONFIGURE_DISABLE_CLASSIC_API_NOTEPADS

This configuration option was introduced in RTEMS 4.9.0 and is obsolete since RTEMS 5.1.

### 26.25.4 CONFIGURE_ENABLE_GO

This configuration option is obsolete since RTEMS 5.1.

### 26.25.5 CONFIGURE_GNAT_RTEMS

This configuration option was present in all RTEMS versions since 1997 and is obsolete since RTEMS 5.1. See also *Ada Configuration* (page 741).

### 26.25.6 CONFIGURE_HAS_OWN_CONFIGURATION_TABLE

This configuration option is obsolete since RTEMS 5.1.

### 26.25.7 CONFIGURE_HAS_OWN_BDBUF_TABLE

This configuration option was introduced in RTEMS 4.7.0 and is obsolete since RTEMS 4.10.0.

### 26.25.8 CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE

This configuration option was present in all RTEMS versions since at least 1995 and is obsolete since RTEMS 5.1.

### 26.25.9 CONFIGURE_HAS_OWN_INIT_TASK_TABLE

This configuration option was present in all RTEMS versions since at least 1995 and is obsolete since RTEMS 5.1. If you used this configuration option or you think that there should be a way to configure more than one Classic API initialization task, then please ask on the Users Mailing List.

### 26.25.10 CONFIGURE_HAS_OWN_MOUNT_TABLE

This configuration option is obsolete since RTEMS 5.1.

### 26.25.11 CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE

This configuration option is obsolete since RTEMS 5.1.

### 26.25.12 CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS

This configuration option was present in all RTEMS versions since 1998 and is obsolete since RTEMS 5.1. See also *CONFIGURE_MAXIMUM_FILE_DESCRIPTORS* (page 576).

### 26.25.13 CONFIGURE_MAXIMUM_ADA_TASKS

This configuration option was present in all RTEMS versions since 1997 and is obsolete since RTEMS 5.1. See also *Ada Configuration* (page 741).

### 26.25.14 CONFIGURE_MAXIMUM_DEVICES

This configuration option was present in all RTEMS versions since at least 1995 and is obsolete since RTEMS 5.1.

### 26.25.15 CONFIGURE_MAXIMUM_FAKE_ADA_TASKS

This configuration option was present in all RTEMS versions since 1997 and is obsolete since RTEMS 5.1. See also *Ada Configuration* (page 741).

### 26.25.16 CONFIGURE_MAXIMUM_GO_CHANNELS

This configuration option is obsolete since RTEMS 5.1.

### 26.25.17 CONFIGURE_MAXIMUM_GOROUTINES

This configuration option is obsolete since RTEMS 5.1.

### 26.25.18 CONFIGURE_MAXIMUM_MRSP_SEMAPHORES

This configuration option is obsolete since RTEMS 5.1.

### 26.25.19 CONFIGURE_NUMBER_OF_TERMIOS_PORTS

This configuration option is obsolete since RTEMS 5.1.

### 26.25.20 CONFIGURE_MAXIMUM_POSIX_BARRIERS

This configuration option is obsolete since RTEMS 5.1.

### 26.25.21 CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES

This configuration option is obsolete since RTEMS 5.1.

### 26.25.22 CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTORS

This configuration option was introduced in RTEMS 4.10.0 and is obsolete since RTEMS 5.1.

### 26.25.23 CONFIGURE_MAXIMUM_POSIX_MUTEXES

This configuration option is obsolete since RTEMS 5.1.

### 26.25.24 CONFIGURE_MAXIMUM_POSIX_RWLOCKS

This configuration option is obsolete since RTEMS 5.1.

### 26.25.25 CONFIGURE_MAXIMUM_POSIX_SPINLOCKS

This configuration option is obsolete since RTEMS 5.1.

### 26.25.26 CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE

This configuration option was present in all RTEMS versions since at least 1995 and is obsolete since RTEMS 5.1. If you used this configuration option or you think that there should be a way to configure more than one POSIX initialization thread, then please ask on the Users Mailing List.

### 26.25.27 CONFIGURE_SMP_APPLICATION

This configuration option was introduced in RTEMS 4.11.0 and is obsolete since RTEMS 5.1.

### 26.25.28 CONFIGURE_SMP_MAXIMUM_PROCESSORS

This configuration option was introduced in RTEMS 4.11.0 and is obsolete since RTEMS 5.1. See also *CONFIGURE_MAXIMUM_PROCESSORS* (page 577).

### 26.25.29 CONFIGURE_TERMIOS_DISABLED

This configuration option is obsolete since RTEMS 5.1.

CHAPTER

**TWENTYSEVEN**

# SELF-CONTAINED OBJECTS

## 27.1 Introduction

One of the original design goals of RTEMS was the support for heterogeneous computing based on message passing. This was realized by synchronization objects with an architecture-independent identifier provided by the system during object creation (a 32-bit unsigned integer used as a bitfield) and a user-defined four character name. This approach in the so called Classic API has some weaknesses:

- Dynamic memory (the workspace) is used to allocate object pools. This requires a complex configuration with heavy use of the C pre-processor. The unlimited objects support optionally expands and shrinks the object pool. Dynamic memory is strongly discouraged by some coding standards, e.g. MISRA C:2012 [BBB+13].
- Objects are created via function calls which return an object identifier. The object operations use this identifier and map it internally to an object representation.
- The object identifier is only known at run-time. This hinders compiler optimizations and static analysis.
- The objects reside in a table, e.g. they are subject to false sharing of cache lines [Dre07].
- The object operations use a rich set of options and attributes. For each object operation these parameters must be evaluated and validated at run-time to figure out what to do exactly for this operation.

For applications that use fine grained locking the mapping of the identifier to the object representation and the parameter evaluation are a significant overhead that may degrade the performance dramatically. An example is the new network stack (libbsd) which uses hundreds of locks in a basic setup. Another example is the OpenMP support (libgomp).

To overcome these issues new self-contained synchronization objects are available since RTEMS 4.11. Self-contained synchronization objects encapsulate all their state in exactly one data structure. The user must provide the storage space for this structure and nothing more. The user is responsible for the object life-cycle. Initialization and destruction of self-contained synchronization objects cannot fail provided all function parameters are valid. In particular, a not enough memory error cannot happen. It is possible to statically initialize self-contained synchronization objects. This allows an efficient use of static analysis tools.

Several header files define self-contained synchronization objects. The Newlib `<sys/lock.h>` header file provides

- mutexes,
- recursive mutexes,
- condition variables,
- counting semaphores,
- binary semaphores, and
- Futex synchronization [FRK02].

They are used internally in Newlib (e.g. for FILE objects), for the C++11 threads and the OpenMP support (libgomp). The Newlib provided self-contained synchronization objects focus on performance. There are no error checks to catch software errors, e.g. invalid parameters. The application configuration is significantly simplified, since it is no longer necessary to

account for lock objects used by Newlib and GCC. The Newlib defined self-contained synchronization objects can be a statically initialized and reside in the `.bss` section. Destruction is a no-operation.

The header file `<pthread.h>` provides

- POSIX barriers (`pthread_barrier_t`),
- POSIX condition variables (`pthread_cond_t`),
- POSIX mutexes (`pthread_mutex_t`),
- POSIX reader/writer locks (`pthread_rwlock_t`), and
- POSIX spinlocks (`pthread_spinlock_t`)

as self-contained synchronization objects. The POSIX synchronization objects are used for example by the Ada run-time support. The header file `<semaphore.h>` provides self-contained

- POSIX unnamed semaphores (`sem_t` initialized via `sem_init()`).

## 27.2 RTEMS Thread API

To give RTEMS users access to self-contained synchronization objects an API is necessary. One option would be to simply use the POSIX threads API (pthreads), C11 threads or C++11 threads. However, these standard APIs lack for example binary semaphores which are important for task/interrupt synchronization. The timed operations use in general time values specified by seconds and nanoseconds. Setting up the time values in seconds (time_t has 64 bits) and nanoseconds is burdened with a high overhead compared to time values in clock ticks for relative timeouts. The POSIX API mutexes can be configured for various protocols and options, this adds a run-time overhead. There are a variety of error conditions. This is a problem in combination with some coding standards, e.g. MISRA C:2012. APIs used by Linux (e.g. <linux/mutex.h>) or the FreeBSD kernel (e.g. MUTEX(9)) are better suited as a template for high-performance synchronization objects. The goal of the *RTEMS Thread API* is to offer the highest performance with the lowest space-overhead on RTEMS. It should be suitable for device drivers.

## 27.3 Mutual Exclusion

The `rtems_mutex` and `rtems_recursive_mutex` objects provide mutual-exclusion synchronization using the *Priority Inheritance Protocol* (page 29) in uniprocessor configurations or the *O(m) Independence-Preserving Protocol (OMIP)* (page 30) in SMP configurations. Recursive locking should be used with care [Wil12]. The storage space for these object must be provided by the user. There are no defined comparison or assignment operators for these type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_mutex_lock()`,
- `rtems_recursive_mutex_lock()`,
- `rtems_mutex_try_lock()`,
- `rtems_recursive_mutex_try_lock()`,
- `rtems_mutex_unlock()`,
- `rtems_recursive_mutex_unlock()`,
- `rtems_mutex_set_name()`,
- `rtems_recursive_mutex_set_name()`,
- `rtems_mutex_get_name()`,
- `rtems_recursive_mutex_get_name()`,
- `rtems_mutex_destroy()`, and
- `rtems_recursive_mutex_destroy()`

is undefined. Objects of the type `rtems_mutex` must be initialized via

- `RTEMS_MUTEX_INITIALIZER()`, or
- `rtems_mutex_init()`.

They must be destroyed via

- `rtems_mutex_destroy()`.

Objects of the type `rtems_recursive_mutex` must be initialized via

- `RTEMS_RECURSIVE_MUTEX_INITIALIZER()`, or
- `rtems_recursive_mutex_init()`.

They must be destroyed via

- `rtems_recursive_mutex_destroy()`.

### 27.3.1 Static mutex initialization

**CALLING SEQUENCE:**

```
rtems_mutex mutex = RTEMS_MUTEX_INITIALIZER(
  name
);

rtems_recursive_mutex mutex = RTEMS_RECURSIVE_MUTEX_INITIALIZER(
  name
);
```

**DESCRIPTION:**

An initializer for static initialization. It is equivalent to a call to `rtems_mutex_init()` or `rtems_recursive_mutex_init()` respectively.

**NOTES:**

Global mutexes with a `name` of `NULL` may reside in the `.bss` section.

### 27.3.2 Run-time mutex initialization

**CALLING SEQUENCE:**

```
void rtems_mutex_init(
  rtems_mutex *mutex,
  const char  *name
);

void rtems_recursive_mutex_init(
  rtems_recursive_mutex *mutex,
  const char            *name
);
```

**DESCRIPTION:**

Initializes the `mutex` with the specified `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the mutex. A `name` of `NULL` is valid. The mutex is unlocked after initialization.

### 27.3.3 Lock the mutex

**CALLING SEQUENCE:**

```
void rtems_mutex_lock(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_lock(
  rtems_recursive_mutex *mutex
);
```

**DESCRIPTION:**

Locks the `mutex`.

**NOTES:**

This function must be called from thread context with interrupts enabled. In case the mutex is currently locked by another thread, then the thread is blocked until it becomes the mutex owner. Threads wait in priority order.

A recursive lock happens in case the mutex owner tries to lock the mutex again. The result of recursively locking a mutex depends on the mutex variant. For a normal (non-recursive) mutex (`rtems_mutex`) the result is unpredictable. It could block the owner indefinetly or lead to a fatal deadlock error. A recursive mutex (`rtems_recursive_mutex`) can be locked recursively by the mutex owner.

Each mutex lock operation must have a corresponding unlock operation.

### 27.3.4 Try to lock the mutex

**CALLING SEQUENCE:**

```
int rtems_mutex_try_lock(
  rtems_mutex *mutex
);

int rtems_recursive_mutex_try_lock(
  rtems_recursive_mutex *mutex
);
```

**DESCRIPTION:**

Tries to lock the `mutex`. In case the mutex is not locked, it will be locked and the function returns with a return value of `0`. If the mutex is already locked, the function will return with a value of `EBUSY`.

**NOTES:**

This function must be called from thread context with interrupts enabled.

For recursively locking a mutex, please also see the notes for `rtems_mutex_lock()` and `rtems_recursive_mutex_lock()`.

Each mutex lock operation must have a corresponding unlock operation.

### 27.3.5 Unlock the mutex

**CALLING SEQUENCE:**

```
void rtems_mutex_unlock(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_unlock(
  rtems_recursive_mutex *mutex
);
```

**DESCRIPTION:**

Unlocks the `mutex`.

**NOTES:**

This function must be called from thread context with interrupts enabled. In case the currently executing thread is not the owner of the `mutex`, then the result is unpredictable.

Exactly the outer-most unlock will make a recursive mutex available to other threads.

### 27.3.6 Set mutex name

**CALLING SEQUENCE:**

```
void rtems_mutex_set_name(
  rtems_mutex *mutex,
  const char  *name
);

void rtems_recursive_mutex_set_name(
  rtems_recursive_mutex *mutex,
  const char            *name
);
```

**DESCRIPTION:**

Sets the `mutex` name to `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the mutex. A `name` of `NULL` is valid.

### 27.3.7 Get mutex name

**CALLING SEQUENCE:**

```
const char *rtems_mutex_get_name(
  const rtems_mutex *mutex
);

const char *rtems_recursive_mutex_get_name(
  const rtems_recursive_mutex *mutex
);
```

**DESCRIPTION:**
Returns the `mutex` name.

**NOTES:**
The name may be `NULL`.

### 27.3.8 Mutex destruction

**CALLING SEQUENCE:**

```
void rtems_mutex_destroy(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_destroy(
  rtems_recursive_mutex *mutex
);
```

**DESCRIPTION:**
Destroys the `mutex`.

**NOTES:**
In case the mutex is locked or still in use, then the result is unpredictable.

## 27.4 Condition Variables

The `rtems_condition_variable` object provides a condition variable synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_condition_variable_wait()`,
- `rtems_condition_variable_signal()`,
- `rtems_condition_variable_broadcast()`,
- `rtems_condition_variable_set_name()`,
- `rtems_condition_variable_get_name()`, and
- `rtems_condition_variable_destroy()`

is undefined. Objects of this type must be initialized via

- `RTEMS_CONDITION_VARIABLE_INITIALIZER()`, or
- `rtems_condition_variable_init()`.

They must be destroyed via

- `rtems_condition_variable_destroy()`.

### 27.4.1 Static condition variable initialization

**CALLING SEQUENCE:**

```
rtems_condition_variable condition_variable = RTEMS_CONDITION_VARIABLE_
↪INITIALIZER(
  name
);
```

**DESCRIPTION:**

An initializer for static initialization. It is equivalent to a call to `rtems_condition_variable_init()`.

**NOTES:**

Global condition variables with a `name` of `NULL` may reside in the `.bss` section.

### 27.4.2 Run-time condition variable initialization

**CALLING SEQUENCE:**

```
void rtems_condition_variable_init(
  rtems_condition_variable *condition_variable,
  const char               *name
);
```

**DESCRIPTION:**

Initializes the `condition_variable` with the specified `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the condition variable. A `name` of `NULL` is valid.

### 27.4.3 Wait for condition signal

**CALLING SEQUENCE:**

```
void rtems_condition_variable_wait(
  rtems_condition_variable *condition_variable,
  rtems_mutex              *mutex
);
```

**DESCRIPTION:**

Atomically waits for a condition signal and unlocks the mutex. Once the condition is signalled to the thread it wakes up and locks the mutex again.

**NOTES:**

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

### 27.4.4 Signals a condition change

**CALLING SEQUENCE:**

```
void rtems_condition_variable_signal(
  rtems_condition_variable *condition_variable
);
```

**DESCRIPTION:**

Signals a condition change to the highest priority waiting thread. If no threads wait currently on this condition variable, then nothing happens.

### 27.4.5 Broadcasts a condition change

**CALLING SEQUENCE:**

```
void rtems_condition_variable_broadcast(
  rtems_condition_variable *condition_variable
);
```

**DESCRIPTION:**

Signals a condition change to all waiting thread. If no threads wait currently on this condition variable, then nothing happens.

### 27.4.6 Set condition variable name

**CALLING SEQUENCE:**

```
void rtems_condition_variable_set_name(
  rtems_condition_variable *condition_variable,
  const char               *name
);
```

**DESCRIPTION:**

Sets the `condition_variable` name to `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the condition variable. A `name` of `NULL` is valid.

### 27.4.7 Get condition variable name

**CALLING SEQUENCE:**

```
const char *rtems_condition_variable_get_name(
  const rtems_condition_variable *condition_variable
);
```

**DESCRIPTION:**
Returns the `condition_variable` name.

**NOTES:**
The name may be `NULL`.

### 27.4.8 Condition variable destruction

**CALLING SEQUENCE:**

```
void rtems_condition_variable_destroy(
  rtems_condition_variable *condition_variable
);
```

**DESCRIPTION:**
Destroys the `condition_variable`.

**NOTES:**
In case the condition variable still in use, then the result is unpredictable.

## 27.5 Counting Semaphores

The `rtems_counting_semaphore` object provides a counting semaphore synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_counting_semaphore_wait()`,
- `rtems_counting_semaphore_wait_timed_ticks()`,
- `rtems_counting_semaphore_try_wait()`,
- `rtems_counting_semaphore_post()`,
- `rtems_counting_semaphore_set_name()`,
- `rtems_counting_semaphore_get_name()`, and
- `rtems_counting_semaphore_destroy()`

is undefined. Objects of this type must be initialized via

- `RTEMS_COUNTING_SEMAPHORE_INITIALIZER()`, or
- `rtems_counting_semaphore_init()`.

They must be destroyed via

- `rtems_counting_semaphore_destroy()`.

### 27.5.1 Static counting semaphore initialization

**CALLING SEQUENCE:**

```
rtems_counting_semaphore counting_semaphore = RTEMS_COUNTING_SEMAPHORE_
↪INITIALIZER(
  name,
  value
);
```

**DESCRIPTION:**

An initializer for static initialization. It is equivalent to a call to `rtems_counting_semaphore_init()`.

**NOTES:**

Global counting semaphores with a `name` of `NULL` may reside in the `.bss` section.

### 27.5.2 Run-time counting semaphore initialization

**CALLING SEQUENCE:**

```
void rtems_counting_semaphore_init(
  rtems_counting_semaphore *counting_semaphore,
  const char               *name,
  unsigned int              value
);
```

**DESCRIPTION:**

Initializes the `counting_semaphore` with the specified `name` and `value`. The initial value is set to `value`.

**NOTES:**

The `name` must be persistent throughout the life-time of the counting semaphore. A `name` of `NULL` is valid.

### 27.5.3 Wait for a counting semaphore

**CALLING SEQUENCE:**

```
void rtems_counting_semaphore_wait(
  rtems_counting_semaphore *counting_semaphore
);
```

**DESCRIPTION:**

Waits for the `counting_semaphore`. In case the current semaphore value is positive, then the value is decremented and the function returns immediately, otherwise the thread is blocked waiting for a semaphore post.

**NOTES:**

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

### 27.5.4 Wait for a counting semaphore with timeout in ticks

**CALLING SEQUENCE:**

```
int rtems_counting_semaphore_wait_timed_ticks(
  rtems_counting_semaphore *counting_semaphore,
  uint32_t                  ticks
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| `0` | The semaphore wait was successful. |
| `ETIMEDOUT` | The semaphore wait timed out. |

**DESCRIPTION:**

Waits for the `counting_semaphore` with a timeout in `ticks`. In case the current semaphore value is positive, then the value is decremented and the function returns immediately with a return value of `0`, otherwise the thread is blocked waiting for a semaphore post. The time waiting for a semaphore post is limited by `ticks`. A `ticks` value of zero specifies an infinite timeout.

**NOTES:**

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

### 27.5.5 Tries to wait for a counting semaphore

**CALLING SEQUENCE:**

```
int rtems_counting_semaphore_try_wait(
  rtems_counting_semaphore *counting_semaphore
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| `0` | The semaphore wait was successful. |
| `EAGAIN` | The semaphore wait failed. |

**DESCRIPTION:**

Tries to wait for the `counting_semaphore`. In case the current semaphore value is positive, then the value is decremented and the function returns immediately with a return value of `0`, otherwise it returns immediately with a return value of `EAGAIN`.

**NOTES:**

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

### 27.5.6 Post a counting semaphore

**CALLING SEQUENCE:**

```
void rtems_counting_semaphore_post(
  rtems_counting_semaphore *counting_semaphore
);
```

**DESCRIPTION:**

Posts the `counting_semaphore`. In case at least one thread is waiting on the counting semaphore, then the highest priority thread is woken up, otherwise the current value is incremented.

**NOTES:**

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

### 27.5.7 Set counting semaphore name

**CALLING SEQUENCE:**

```
void rtems_counting_semaphore_set_name(
  rtems_counting_semaphore *counting_semaphore,
  const char               *name
);
```

**DESCRIPTION:**

Sets the `counting_semaphore` name to `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the counting semaphore. A `name` of `NULL` is valid.

### 27.5.8 Get counting semaphore name

**CALLING SEQUENCE:**

```
const char *rtems_counting_semaphore_get_name(
  const rtems_counting_semaphore *counting_semaphore
);
```

**DESCRIPTION:**
Returns the `counting_semaphore` name.

**NOTES:**
The name may be `NULL`.

### 27.5.9 Counting semaphore destruction

**CALLING SEQUENCE:**

```
void rtems_counting_semaphore_destroy(
  rtems_counting_semaphore *counting_semaphore
);
```

**DESCRIPTION:**
Destroys the `counting_semaphore`.

**NOTES:**
In case the counting semaphore still in use, then the result is unpredictable.

## 27.6 Binary Semaphores

The `rtems_binary_semaphore` object provides a binary semaphore synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_binary_semaphore_wait()`,
- `rtems_binary_semaphore_wait_timed_ticks()`,
- `rtems_binary_semaphore_try_wait()`,
- `rtems_binary_semaphore_post()`,
- `rtems_binary_semaphore_set_name()`,
- `rtems_binary_semaphore_get_name()`, and
- `rtems_binary_semaphore_destroy()`

is undefined. Objects of this type must be initialized via

- `RTEMS_BINARY_SEMAPHORE_INITIALIZER()`, or
- `rtems_binary_semaphore_init()`.

They must be destroyed via

- `rtems_binary_semaphore_destroy()`.

### 27.6.1 Static binary semaphore initialization

**CALLING SEQUENCE:**

```
rtems_binary_semaphore binary_semaphore = RTEMS_BINARY_SEMAPHORE_INITIALIZER(
  name
);
```

**DESCRIPTION:**

An initializer for static initialization. It is equivalent to a call to `rtems_binary_semaphore_init()`.

**NOTES:**

Global binary semaphores with a `name` of `NULL` may reside in the `.bss` section.

### 27.6.2 Run-time binary semaphore initialization

**CALLING SEQUENCE:**

```
void rtems_binary_semaphore_init(
  rtems_binary_semaphore *binary_semaphore,
  const char             *name
);
```

**DESCRIPTION:**

Initializes the `binary_semaphore` with the specified `name`. The initial value is set to zero.

**NOTES:**

The `name` must be persistent throughout the life-time of the binary semaphore. A `name` of `NULL` is valid.

### 27.6.3 Wait for a binary semaphore

**CALLING SEQUENCE:**

```
void rtems_binary_semaphore_wait(
  rtems_binary_semaphore *binary_semaphore
);
```

**DESCRIPTION:**

Waits for the `binary_semaphore`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately, otherwise the thread is blocked waiting for a semaphore post.

**NOTES:**

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

### 27.6.4 Wait for a binary semaphore with timeout in ticks

**CALLING SEQUENCE:**

```
int rtems_binary_semaphore_wait_timed_ticks(
  rtems_binary_semaphore *binary_semaphore,
  uint32_t               ticks
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| `0` | The semaphore wait was successful. |
| `ETIMEDOUT` | The semaphore wait timed out. |

**DESCRIPTION:**

Waits for the `binary_semaphore` with a timeout in `ticks`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately with a return value of `0`, otherwise the thread is blocked waiting for a semaphore post. The time waiting for a semaphore post is limited by `ticks`. A `ticks` value of zero specifies an infinite timeout.

**NOTES:**

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

### 27.6.5 Tries to wait for a binary semaphore

**CALLING SEQUENCE:**

```
int rtems_binary_semaphore_try_wait(
  rtems_binary_semaphore *binary_semaphore
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| `0` | The semaphore wait was successful. |
| `EAGAIN` | The semaphore wait failed. |

**DESCRIPTION:**

Tries to wait for the `binary_semaphore`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately with a return value of `0`, otherwise it returns immediately with a return value of `EAGAIN`.

**NOTES:**

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

### 27.6.6 Post a binary semaphore

**CALLING SEQUENCE:**

```
void rtems_binary_semaphore_post(
  rtems_binary_semaphore *binary_semaphore
);
```

**DESCRIPTION:**

Posts the `binary_semaphore`. In case at least one thread is waiting on the binary semaphore, then the highest priority thread is woken up, otherwise the current value is set to one.

**NOTES:**

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

### 27.6.7 Set binary semaphore name

**CALLING SEQUENCE:**

```
void rtems_binary_semaphore_set_name(
  rtems_binary_semaphore *binary_semaphore,
  const char             *name
);
```

**DESCRIPTION:**

Sets the `binary_semaphore` name to `name`.

**NOTES:**

The `name` must be persistent throughout the life-time of the binary semaphore. A `name` of `NULL` is valid.

### 27.6.8 Get binary semaphore name

**CALLING SEQUENCE:**

```
const char *rtems_binary_semaphore_get_name(
  const rtems_binary_semaphore *binary_semaphore
);
```

**DESCRIPTION:**
Returns the `binary_semaphore` name.

**NOTES:**
The name may be `NULL`.

### 27.6.9 Binary semaphore destruction

**CALLING SEQUENCE:**

```
void rtems_binary_semaphore_destroy(
  rtems_binary_semaphore *binary_semaphore
);
```

**DESCRIPTION:**
Destroys the `binary_semaphore`.

**NOTES:**
In case the binary semaphore still in use, then the result is unpredictable.

## 27.7 Threads

**Warning**

The self-contained threads support is work in progress. In contrast to the synchronization objects the self-contained thread support is not just an API glue layer to already existing implementations.

The `rtems_thread` object provides a thread of execution.

**CALLING SEQUENCE:**

```
RTEMS_THREAD_INITIALIZER(
  name,
  thread_size,
  priority,
  flags,
  entry,
  arg
);

void rtems_thread_start(
  rtems_thread *thread,
  const char   *name,
  size_t        thread_size,
  uint32_t      priority,
  uint32_t      flags,
  void       ( *entry )( void * ),
  void         *arg
);

void rtems_thread_restart(
  rtems_thread *thread,
  void         *arg
) RTEMS_NO_RETURN;

void rtems_thread_event_send(
  rtems_thread *thread,
  uint32_t      events
);

uint32_t rtems_thread_event_poll(
  rtems_thread *thread,
  uint32_t      events_of_interest
);

uint32_t rtems_thread_event_wait_all(
  rtems_thread *thread,
  uint32_t      events_of_interest
```

(continues on next page)

(continued from previous page)

```
);

uint32_t rtems_thread_event_wait_any(
  rtems_thread *thread,
  uint32_t      events_of_interest
);

void rtems_thread_destroy(
  rtems_thread *thread
);

void rtems_thread_destroy_self(
  void
) RTEMS_NO_RETURN;
```

CHAPTER

**TWENTYEIGHT**

# REGULATOR MANAGER

## 28.1 Introduction

The Regulator Manager provides a set of directives to manage a data flow from a source to a destination. The focus is on regulating the bursty input so that it is delivered to the destination at a regular rate. The directives provided by the Regulator Manager are:

- *rtems_regulator_create()* (page 832) - Creates a regulator.
- *rtems_regulator_delete()* (page 834) - Deletes the regulator.
- *rtems_regulator_obtain_buffer()* (page 836) - Obtain buffer from a regulator.
- *rtems_regulator_release_buffer()* (page 837) - Release buffer to a regulator.
- *rtems_regulator_send()* (page 838) - Send buffer to a regulator.
- *rtems_regulator_get_statistics()* (page 840) - Obtain statistics for a regulator.

## 28.2 Background

The regulator provides facilities to accept bursty input and buffer it as needed before delivering it at a pre-defined periodic rate. The input is referred to as the Source, with the output referred to as the Destination. Messages are accepted from the Source and delivered to the Destination by a user-provided Delivery function.

The Regulator implementation uses the RTEMS Classic API Partition Manager to manage the buffer pool and the RTEMS Classic API Message Queue Manager to send the buffer to the Delivery thread. The Delivery thread invokes a user-provided delivery function to get the message to the Destination.

### 28.2.1 Regulator Buffering

The regulator is designed to sit logically between two entities – a source and a destination, where it limits the traffic sent to the destination to prevent it from being flooded with messages from the source. This can be used to accommodate bursts of input from a source and meter it out to a destination. The maximum number of messages which can be buffered in the regulator is specified by the `maximum_messages` field in the *rtems_regulator_attributes* (page 50) structure passed as an argument to *rtems_regulator_create()* (page 832).

The regulator library accepts an input stream of messages from a source and delivers them to a destination. The regulator assumes that the input stream from the source contains sporadic bursts of data which can exceed the acceptable rate of the destination. By limiting the message rate, the regulator prevents an overflow of messages.

The regulator can be configured for the input buffering required to manage the maximum burst and for the metering rate for the delivery. The delivery rate is in messages per second. If the sender produces data too fast, the regulator will buffer the configured number of messages.

A configuration capability is provided to allow for adaptation to different message streams. The regulator can also support running multiple instances, which could be used on independent message streams.

It is assumed that the application has a design limit on the number of messages which may be buffered. All messages accepted by the regulator, assuming no overflow on input, will eventually be output by the Delivery thread.

### 28.2.2 Message Delivery Rate

The Source sends buffers to the Regulator instance. The Regulator then sends the buffer via a message queue which delivers them to the Delivery thread. The Delivery thread executes periodically at a rate specified by the `delivery_thread_period` field in the *rtems_regulator_attributes* (page 50) structure passed as an argument to *rtems_regulator_create()* (page 832).

During each period, the Delivery thread attempts to receive up to `maximum_to_dequeue_per_period` number of buffers and invoke the Delivery function to deliver each of them to the Destination. The `maximum_to_dequeue_per_period` field in the *rtems_regulator_attributes* (page 50) structure passed as an argument to *rtems_regulator_create()* (page 832).

For example, consider a Source that may produce a burst of up to seven messages every five seconds. But the Destination cannot handle a burst of seven and either drops messages or gives an error. This can be accommodated by a Regulator instance configured as follows:

- `maximum_messages` - 7

- `delivery_thread_period` - one second
- `maximum_to_dequeue_per_period` - 3

In this scenario, the application will use the Delivery thread *rtems_regulator_send()* (page 838) to enqueue the seven messages when they arrive. The Delivery thread will deliver three messages per second. The following illustrates this sequence:

- Time 0: Source sends seven messages
- Time 1: Delivery of messages 1 to 3
- Time 3: Delivery of messages 4 to 6
- Time 3: Delivery of message 7
- Time 4: No messages to deliver

This configuration of the regulator ensures that the Destination does not overflow.

## 28.3 Operations

### 28.3.1 Application Sourcing Data

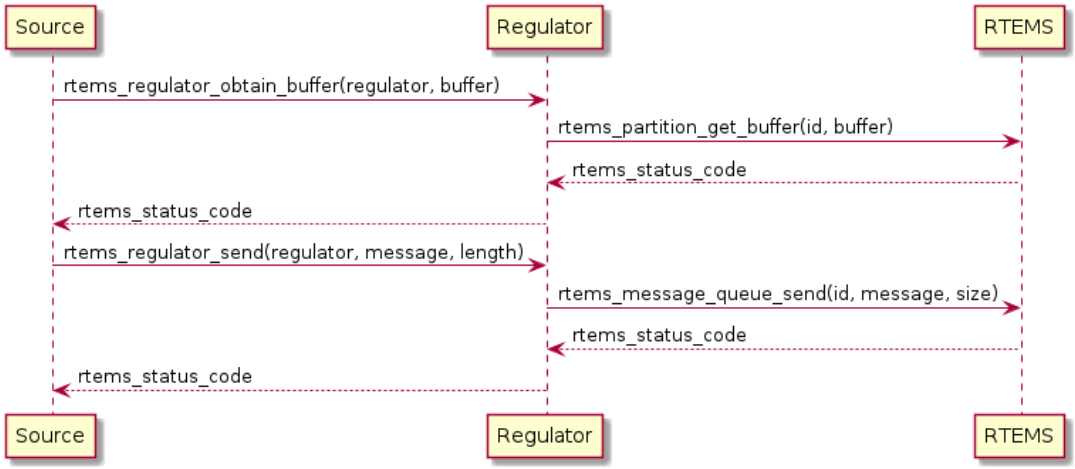
The application interacting with the Source will obtain buffers from the regulator instance, fill them with information, and send them to the regulator instance. This allows the regulator to buffer bursty input.

A regulator instance is used as follows from the Source side:

```
while (1) {
  use rtems_regulator_obtain_buffer to obtain a buffer
  // Perform some input operation to fetch data into the buffer
  rtems_regulator_send(buffer, size of message)
}
```

The delivery of message buffers to the Destination and subsequent release is performed in the context of the delivery thread by either the delivery function or delivery thread. Details are below.

The sequence diagram below shows the interaction between a message Source, a Regulator instance, and RTEMS, given the usage described in the above paragraphs.




As illustrated in the preceding sequence diagram, the Source usually corresponds to application software reading a system input. The Source obtains a buffer from the Regulator instance and fills it with incoming data. The application explicitly obtaining a buffer and filling it in allows for zero copy operations on the Source side.

After the Source has sent the message to the Regulator instance, the Source is free to process another input and the Regulator instance will ensure that the buffer is delivered to the Delivery function and Destination.

### 28.3.2 Delivery Function

The Delivery function is provided by the application for a specific Regulator instance. Depending on the Destination, it may use a function which copies the buffer contents (e.g., write()) or which operates directly on the buffer contents (e.g. DMA from buffer). In the case of a Destination which copies the buffer contents, the buffer can be released via *rtems_regulator_release_buffer()* (page 837) as soon as the function or copying completes. In the case where the delivery uses the buffer and returns, the call to *rtems_regulator_release_buffer()*

(page 837) will occur when the use of the buffer is complete (e.g. completion of DMA transfer). This explicit and deliberate exposure of buffering provides the application with the ability to avoid copying the contents.

## 28.4 Directives

This section details the directives of the Regulator Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 28.4.1 rtems_regulator_create()

Creates a regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_create(
  rtems_regulator_attributes  *attributes,
  rtems_regulator_instance   **regulator
);
```

PARAMETERS:

**attributes**
This parameter is the attributes associated with the regulator being created.

**regulator**
This parameter is the pointer to a regulator instance. When the directive call is successful, a pointer to the created regulator will be stored in this object.

DESCRIPTION:

This function creates an instance of a regulator. It uses the provided `attributes` to create the instance return in `regulator`. This instance will allocate the buffers associated with the regulator instance as well as the Delivery Thread.

The `attributes` parameter points to an instance of *rtems_regulator_attributes* (page 50) which is filled in to reflect the desired configuration of the regulator instance. It defines multiple characteristics of the the Delivery thread dedicated to this regulator instance including the priority and stack size. It also defines the period of the Delivery thread and the maximum number of messages that may be delivered per period via invocation of the delivery function.

For each regulator instance, the following resources are allocated:

- A memory area for the regulator control block using `malloc()`.
- A RTEMS Classic API Message Queue is constructed with message buffer memory allocated using `malloc()`. Each message consists of a pointer to the contents and a length field.
- A RTEMS Classic API Partition.
- A RTEMS Classic API Rate Monotonic Period.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `attributes` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `regulator` parameter was NULL.

**RTEMS_INVALID_ADDRESS**
The `deliverer` field in the structure pointed to by the `attributes` parameter was NULL.

**RTEMS_INVALID_SIZE**
The `maximum_messages` field in the structure pointed to by the `attributes` parameter was 0.

**RTEMS_INVALID_NUMBER**
The `maximum_to_dequeue_per_period` field in the structure pointed to by the `attributes` parameter was 0.

**RTEMS_NO_MEMORY**
The allocation of memory for the regulator instance failed.

**RTEMS_NO_MEMORY**
The allocation of memory for the buffers failed.

**RTEMS_NO_MEMORY**
The allocation of memory for the internal message queue failed.

NOTES:

*rtems_regulator_create()* (page 832) uses *rtems_partition_create()* (page 420), *rtems_message_queue_construct()* (page 375), *rtems_task_create()* (page 109), and *rtems_task_start()* (page 118). If any of those directives return a status indicating failure, it will be returned to the caller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The number of tasks available to the application is configured through the *CONFIGURE_MAXIMUM_TASKS* (page 621) application configuration option.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may allocate memory from the RTEMS Workspace.

### 28.4.2 rtems_regulator_delete()

Deletes the regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_delete(
  rtems_regulator_instance    *regulator,
  rtems_interval               ticks
);
```

PARAMETERS:

**regulator**
This parameter points to the regulator instance.

**ticks**
This parameter specifies the maximum length of time to wait.

DESCRIPTION:

This directive is used to delete the specified `regulator` instance. It will deallocate the resources that were allocated by the *rtems_regulator_create()* (page 832) directive.

This directive ensures that no buffers are outstanding either because the Source is holding one of more buffers or because they are being held by the regulator instance pending delivery.

If the Delivery Thread has been created and is running, this directive will request the thread to voluntarily exit. This call will wait up to `ticks` for the thread to exit.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `regulator` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The `regulator` instance was not initialized.

**RTEMS_RESOURCE_IN_USE**
The `regulator` instance has buffers outstanding.

**RTEMS_TIMEOUT**
The `regulator` instance was not able to be deleted within the specific number of `ticks`.

NOTES:

It is the responsibility of the user to ensure that any resources such as sockets or open file descriptors used by the Source or delivery function are also deleted if necessary. It is likely safer to delete those delivery resources after deleting the regulator instance rather than before.

It is the responsibility of the user to ensure that all buffers associated with this regulator instance have been released and that none are in the process of being delivered.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.
- The calling task does not have to be the task that created the object. Any local task that knows the object identifier can delete the object.
- Where the object class corresponding to the directive is configured to use unlimited objects, the directive may free memory to the RTEMS Workspace.

### 28.4.3 rtems_regulator_obtain_buffer()

Obtain buffer from regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_obtain_buffer(
  rtems_regulator_instance   *regulator,
  void                       **buffer
);
```

PARAMETERS:

**regulator**
This parameter is the regulator instance to operate upon.

**buffer**
This parameter will point to the buffer allocated.

DESCRIPTION:

This function is used to obtain a buffer from the regulator's pool. The `buffer` returned is assumed to be filled in with contents and used in a subsequent call to *rtems_regulator_send()* (page 838).

When the `buffer` is delivered, it is expected to be released. If the `buffer` is not successfully accepted by this method, then it should be returned using *rtems_regulator_release_buffer()* (page 837) or used to send another message.

The `buffer` returned is of the maximum_message_size specified in the attributes passed in to *rtems_regulator_create()* (page 832).

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `regulator` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The `regulator` instance was not initialized.

NOTES:

*rtems_regulator_obtain_buffer()* (page 836) uses *rtems_partition_get_buffer()* (page 427) and if it returns a status indicating failure, it will be returned to the caller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

### 28.4.4 rtems_regulator_release_buffer()

Release buffer to regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_release_buffer(
  rtems_regulator_instance  *regulator,
  void                      *buffer
);
```

PARAMETERS:

**regulator**
This parameter is the regulator instance to operate upon.

**buffer**
This parameter will point to the buffer to be released.

DESCRIPTION:

This function is used to release a buffer to the regulator's pool. It is assumed that the `buffer` returned will not be used by the application anymore.

The `buffer` must have previously been allocated by *rtems_regulator_obtain_buffer()* (page 836) and NOT yet passed to *rtems_regulator_send()* (page 838), or it has been sent and delivery has been completed by the delivery function.

If a subsequent *rtems_regulator_send()* (page 838) using this `buffer` is successful, the `buffer` will eventually be processed by the delivery thread and released.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `regulator` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The `regulator` instance was not initialized.

NOTES:

*rtems_regulator_release_buffer()* (page 837) uses *rtems_partition_return_buffer()* (page 429) and if it returns a status indicating failure, it will be returned to the caller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

### 28.4.5 rtems_regulator_send()

Send buffer to regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_send(
  rtems_regulator_instance  *regulator,
  void                      *message,
  size_t                     length
);
```

PARAMETERS:

**regulator**
This parameter is the regulator instance to operate upon.

**message**
This parameter points to the buffer to send.

**length**
This parameter specifies the number of bytes in the `message`.

DESCRIPTION:

This method is used by the producer to send a `message` to the `regulator` for later delivery by the delivery thread. The message is contained in the memory pointed to by `message` and is `length` bytes in length.

It is required that the message buffer was obtained via *rtems_regulator_obtain_buffer()* (page 836).

It is assumed that the `message` buffer has been filled in with application content to deliver.

If the *rtems_regulator_send()* (page 838) is successful, the `message` buffer is enqueued inside the regulator instance for subsequent delivery. After the `message` is delivered, it may be released by either delivery function or other application code depending on the implementation.

The status `RTEMS_TOO_MANY` is returned if the regulator's internal queue is full. This indicates that the configured maximum number of messages was insufficient. It is the responsibility of the caller to decide whether to hold messages, drop them, or print a message that the maximum number of messages should be increased

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `regulator` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The `regulator` instance was not initialized.

NOTES:

*rtems_regulator_send()* (page 838) uses *rtems_message_queue_send()* (page 381) and if it returns a status indicating failure, it will be returned to the caller.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

### 28.4.6 rtems_regulator_get_statistics()

Obtain statistics from regulator.

CALLING SEQUENCE:

```
rtems_status_code rtems_regulator_get_statistics(
  rtems_regulator_instance   *regulator,
  rtems_regulator_statistics *statistics
);
```

PARAMETERS:

**regulator**
This parameter is the regulator instance to operate upon.

**statistics**
This parameter points to the statistics structure to be filled in.

DESCRIPTION:

This method is used by the application to obtain the current `statistics` for this `regulator`. The statistics information provided includes:

- the number of buffers obtained via *rtems_regulator_obtain_buffer()* (page 836)
- the number of buffers released via *rtems_regulator_release_buffer()* (page 837)
- the number of buffers delivered by the Delivery Thread via the `deliverer` function specified in the *rtems_regulator_attributes* (page 50) structure provided to InterfaceRtemsRegulatorCreate` via the `attibutes` parameter.
- the `period_statistics` for the Delivery Thread. For more details on period statistics, see *rtems_rate_monotonic_period_statistics* (page 49).

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `regulator` or `statistics` parameter was NULL.

**RTEMS_INCORRECT_STATE**
The `regulator` instance was not initialized.

NOTES:

The number of buffers outstanding is `released` minus `obtained`. The regulator instance cannot be deleted using *rtems_regulator_delete()* (page 834) until all buffers are released.

The `obtained` and `released` values are cumulative over the life of the Regulator instance and are likely to larger than the `maximum_messages` value in the `attributes` structure (*rtems_regulator_attributes* (page 50) provided to *rtems_regulator_create()* (page 832).

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.

CHAPTER

**TWENTYNINE**

# MULTIPROCESSING MANAGER

## 29.1 Introduction

The Multiprocessing Manager provides support for heterogeneous multiprocessing systems based on message passing in a network of multiprocessing nodes.

In multiprocessor real-time systems, new requirements, such as sharing data and global resources between processors, are introduced. This requires an efficient and reliable communications vehicle which allows all processors to communicate with each other as necessary. In addition, the ramifications of multiple processors affect each and every characteristic of a real-time system, almost always making them more complicated.

RTEMS addresses these issues by providing simple and flexible real-time multiprocessing capabilities. The executive easily lends itself to both tightly-coupled and loosely-coupled configurations of the target system hardware. In addition, RTEMS supports systems composed of both homogeneous and heterogeneous mixtures of processors and target boards.

A major design goal of the RTEMS executive was to transcend the physical boundaries of the target hardware configuration. This goal is achieved by presenting the application software with a logical view of the target system where the boundaries between processor nodes are transparent. As a result, the application developer may designate objects such as tasks, queues, events, signals, semaphores, and memory blocks as global objects. These global objects may then be accessed by any task regardless of the physical location of the object and the accessing task. RTEMS automatically determines that the object being accessed resides on another processor and performs the actions required to access the desired object. Simply stated, RTEMS allows the entire system, both hardware and software, to be viewed logically as a single system. The directives provided by the Multiprocessing Manager are:

- *rtems_multiprocessing_announce()* (page 854) - Announces the arrival of a packet.

## 29.2 Background

RTEMS makes no assumptions regarding the connection media or topology of a multiprocessor system. The tasks which compose a particular application can be spread among as many processors as needed to satisfy the application's timing requirements. The application tasks can interact using a subset of the RTEMS directives as if they were on the same processor. These directives allow application tasks to exchange data, communicate, and synchronize regardless of which processor they reside upon.

The RTEMS multiprocessor execution model is multiple instruction streams with multiple data streams (MIMD). This execution model has each of the processors executing code independent of the other processors. Because of this parallelism, the application designer can more easily guarantee deterministic behavior.

By supporting heterogeneous environments, RTEMS allows the systems designer to select the most efficient processor for each subsystem of the application. Configuring RTEMS for a heterogeneous environment is no more difficult than for a homogeneous one. In keeping with RTEMS philosophy of providing transparent physical node boundaries, the minimal heterogeneous processing required is isolated in the MPCI layer.

### 29.2.1 Nodes

A processor in a RTEMS system is referred to as a node. Each node is assigned a unique nonzero node number by the application designer. RTEMS assumes that node numbers are assigned consecutively from one to the `maximum_nodes` configuration parameter. The node number, node, and the maximum number of nodes, `maximum_nodes`, in a system are found in the Multiprocessor Configuration Table. The `maximum_nodes` field and the number of global objects, `maximum_global_objects`, is required to be the same on all nodes in a system.

The node number is used by RTEMS to identify each node when performing remote operations. Thus, the Multiprocessor Communications Interface Layer (MPCI) must be able to route messages based on the node number.

### 29.2.2 Global Objects

All RTEMS objects which are created with the GLOBAL attribute will be known on all other nodes. Global objects can be referenced from any node in the system, although certain directive specific restrictions (e.g. one cannot delete a remote object) may apply. A task does not have to be global to perform operations involving remote objects. The maximum number of global objects is the system is user configurable and can be found in the maximum_global_objects field in the Multiprocessor Configuration Table. The distribution of tasks to processors is performed during the application design phase. Dynamic task relocation is not supported by RTEMS.

### 29.2.3 Global Object Table

RTEMS maintains two tables containing object information on every node in a multiprocessor system: a local object table and a global object table. The local object table on each node is unique and contains information for all objects created on this node whether those objects are local or global. The global object table contains information regarding all global objects in the system and, consequently, is the same on every node.

Since each node must maintain an identical copy of the global object table, the maximum number of entries in each copy of the table must be the same. The maximum number of entries in each copy is determined by the maximum_global_objects parameter in the Multiprocessor

Configuration Table. This parameter, as well as the maximum_nodes parameter, is required to be the same on all nodes. To maintain consistency among the table copies, every node in the system must be informed of the creation or deletion of a global object.

### 29.2.4 Remote Operations

When an application performs an operation on a remote global object, RTEMS must generate a Remote Request (RQ) message and send it to the appropriate node. After completing the requested operation, the remote node will build a Remote Response (RR) message and send it to the originating node. Messages generated as a side-effect of a directive (such as deleting a global task) are known as Remote Processes (RP) and do not require the receiving node to respond.

Other than taking slightly longer to execute directives on remote objects, the application is unaware of the location of the objects it acts upon. The exact amount of overhead required for a remote operation is dependent on the media connecting the nodes and, to a lesser degree, on the efficiency of the user-provided MPCI routines.

The following shows the typical transaction sequence during a remote application:

1. The application issues a directive accessing a remote global object.
2. RTEMS determines the node on which the object resides.
3. RTEMS calls the user-provided MPCI routine `GET_PACKET` to obtain a packet in which to build a RQ message.
4. After building a message packet, RTEMS calls the user-provided MPCI routine `SEND_PACKET` to transmit the packet to the node on which the object resides (referred to as the destination node).
5. The calling task is blocked until the RR message arrives, and control of the processor is transferred to another task.
6. The MPCI layer on the destination node senses the arrival of a packet (commonly in an ISR), and calls the `rtems_multiprocessing_announce` directive. This directive readies the Multiprocessing Server.
7. The Multiprocessing Server calls the user-provided MPCI routine `RECEIVE_PACKET`, performs the requested operation, builds an RR message, and returns it to the originating node.
8. The MPCI layer on the originating node senses the arrival of a packet (typically via an interrupt), and calls the RTEMS `rtems_multiprocessing_announce` directive. This directive readies the Multiprocessing Server.
9. The Multiprocessing Server calls the user-provided MPCI routine `RECEIVE_PACKET`, readies the original requesting task, and blocks until another packet arrives. Control is transferred to the original task which then completes processing of the directive.

If an uncorrectable error occurs in the user-provided MPCI layer, the fatal error handler should be invoked. RTEMS assumes the reliable transmission and reception of messages by the MPCI and makes no attempt to detect or correct errors.

### 29.2.5 Proxies

A proxy is an RTEMS data structure which resides on a remote node and is used to represent a task which must block as part of a remote operation. This action can occur as part of the `rtems_semaphore_obtain` and `rtems_message_queue_receive` directives. If the object were local, the task's control block would be available for modification to indicate it was blocking on a message queue or semaphore. However, the task's control block resides only on the same node as the task. As a result, the remote node must allocate a proxy to represent the task until it can be readied.

The maximum number of proxies is defined in the Multiprocessor Configuration Table. Each node in a multiprocessor system may require a different number of proxies to be configured. The distribution of proxy control blocks is application dependent and is different from the distribution of tasks.

### 29.2.6 Multiprocessor Configuration Table

The Multiprocessor Configuration Table contains information needed by RTEMS when used in a multiprocessor system. This table is discussed in detail in the section Multiprocessor Configuration Table of the Configuring a System chapter.

## 29.3 Multiprocessor Communications Interface Layer

The Multiprocessor Communications Interface Layer (MPCI) is a set of user-provided procedures which enable the nodes in a multiprocessor system to communicate with one another. These routines are invoked by RTEMS at various times in the preparation and processing of remote requests. Interrupts are enabled when an MPCI procedure is invoked. It is assumed that if the execution mode and/or interrupt level are altered by the MPCI layer, that they will be restored prior to returning to RTEMS.

The MPCI layer is responsible for managing a pool of buffers called packets and for sending these packets between system nodes. Packet buffers contain the messages sent between the nodes. Typically, the MPCI layer will encapsulate the packet within an envelope which contains the information needed by the MPCI layer. The number of packets available is dependent on the MPCI layer implementation.

The entry points to the routines in the user's MPCI layer should be placed in the Multiprocessor Communications Interface Table. The user must provide entry points for each of the following table entries in a multiprocessor system:

| | |
|---|---|
| initialization | initialize the MPCI |
| get_packet | obtain a packet buffer |
| return_packet | return a packet buffer |
| send_packet | send a packet to another node |
| receive_packet | called to get an arrived packet |

A packet is sent by RTEMS in each of the following situations:

- an RQ is generated on an originating node;
- an RR is generated on a destination node;
- a global object is created;
- a global object is deleted;
- a local task blocked on a remote object is deleted;
- during system initialization to check for system consistency.

If the target hardware supports it, the arrival of a packet at a node may generate an interrupt. Otherwise, the real-time clock ISR can check for the arrival of a packet. In any case, the `rtems_multiprocessing_announce` directive must be called to announce the arrival of a packet. After exiting the ISR, control will be passed to the Multiprocessing Server to process the packet. The Multiprocessing Server will call the get_packet entry to obtain a packet buffer and the receive_entry entry to copy the message into the buffer obtained.

### 29.3.1 INITIALIZATION

The INITIALIZATION component of the user-provided MPCI layer is called as part of the `rtems_initialize_executive` directive to initialize the MPCI layer and associated hardware. It is invoked immediately after all of the device drivers have been initialized. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_initialization( void );
```

Operations on global objects cannot be performed until this component is invoked. The INITIALIZATION component is invoked only once in the life of any system. If the MPCI layer cannot be successfully initialized, the fatal error manager should be invoked by this routine.

One of the primary functions of the MPCI layer is to provide the executive with packet buffers. The INITIALIZATION routine must create and initialize a pool of packet buffers. There must be enough packet buffers so RTEMS can obtain one whenever needed.

### 29.3.2 GET_PACKET

The GET_PACKET component of the user-provided MPCI layer is called when RTEMS must obtain a packet buffer to send or broadcast a message. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_get_packet(
    rtems_packet_prefix **packet
);
```

where packet is the address of a pointer to a packet. This routine always succeeds and, upon return, packet will contain the address of a packet. If for any reason, a packet cannot be successfully obtained, then the fatal error manager should be invoked.

RTEMS has been optimized to avoid the need for obtaining a packet each time a message is sent or broadcast. For example, RTEMS sends response messages (RR) back to the originator in the same packet in which the request message (RQ) arrived.

### 29.3.3 RETURN_PACKET

The RETURN_PACKET component of the user-provided MPCI layer is called when RTEMS needs to release a packet to the free packet buffer pool. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_return_packet(
    rtems_packet_prefix *packet
);
```

where packet is the address of a packet. If the packet cannot be successfully returned, the fatal error manager should be invoked.

### 29.3.4 RECEIVE_PACKET

The RECEIVE_PACKET component of the user-provided MPCI layer is called when RTEMS needs to obtain a packet which has previously arrived. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_receive_packet(
    rtems_packet_prefix **packet
);
```

where packet is a pointer to the address of a packet to place the message from another node. If a message is available, then packet will contain the address of the message from another node. If no messages are available, this entry packet should contain NULL.

### 29.3.5 SEND_PACKET

The SEND_PACKET component of the user-provided MPCI layer is called when RTEMS needs to send a packet containing a message to another node. This component should be adhere to the following prototype:

```
rtems_mpci_entry user_mpci_send_packet(
    uint32_t               node,
    rtems_packet_prefix  **packet
);
```

where node is the node number of the destination and packet is the address of a packet which containing a message. If the packet cannot be successfully sent, the fatal error manager should be invoked.

If node is set to zero, the packet is to be broadcasted to all other nodes in the system. Although some MPCI layers will be built upon hardware which support a broadcast mechanism, others may be required to generate a copy of the packet for each node in the system.

Many MPCI layers use the `packet_length` field of the `rtems_packet_prefix` portion of the packet to avoid sending unnecessary data. This is especially useful if the media connecting the nodes is relatively slow.

The `to_convert` field of the `rtems_packet_prefix` portion of the packet indicates how much of the packet in 32-bit units may require conversion in a heterogeneous system.

### 29.3.6 Supporting Heterogeneous Environments

Developing an MPCI layer for a heterogeneous system requires a thorough understanding of the differences between the processors which comprise the system. One difficult problem is the varying data representation schemes used by different processor types. The most pervasive data representation problem is the order of the bytes which compose a data entity. Processors which place the least significant byte at the smallest address are classified as little endian processors. Little endian byte-ordering is shown below:

```
+---------------+----------------+---------------+----------------+
|               |                |               |                |
|    Byte 3     |     Byte 2     |    Byte 1     |    Byte 0      |
|               |                |               |                |
+---------------+----------------+---------------+----------------+
```

Conversely, processors which place the most significant byte at the smallest address are classified as big endian processors. Big endian byte-ordering is shown below:

```
+---------------+----------------+---------------+----------------+
|               |                |               |                |
|    Byte 0     |     Byte 1     |    Byte 2     |    Byte 3      |
|               |                |               |                |
+---------------+----------------+---------------+----------------+
```

Unfortunately, sharing a data structure between big endian and little endian processors requires translation into a common endian format. An application designer typically chooses the common endian format to minimize conversion overhead.

Another issue in the design of shared data structures is the alignment of data structure elements. Alignment is both processor and compiler implementation dependent. For example, some processors allow data elements to begin on any address boundary, while others impose restrictions. Common restrictions are that data elements must begin on either an even address or on a long word boundary. Violation of these restrictions may cause an exception or impose a performance penalty.

Other issues which commonly impact the design of shared data structures include the representation of floating point numbers, bit fields, decimal data, and character strings. In addition, the representation method for negative integers could be one's or two's complement. These factors combine to increase the complexity of designing and manipulating data structures shared between processors.

RTEMS addressed these issues in the design of the packets used to communicate between nodes. The RTEMS packet format is designed to allow the MPCI layer to perform all necessary conversion without burdening the developer with the details of the RTEMS packet format. As a result, the MPCI layer must be aware of the following:

- All packets must begin on a four byte boundary.
- Packets are composed of both RTEMS and application data. All RTEMS data is treated as 32-bit unsigned quantities and is in the first `to_convert` 32-bit quantities of the packet. The `to_convert` field is part of the `rtems_packet_prefix` portion of the packet.
- The RTEMS data component of the packet must be in native endian format. Endian conversion may be performed by either the sending or receiving MPCI layer.
- RTEMS makes no assumptions regarding the application data component of the packet.

## 29.4 Operations

### 29.4.1 Announcing a Packet

The `rtems_multiprocessing_announce` directive is called by the MPCI layer to inform RTEMS that a packet has arrived from another node. This directive can be called from an interrupt service routine or from within a polling routine.

## 29.5 Directives

This section details the directives of the Multiprocessing Manager. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 29.5.1 rtems_multiprocessing_announce()

Announces the arrival of a packet.

CALLING SEQUENCE:

```
void rtems_multiprocessing_announce( void );
```

DESCRIPTION:

This directive informs RTEMS that a multiprocessing communications packet has arrived from another node. This directive is called by the user-provided MPCI, and is only used in multiprocessing configurations.

NOTES:

This directive is typically called from an *ISR*.

This directive does not generate activity on remote nodes.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within interrupt context.
- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may unblock a task. This may cause the calling task to be preempted.

CHAPTER

**THIRTY**

# SYMMETRIC MULTIPROCESSING (SMP)

## 30.1 Introduction

RTEMS Symmetric Multiprocessing (SMP) support is available on a subset of target architectures supported by RTEMS. Further on some target architectures, it is only available on a subset of BSPs. The user is advised to check the BSP specific documentation and RTEMS source code to verify the status of SMP support for a specific BSP. The following architectures have support for SMP:

- AArch64,
- ARMv7-A,
- i386,
- PowerPC,
- RISC-V, and
- SPARC.

**Warning**

SMP support is only available if RTEMS was built with the SMP build configuration option enabled.

RTEMS is supposed to be a real-time operating system. What does this mean in the context of SMP? The RTEMS interpretation of real-time on SMP is the support for *Clustered Scheduling* (page 858) with priority based schedulers and adequate locking protocols. One aim is to enable a schedulability analysis under the sporadic task model [Bra11] [BW13].

## 30.2 Background

### 30.2.1 Application Configuration

By default, the maximum processor count is set to one in the application configuration. To enable SMP, the application configuration option *CONFIGURE_MAXIMUM_PROCESSORS* (page 577) must be defined to a value greater than one. It is recommended to use the smallest value suitable for the application in order to save memory. Each processor needs an idle thread and interrupt stack for example.

The default scheduler for SMP applications supports up to 32 processors and is a global fixed priority scheduler, see also *Clustered Scheduler Configuration* (page 726).

The following compile-time test can be used to check if the SMP support is available or not.

```
#include <rtems.h>

#ifdef RTEMS_SMP
#warning "SMP support is enabled"
#else
#warning "SMP support is disabled"
#endif
```

### 30.2.2 Examples

For example applications see testsuites/smptests.

### 30.2.3 Uniprocessor versus SMP Parallelism

Uniprocessor systems have long been used in embedded systems. In this hardware model, there are some system execution characteristics which have long been taken for granted:

- one task executes at a time
- hardware events result in interrupts

There is no true parallelism. Even when interrupts appear to occur at the same time, they are processed in largely a serial fashion. This is true even when the interupt service routines are allowed to nest. From a tasking viewpoint, it is the responsibility of the real-time operatimg system to simulate parallelism by switching between tasks. These task switches occur in response to hardware interrupt events and explicit application events such as blocking for a resource or delaying.

With symmetric multiprocessing, the presence of multiple processors allows for true concurrency and provides for cost-effective performance improvements. Uniprocessors tend to increase performance by increasing clock speed and complexity. This tends to lead to hot, power hungry microprocessors which are poorly suited for many embedded applications.

The true concurrency is in sharp contrast to the single task and interrupt model of uniprocessor systems. This results in a fundamental change to uniprocessor system characteristics listed above. Developers are faced with a different set of characteristics which, in turn, break some existing assumptions and result in new challenges. In an SMP system with N processors, these are the new execution characteristics.

- N tasks execute in parallel
- hardware events result in interrupts

There is true parallelism with a task executing on each processor and the possibility of interrupts occurring on each processor. Thus in contrast to their being one task and one interrupt to consider on a uniprocessor, there are N tasks and potentially N simultaneous interrupts to consider on an SMP system.

This increase in hardware complexity and presence of true parallelism results in the application developer needing to be even more cautious about mutual exclusion and shared data access than in a uniprocessor embedded system. Race conditions that never or rarely happened when an application executed on a uniprocessor system, become much more likely due to multiple threads executing in parallel. On a uniprocessor system, these race conditions would only happen when a task switch occurred at just the wrong moment. Now there are N-1 tasks executing in parallel all the time and this results in many more opportunities for small windows in critical sections to be hit.

### 30.2.4 Task Affinity

RTEMS provides services to manipulate the affinity of a task. Affinity is used to specify the subset of processors in an SMP system on which a particular task can execute.

By default, tasks have an affinity which allows them to execute on any available processor.

Task affinity is a possible feature to be supported by SMP-aware schedulers. However, only a subset of the available schedulers support affinity. Although the behavior is scheduler specific, if the scheduler does not support affinity, it is likely to ignore all attempts to set affinity.

The scheduler with support for arbitary processor affinities uses a proof of concept implementation. See https://gitlab.rtems.org/rtems/programs/gsoc/-/issues/34

### 30.2.5 Task Migration

With more than one processor in the system tasks can migrate from one processor to another. There are four reasons why tasks migrate in RTEMS.

- The scheduler changes explicitly via rtems_task_set_scheduler() or similar directives.
- The task processor affinity changes explicitly via rtems_task_set_affinity() or similar directives.
- The task resumes execution after a blocking operation. On a priority based scheduler it will evict the lowest priority task currently assigned to a processor in the processor set managed by the scheduler instance.
- The task moves temporarily to another scheduler instance due to locking protocols like the *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30) or the *O(m) Independence-Preserving Protocol (OMIP)* (page 30).

Task migration should be avoided so that the working set of a task can stay on the most local cache level.

### 30.2.6 Clustered Scheduling

The scheduler is responsible to assign processors to some of the threads which are ready to execute. Trouble starts if more ready threads than processors exist at the same time. There are various rules how the processor assignment can be performed attempting to fulfill additional constraints or yield some overall system properties. As a matter of fact it is impossible to meet all requirements at the same time. The way a scheduler works distinguishes real-time operating systems from general purpose operating systems.

We have clustered scheduling in case the set of processors of a system is partitioned into non-empty pairwise-disjoint subsets of processors. These subsets are called clusters. Clusters with a cardinality of one are partitions. Each cluster is owned by exactly one scheduler instance. In case the cluster size equals the processor count, it is called global scheduling.

Modern SMP systems have multi-layer caches. An operating system which neglects cache constraints in the scheduler will not yield good performance. Real-time operating systems usually provide priority (fixed or job-level) based schedulers so that each of the highest priority threads is assigned to a processor. Priority based schedulers have difficulties in providing cache locality for threads and may suffer from excessive thread migrations [Bra11] [CMV14]. Schedulers that use local run queues and some sort of load-balancing to improve the cache utilization may not fulfill global constraints [GCB13] and are more difficult to implement than one would normally expect [LLF+16].

Clustered scheduling was implemented for RTEMS SMP to best use the cache topology of a system and to keep the worst-case latencies under control. The low-level SMP locks use FIFO ordering. So, the worst-case run-time of operations increases with each processor involved. The scheduler configuration is quite flexible and done at link-time, see *Clustered Scheduler Configuration* (page 726). It is possible to re-assign processors to schedulers during run-time via rtems_scheduler_add_processor() and rtems_scheduler_remove_processor(). The schedulers are implemented in an object-oriented fashion.

The problem is to provide synchronization primitives for inter-cluster synchronization (more than one cluster is involved in the synchronization process). In RTEMS there are currently some means available

- events,
- message queues,
- mutexes using the *O(m) Independence-Preserving Protocol (OMIP)* (page 30),
- mutexes using the *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30), and
- binary and counting semaphores.

The clustered scheduling approach enables separation of functions with real-time requirements and functions that profit from fairness and high throughput provided the scheduler instances are fully decoupled and adequate inter-cluster synchronization primitives are used.

To set the scheduler of a task see rtems_scheduler_ident() and rtems_task_set_scheduler().

### 30.2.7 OpenMP

OpenMP support for RTEMS is available via the GCC provided libgomp. There is libgomp support for RTEMS in the POSIX configuration of libgomp since GCC 4.9 (requires a Newlib snapshot after 2015-03-12). In GCC 6.1 or later (requires a Newlib snapshot after 2015-07-30 for <sys/lock.h> provided self-contained synchronization objects) there is a specialized libgomp configuration for RTEMS which offers a significantly better performance compared to the POSIX configuration of libgomp. In addition application configurable thread pools for each scheduler instance are available in GCC 6.1 or later.

The run-time configuration of libgomp is done via environment variables documented in the libgomp manual. The environment variables are evaluated in a constructor function which executes in the context of the first initialization task before the actual initialization task function is called (just like a global C++ constructor). To set application specific values, a higher priority constructor function must be used to set up the environment variables.

```
#include <stdlib.h>
void __attribute__((constructor(1000))) config_libgomp( void )
{
    setenv( "OMP_DISPLAY_ENV", "VERBOSE", 1 );
    setenv( "GOMP_SPINCOUNT", "30000", 1 );
    setenv( "GOMP_RTEMS_THREAD_POOLS", "1$2@SCHD", 1 );
}
```

The environment variable `GOMP_RTEMS_THREAD_POOLS` is RTEMS-specific. It determines the thread pools for each scheduler instance. The format for `GOMP_RTEMS_THREAD_POOLS` is a list of optional `<thread-pool-count>[$<priority>]@<scheduler-name>` configurations separated by `:` where:

- `<thread-pool-count>` is the thread pool count for this scheduler instance.
- `$<priority>` is an optional priority for the worker threads of a thread pool according to `pthread_setschedparam`. In case a priority value is omitted, then a worker thread will inherit the priority of the OpenMP master thread that created it. The priority of the worker thread is not changed by libgomp after creation, even if a new OpenMP master thread using the worker has a different priority.
- `@<scheduler-name>` is the scheduler instance name according to the RTEMS application configuration.

In case no thread pool configuration is specified for a scheduler instance, then each OpenMP master thread of this scheduler instance will use its own dynamically allocated thread pool. To limit the worker thread count of the thread pools, each OpenMP master thread must call `omp_set_num_threads`.

Lets suppose we have three scheduler instances `IO`, `WRK0`, and `WRK1` with `GOMP_RTEMS_THREAD_POOLS` set to `"1@WRK0:3$4@WRK1"`. Then there are no thread pool restrictions for scheduler instance `IO`. In the scheduler instance `WRK0` there is one thread pool available. Since no priority is specified for this scheduler instance, the worker thread inherits the priority of the OpenMP master thread that created it. In the scheduler instance `WRK1` there are three thread pools available and their worker threads run at priority four.

### 30.2.8 Atomic Operations

There is no public RTEMS API for atomic operations. It is recommended to use the standard C <stdatomic.h> or C++ <atomic> APIs in applications.

## 30.3 Application Issues

Most operating system services provided by the uniprocessor RTEMS are available in SMP configurations as well. However, applications designed for an uniprocessor environment may need some changes to correctly run in an SMP configuration.

As discussed earlier, SMP systems have opportunities for true parallelism which was not possible on uniprocessor systems. Consequently, multiple techniques that provided adequate critical sections on uniprocessor systems are unsafe on SMP systems. In this section, some of these unsafe techniques will be discussed.

In general, applications must use proper operating system provided mutual exclusion mechanisms to ensure correct behavior.

### 30.3.1 Task variables

Task variables are ordinary global variables with a dedicated value for each thread. During a context switch from the executing thread to the heir thread, the value of each task variable is saved to the thread control block of the executing thread and restored from the thread control block of the heir thread. This is inherently broken if more than one executing thread exists. Alternatives to task variables are POSIX keys and *TLS*. All use cases of task variables in the RTEMS code base were replaced with alternatives. The task variable API has been removed in RTEMS 5.1.

### 30.3.2 Highest Priority Thread Never Walks Alone

On a uniprocessor system, it is safe to assume that when the highest priority task in an application executes, it will execute without being preempted until it voluntarily blocks. Interrupts may occur while it is executing, but there will be no context switch to another task unless the highest priority task voluntarily initiates it.

Given the assumption that no other tasks will have their execution interleaved with the highest priority task, it is possible for this task to be constructed such that it does not need to acquire a mutex for protected access to shared data.

In an SMP system, it cannot be assumed there will never be a single task executing. It should be assumed that every processor is executing another application task. Further, those tasks will be ones which would not have been executed in a uniprocessor configuration and should be assumed to have data synchronization conflicts with what was formerly the highest priority task which executed without conflict.

### 30.3.3 Disabling of Thread Preemption

A thread which disables preemption prevents that a higher priority thread gets hold of its processor involuntarily. In uniprocessor configurations, this can be used to ensure mutual exclusion at thread level. In SMP configurations, however, more than one executing thread may exist. Thus, it is impossible to ensure mutual exclusion using this mechanism. In order to prevent that applications using preemption for this purpose, would show inappropriate behaviour, this feature is disabled in SMP configurations and its use would case run-time errors.

### 30.3.4 Disabling of Interrupts

A low overhead means that ensures mutual exclusion in uniprocessor configurations is the disabling of interrupts around a critical section. This is commonly used in device driver code. In SMP configurations, however, disabling the interrupts on one processor has no effect on other processors. So, this is insufficient to ensure system-wide mutual exclusion. The macros

- rtems_interrupt_disable(),
- rtems_interrupt_enable(), and
- rtems_interrupt_flash().

are disabled in SMP configurations and its use will cause compile-time warnings and link-time errors. In the unlikely case that interrupts must be disabled on the current processor, the

- rtems_interrupt_local_disable(), and
- rtems_interrupt_local_enable().

macros are now available in all configurations.

Since disabling of interrupts is insufficient to ensure system-wide mutual exclusion on SMP a new low-level synchronization primitive was added – interrupt locks. The interrupt locks are a simple API layer on top of the SMP locks used for low-level synchronization in the operating system core. Currently, they are implemented as a ticket lock. In uniprocessor configurations, they degenerate to simple interrupt disable/enable sequences by means of the C pre-processor. It is disallowed to acquire a single interrupt lock in a nested way. This will result in an infinite loop with interrupts disabled. While converting legacy code to interrupt locks, care must be taken to avoid this situation to happen.

```c
#include <rtems.h>

void legacy_code_with_interrupt_disable_enable( void )
{
  rtems_interrupt_level level;

  rtems_interrupt_disable( level );
  /* Critical section */
  rtems_interrupt_enable( level );
}

RTEMS_INTERRUPT_LOCK_DEFINE( static, lock, "Name" )

void smp_ready_code_with_interrupt_lock( void )
{
  rtems_interrupt_lock_context lock_context;

  rtems_interrupt_lock_acquire( &lock, &lock_context );
  /* Critical section */
  rtems_interrupt_lock_release( &lock, &lock_context );
}
```

An alternative to the RTEMS-specific interrupt locks are POSIX spinlocks. The `pthread_spinlock_t` is defined as a self-contained object, e.g. the user must provide the storage for this synchronization object.

```c
#include <assert.h>
#include <pthread.h>

pthread_spinlock_t lock;

void smp_ready_code_with_posix_spinlock( void )
{
  int error;

  error = pthread_spin_lock( &lock );
  assert( error == 0 );
  /* Critical section */
  error = pthread_spin_unlock( &lock );
  assert( error == 0 );
}
```

In contrast to POSIX spinlock implementation on Linux or FreeBSD, it is not allowed to call blocking operating system services inside the critical section. A recursive lock attempt is a severe usage error resulting in an infinite loop with interrupts disabled. Nesting of different locks is allowed. The user must ensure that no deadlock can occur. As a non-portable feature the locks are zero-initialized, e.g. statically initialized global locks reside in the `.bss` section and there is no need to call `pthread_spin_init()`.

### 30.3.5 Interrupt Service Routines Execute in Parallel With Threads

On a machine with more than one processor, interrupt service routines (this includes timer service routines installed via rtems_timer_fire_after()) and threads can execute in parallel. Interrupt service routines must take this into account and use proper locking mechanisms to protect critical sections from interference by threads (interrupt locks or POSIX spinlocks). This likely requires code modifications in legacy device drivers.

### 30.3.6 Timers Do Not Stop Immediately

Timer service routines run in the context of the clock interrupt. On uniprocessor configurations, it is sufficient to disable interrupts and remove a timer from the set of active timers to stop it. In SMP configurations, however, the timer service routine may already run and wait on an SMP lock owned by the thread which is about to stop the timer. This opens the door to subtle synchronization issues. During destruction of objects, special care must be taken to ensure that timer service routines cannot access (partly or fully) destroyed objects.

### 30.3.7 False Sharing of Cache Lines Due to Objects Table

The Classic API and most POSIX API objects are indirectly accessed via an object identifier. The user-level functions validate the object identifier and map it to the actual object structure which resides in a global objects table for each object class. So, unrelated objects are packed together in a table. This may result in false sharing of cache lines. The effect of false sharing of cache lines can be observed with the TMFINE 1 test program on a suitable platform, e.g. QorIQ T4240. High-performance SMP applications need full control of the object storage [Dre07]. Therefore, self-contained synchronization objects are now available for RTEMS.

## 30.4 Implementation Details

This section covers some implementation details of the RTEMS SMP support.

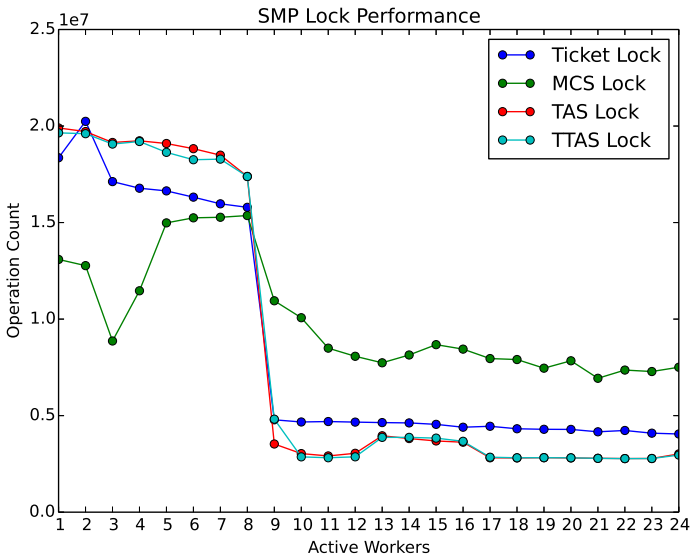
### 30.4.1 Low-Level Synchronization

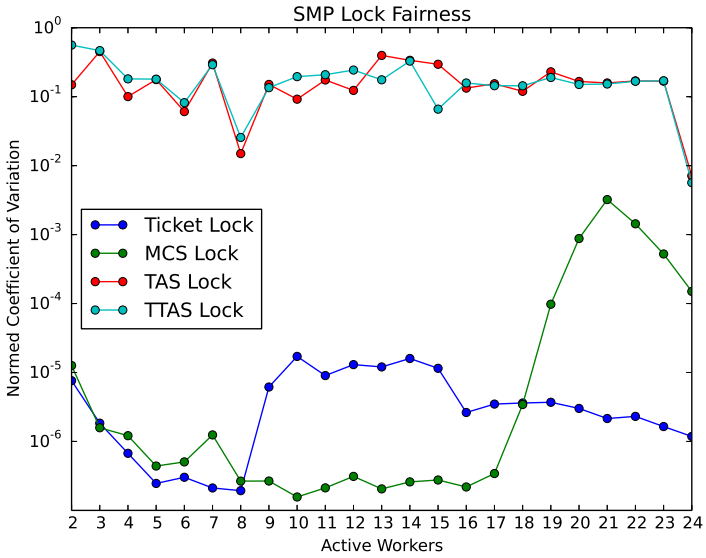
All low-level synchronization primitives are implemented using *C11* atomic operations, so no target-specific hand-written assembler code is necessary. Four synchronization primitives are currently available

- ticket locks (mutual exclusion),
- *MCS* locks (mutual exclusion),
- barriers, implemented as a sense barrier, and
- sequence locks [Boe12].

A vital requirement for low-level mutual exclusion is *FIFO* fairness since we are interested in a predictable system and not maximum throughput. With this requirement, there are only few options to resolve this problem. For reasons of simplicity, the ticket lock algorithm was chosen to implement the SMP locks. However, the API is capable to support MCS locks, which may be interesting in the future for systems with a processor count in the range of 32 or more, e.g. *NUMA*, many-core systems.

The test program SMPLOCK 1 can be used to gather performance and fairness data for several scenarios. The SMP lock performance and fairness measured on the QorIQ T4240 follows as an example. This chip contains three L2 caches. Each L2 cache is shared by eight processors.






### 30.4.2 Internal Locking

In SMP configurations, the operating system uses non-recursive SMP locks for low-level mutual exclusion. The locking domains are roughly

- a particular data structure,
- the thread queue operations,
- the thread state changes, and
- the scheduler operations.

For a good average-case performance it is vital that every high-level synchronization object, e.g. mutex, has its own SMP lock. In the average-case, only this SMP lock should be involved to carry out a specific operation, e.g. obtain/release a mutex. In general, the high-level synchronization objects have a thread queue embedded and use its SMP lock.

In case a thread must block on a thread queue, then things get complicated. The executing thread first acquires the SMP lock of the thread queue and then figures out that it needs to block. The procedure to block the thread on this particular thread queue involves state changes of the thread itself and for this thread-specific SMP locks must be used.

In order to determine if a thread is blocked on a thread queue or not thread-specific SMP locks must be used. A thread priority change must propagate this to the thread queue (possibly recursively). Care must be taken to not have a lock order reversal between thread queue and thread-specific SMP locks.

Each scheduler instance has its own SMP lock. For the scheduler helping protocol multiple scheduler instances may be in charge of a thread. It is not possible to acquire two scheduler instance SMP locks at the same time, otherwise deadlocks would happen. A thread-specific SMP lock is used to synchronize the thread data shared by different scheduler instances.

The thread state SMP lock protects various things, e.g. the thread state, join operations, signals, post-switch actions, the home scheduler instance, etc.

### 30.4.3 Profiling

To identify the bottlenecks in the system, support for profiling of low-level synchronization is optionally available. The profiling support is an RTEMS build time configuration option and is implemented with an acceptable overhead, even for production systems. A low-overhead counter for short time intervals must be provided by the hardware.

Profiling reports are generated in XML for most test programs of the RTEMS testsuite (more than 500 test programs). This gives a good sample set for statistics. For example the maximum thread dispatch disable time, the maximum interrupt latency or lock contention can be determined.

```
<ProfilingReport name="SMPMIGRATION 1">
  <PerCPUProfilingReport processorIndex="0">
    <MaxThreadDispatchDisabledTime unit="ns">36636</MaxThreadDispatchDisabledTime>
    <MeanThreadDispatchDisabledTime unit="ns">5065</
↪MeanThreadDispatchDisabledTime>
    <TotalThreadDispatchDisabledTime unit="ns">3846635988
      </TotalThreadDispatchDisabledTime>
    <ThreadDispatchDisabledCount>759395</ThreadDispatchDisabledCount>
    <MaxInterruptDelay unit="ns">8772</MaxInterruptDelay>
    <MaxInterruptTime unit="ns">13668</MaxInterruptTime>
    <MeanInterruptTime unit="ns">6221</MeanInterruptTime>
    <TotalInterruptTime unit="ns">6757072</TotalInterruptTime>
    <InterruptCount>1086</InterruptCount>
  </PerCPUProfilingReport>
  <PerCPUProfilingReport processorIndex="1">
    <MaxThreadDispatchDisabledTime unit="ns">39408</MaxThreadDispatchDisabledTime>
    <MeanThreadDispatchDisabledTime unit="ns">5060</
↪MeanThreadDispatchDisabledTime>
    <TotalThreadDispatchDisabledTime unit="ns">3842749508
      </TotalThreadDispatchDisabledTime>
    <ThreadDispatchDisabledCount>759391</ThreadDispatchDisabledCount>
    <MaxInterruptDelay unit="ns">8412</MaxInterruptDelay>
    <MaxInterruptTime unit="ns">15868</MaxInterruptTime>
    <MeanInterruptTime unit="ns">3525</MeanInterruptTime>
    <TotalInterruptTime unit="ns">3814476</TotalInterruptTime>
    <InterruptCount>1082</InterruptCount>
  </PerCPUProfilingReport>
  <!-- more reports omitted --->
  <SMPLockProfilingReport name="Scheduler">
    <MaxAcquireTime unit="ns">7092</MaxAcquireTime>
    <MaxSectionTime unit="ns">10984</MaxSectionTime>
    <MeanAcquireTime unit="ns">2320</MeanAcquireTime>
    <MeanSectionTime unit="ns">199</MeanSectionTime>
    <TotalAcquireTime unit="ns">3523939244</TotalAcquireTime>
    <TotalSectionTime unit="ns">302545596</TotalSectionTime>
    <UsageCount>1518758</UsageCount>
    <ContentionCount initialQueueLength="0">759399</ContentionCount>
    <ContentionCount initialQueueLength="1">759359</ContentionCount>
    <ContentionCount initialQueueLength="2">0</ContentionCount>
    <ContentionCount initialQueueLength="3">0</ContentionCount>
  </SMPLockProfilingReport>
```

(continues on next page)

(continued from previous page)

```
</ProfilingReport>
```

### 30.4.4 Scheduler Helping Protocol

The scheduler provides a helping protocol to support locking protocols like the *O(m) Independence-Preserving Protocol (OMIP)* (page 30) or the *Multiprocessor Resource Sharing Protocol (MrsP)* (page 30). Each thread has a scheduler node for each scheduler instance in the system which are located in its *TCB*. A thread has exactly one home scheduler instance which is set during thread creation. The home scheduler instance can be changed with rtems_task_set_scheduler(). Due to the locking protocols a thread may gain access to scheduler nodes of other scheduler instances. This allows the thread to temporarily migrate to another scheduler instance in case of preemption.

The scheduler infrastructure is based on an object-oriented design. The scheduler operations for a thread are defined as virtual functions. For the scheduler helping protocol the following operations must be implemented by an SMP-aware scheduler

- ask a scheduler node for help,
- reconsider the help request of a scheduler node,
- withdraw a schedule node.

All currently available SMP-aware schedulers use a framework which is customized via inline functions. This eases the implementation of scheduler variants. Up to now, only priority-based schedulers are implemented.

In case a thread is allowed to use more than one scheduler node it will ask these nodes for help

- in case of preemption, or
- an unblock did not schedule the thread, or
- a yield was successful.

The actual ask for help scheduler operations are carried out as a side-effect of the thread dispatch procedure. Once a need for help is recognized, a help request is registered in one of the processors related to the thread and a thread dispatch is issued. This indirection leads to a better decoupling of scheduler instances. Unrelated processors are not burdened with extra work for threads which participate in resource sharing. Each ask for help operation indicates if it could help or not. The procedure stops after the first successful ask for help. Unsuccessful ask for help operations will register this need in the scheduler context.

After a thread dispatch the reconsider help request operation is used to clean up stale help registrations in the scheduler contexts.

The withdraw operation takes away scheduler nodes once the thread is no longer allowed to use them, e.g. it released a mutex. The availability of scheduler nodes for a thread is controlled by the thread queues.

### 30.4.5 Thread Dispatch Details

This section gives background information to developers interested in the interrupt latencies introduced by thread dispatching. A thread dispatch consists of all work which must be done to stop the currently executing thread on a processor and hand over this processor to an heir thread.

In SMP systems, scheduling decisions on one processor must be propagated to other processors through inter-processor interrupts. A thread dispatch which must be carried out on another processor does not happen instantaneously. Thus, several thread dispatch requests might be in the air and it is possible that some of them may be out of date before the corresponding processor has time to deal with them. The thread dispatch mechanism uses three per-processor variables,

- the executing thread,
- the heir thread, and
- a boolean flag indicating if a thread dispatch is necessary or not.

Updates of the heir thread are done via a normal store operation. The thread dispatch necessary indicator of another processor is set as a side-effect of an inter-processor interrupt. So, this change notification works without the use of locks. The thread context is protected by a *TTAS* lock embedded in the context to ensure that it is used on at most one processor at a time. Normally, only thread-specific or per-processor locks are used during a thread dispatch. This implementation turned out to be quite efficient and no lock contention was observed in the testsuite. The heavy-weight thread dispatch sequence is only entered in case the thread dispatch indicator is set.

The context-switch is performed with interrupts enabled. During the transition from the executing to the heir thread neither the stack of the executing nor the heir thread must be used during interrupt processing. For this purpose a temporary per-processor stack is set up which may be used by the interrupt prologue before the stack is switched to the interrupt stack.

### 30.4.6 Per-Processor Data

RTEMS provides two means for per-processor data:

1. Per-processor data which is used by RTEMS itself is contained in the *Per_CPU_Control* structure. The application configuration via *<rtems/confdefs.h>* creates a table of these structures (*_Per_CPU_Information[]*). The table is dimensioned according to the count of configured processors (*CONFIGURE_MAXIMUM_PROCESSORS* (page 577)).
2. For low level support libraries an API for statically allocated per-processor data is available via <rtems/score/percpudata.h>. This API is not intended for general application use. Please ask on the development mailing list in case you want to use it.

### 30.4.7 Thread Pinning

Thread pinning ensures that a thread is only dispatched to the processor on which it is pinned. It may be used to access per-processor data structures in critical sections with enabled thread dispatching, e.g. a pinned thread is allowed to block. The *_Thread_Pin()* operation will pin the executing thread to its current processor. A thread may be pinned recursively, the last unpin request via *_Thread_Unpin()* revokes the pinning.

Thread pinning should be used only for short critical sections and not all the time. Thread pinning is a very low overhead operation in case the thread is not preempted during the pinning. A preemption will result in scheduler operations to ensure that the thread executes only on its pinned processor. Thread pinning must be used with care, since it prevents help through the locking protocols. This makes the *OMIP* (page 30) and *MrsP* (page 30) locking protocols ineffective if pinned threads are involved.

The thread pinning is not intended for general application use. Please ask on the development mailing list in case you want to use it.

CHAPTER

**THIRTYONE**

# PCI LIBRARY

## 31.1 Introduction

The Peripheral Component Interconnect (PCI) bus is a very common computer bus architecture that is found in almost every PC today. The PCI bus is normally located at the motherboard where some PCI devices are soldered directly onto the PCB and expansion slots allows the user to add custom devices easily. There is a wide range of PCI hardware available implementing all sorts of interfaces and functions.

This section describes the PCI Library available in RTEMS used to access the PCI bus in a portable way across computer architectures supported by RTEMS.

The PCI Library aims to be compatible with PCI 2.3 with a couple of limitations, for example there is no support for hot-plugging, 64-bit memory space and cardbus bridges.

In order to support different architectures and with small foot-print embedded systems in mind the PCI Library offers four different configuration options listed below. It is selected during compile time by defining the appropriate macros in confdefs.h. It is also possible to enable PCI_LIB_NONE (No Configuration) which can be used for debuging PCI access functions.

- Auto Configuration (Plug & Play)
- Read Configuration (read BIOS or boot loader configuration)
- Static Configuration (write user defined configuration)
- Peripheral Configuration (no access to cfg-space)

## 31.2 Background

The PCI bus is constructed in a way where on-board devices and devices in expansion slots can be automatically found (probed) and configured using Plug & Play completely implemented in software. The bus is set up once during boot up. The Plug & Play information can be read and written from PCI configuration space. A PCI device is identified in configuration space by a unique bus, slot and function number. Each PCI slot can have up to 8 functions and interface to another PCI sub-bus by implementing a PCI-to-PCI bridge according to the PCI Bridge Architecture specification.

Using the unique [bus:slot:func] any device can be configured regardless of how PCI is currently set up as long as all PCI buses are enumerated correctly. The enumeration is done during probing, all bridges are given a bus number in order for the bridges to respond to accesses from both directions. The PCI library can assign address ranges to which a PCI device should respond using Plug & Play technique or a static user defined configuration. After the configuration has been performed the PCI device drivers can find devices by the read-only PCI Class type, Vendor ID and Device ID information found in configuration space for each device.

In some systems there is a boot loader or BIOS which have already configured all PCI devices, but on embedded targets it is quite common that there is no BIOS or boot loader, thus RTEMS must configure the PCI bus. Only the PCI host may do configuration space access, the host driver or BSP is responsible to translate the [bus:slot:func] into a valid PCI configuration space access.

If the target is not a host, but a peripheral, configuration space can not be accessed, the peripheral is set up by the host during start up. In complex embedded PCI systems the peripheral may need to access other PCI boards than the host. In such systems a custom (static) configuration of both the host and peripheral may be a convenient solution.

The PCI bus defines four interrupt signals INTA#..INTD#. The interrupt signals must be mapped into a system interrupt/vector, it is up to the BSP or host driver to know the mapping, however the BIOS or boot loader may use the 8-bit read/write “Interrupt Line” register to pass the knowledge along to the OS.

The PCI standard defines and recommends that the backplane route the interupt lines in a systematic way, however in standard there is no such requirement. The PCI Auto Configuration Library implements the recommended way of routing which is very common but it is also supported to some extent to override the interrupt routing from the BSP or Host Bridge driver using the configuration structure.

### 31.2.1 Software Components

The PCI library is located in cpukit/libpci, it consists of different parts:

- PCI Host bridge driver interface
- Configuration routines
- Access (Configuration, I/O and Memory space) routines
- Interrupt routines (implemented by BSP)
- Print routines
- Static/peripheral configuration creation
- PCI shell command

### 31.2.2 PCI Configuration

During start up the PCI bus must be configured in order for host and peripherals to access one another using Memory or I/O accesses and that interrupts are properly handled. Three different spaces are defined and mapped separately:

1. I/O space (IO)
2. non-prefetchable Memory space (MEMIO)
3. prefetchable Memory space (MEM)

Regions of the same type (I/O or Memory) may not overlap which is guaranteed by the software. MEM regions may be mapped into MEMIO regions, but MEMIO regions can not be mapped into MEM, for that could lead to prefetching of registers. The interrupt pin which a board is driving can be read out from PCI configuration space, however it is up to software to know how interrupt signals are routed between PCI-to-PCI bridges and how PCI INT[A..D]# pins are mapped to system IRQ. In systems where previous software (boot loader or BIOS) has already set up this the configuration is overwritten or simply read out.

In order to support different configuration methods the following configuration libraries are selectable by the user:

- Auto Configuration (run Plug & Play software)
- Read Configuration (relies on a boot loader or BIOS)
- Static Configuration (write user defined setup, no Plug & Play)
- Peripheral Configuration (user defined setup, no access to configuration space)

A host driver can be made to support all three configuration methods, or any combination. It may be defined by the BSP which approach is used.

The configuration software is called from the PCI driver (`pci_config_init()`).

Regardless of configuration method a PCI device tree is created in RAM during initialization, the tree can be accessed to find devices and resources without accessing configuration space later on. The user is responsible to create the device tree at compile time when using the static/peripheral method.

#### 31.2.2.1 RTEMS Configuration selection

The active configuration method can be selected at compile time in the same way as other project parameters by including rtems/confdefs.h and setting

- `CONFIGURE_INIT`
- `RTEMS_PCI_CONFIG_LIB`
- `CONFIGURE_PCI_LIB` = PCI_LIB_(AUTO,STATIC,READ,PERIPHERAL)

See the RTEMS configuration section how to setup the PCI library.

#### 31.2.2.2 Auto Configuration

The auto configuration software enumerates PCI buses and initializes all PCI devices found using Plug & Play. The auto configuration software requires that a configuration setup has been registered by the driver or BSP in order to setup the I/O and Memory regions at the correct address ranges. PCI interrupt pins can optionally be routed over PCI-to-PCI bridges and mapped

to a system interrupt number. BAR resources are sorted by size and required alignment, unused "dead" space may be created when PCI bridges are present due to the PCI bridge window size does not equal the alignment. To cope with that resources are reordered to fit smaller BARs into the dead space to minimize the PCI space required. If a BAR or ROM register can not be allocated a PCI address region (due to too few resources available) the register will be given the value of pci_invalid_address which defaults to 0.

The auto configuration routines support:

- PCI 2.3
- Little and big endian PCI bus
- one I/O 16 or 32-bit range (IO)
- memory space (MEMIO)
- prefetchable memory space (MEM), if not present MEM will be mapped into MEMIO
- multiple PCI buses - PCI-to-PCI bridges
- standard BARs, PCI-to-PCI bridge BARs, ROM BARs
- Interrupt routing over bridges
- Interrupt pin to system interrupt mapping

Not supported:

- hot-pluggable devices
- Cardbus bridges
- 64-bit memory space
- 16-bit and 32-bit I/O address ranges at the same time

In PCI 2.3 there may exist I/O BARs that must be located at the low 64kBytes address range, in order to support this the host driver or BSP must make sure that I/O addresses region is within this region.

#### 31.2.2.3 Read Configuration

When a BIOS or boot loader already has setup the PCI bus the configuration can be read directly from the PCI resource registers and buses are already enumerated, this is a much simpler approach than configuring PCI ourselves. The PCI device tree is automatically created based on the current configuration and devices present. After initialization is done there is no difference between the auto or read configuration approaches.

#### 31.2.2.4 Static Configuration

To support custom configurations and small-footprint PCI systems, the user may provide the PCI device tree which contains the current configuration. The PCI buses are enumerated and all resources are written to PCI devices during initialization. When this approach is selected PCI boards must be located at the same slots every time and devices can not be removed or added, Plug & Play is not performed. Boards of the same type may of course be exchanged.

The user can create a configuration by calling pci_cfg_print() on a running system that has had PCI setup by the auto or read configuration routines, it can be called from the PCI shell command. The user must provide the PCI device tree named pci_hb.

#### 31.2.2.5 Peripheral Configuration

On systems where a peripheral PCI device needs to access other PCI devices than the host the peripheral configuration approach may be handy. Most PCI devices answers on the PCI host's requests and start DMA accesses into the Hosts memory, however in some complex systems PCI devices may want to access other devices on the same bus or at another PCI bus.

A PCI peripheral is not allowed to do PCI configuration cycles, which means that it must either rely on the host to give it the addresses it needs, or that the addresses are predefined.

This configuration approach is very similar to the static option, however the configuration is never written to PCI bus, instead it is only used for drivers to find PCI devices and resources using the same PCI API as for the host

### 31.2.3 PCI Access

The PCI access routines are low-level routines provided for drivers, configuration software, etc. in order to access different regions in a way not dependent upon the host driver, BSP or platform.

- PCI configuration space
- PCI I/O space
- Registers over PCI memory space
- Translate PCI address into CPU accessible address and vice versa

By using the access routines drivers can be made portable over different architectures. The access routines take the architecture endianness into consideration and let the host driver or BSP implement I/O space and configuration space access.

Some non-standard hardware may also define the PCI bus big-endian, for example the LEON2 AT697 PCI host bridge and some LEON3 systems may be configured that way. It is up to the BSP to set the appropriate PCI endianness on compile time (`BSP_PCI_BIG_ENDIAN`) in order for inline macros to be correctly defined. Another possibility is to use the function pointers defined by the access layer to implement drivers that support "run-time endianness detection".

#### 31.2.3.1 Configuration space

Configuration space is accessed using the routines listed below. The pci_dev_t type is used to specify a specific PCI bus, device and function. It is up to the host driver or BSP to create a valid access to the requested PCI slot. Requests made to slots that are not supported by hardware should result in `PCISTS_MSTABRT` and/or data must be ignored (writes) or `0xFFFFFFFF` is always returned (reads).

```
/* Configuration Space Access Read Routines */
extern int pci_cfg_r8(pci_dev_t dev, int ofs, uint8_t *data);
extern int pci_cfg_r16(pci_dev_t dev, int ofs, uint16_t *data);
extern int pci_cfg_r32(pci_dev_t dev, int ofs, uint32_t *data);

/* Configuration Space Access Write Routines */
extern int pci_cfg_w8(pci_dev_t dev, int ofs, uint8_t data);
extern int pci_cfg_w16(pci_dev_t dev, int ofs, uint16_t data);
extern int pci_cfg_w32(pci_dev_t dev, int ofs, uint32_t data);
```

#### 31.2.3.2 I/O space

The BSP or driver provide special routines in order to access I/O space. Some architectures have a special instruction accessing I/O space, others have it mapped into a "PCI I/O window" in the standard address space accessed by the CPU. The window size may vary and must be taken into consideration by the host driver. The below routines must be used to access I/O space. The address given to the functions is not the PCI I/O addresses, the caller must have translated PCI I/O addresses (available in the PCI BARs) into a BSP or host driver custom address, see *Access functions* (page 877) for how addresses are translated.

```
/* Read a register over PCI I/O Space */
extern uint8_t pci_io_r8(uint32_t adr);
extern uint16_t pci_io_r16(uint32_t adr);
extern uint32_t pci_io_r32(uint32_t adr);

/* Write a register over PCI I/O Space */
extern void pci_io_w8(uint32_t adr, uint8_t data);
extern void pci_io_w16(uint32_t adr, uint16_t data);
extern void pci_io_w32(uint32_t adr, uint32_t data);
```

#### 31.2.3.3 Registers over Memory space

PCI host bridge hardware normally swap data accesses into the endianness of the host architecture in order to lower the load of the CPU, peripherals can do DMA without swapping. However, the host controller can not separate a standard memory access from a memory access to a register, registers may be mapped into memory space. This leads to register content being swapped, which must be swapped back. The below routines makes it possible to access registers over PCI memory space in a portable way on different architectures, the BSP or architecture must provide necessary functions in order to implement this.

```
static inline uint16_t pci_ld_le16(volatile uint16_t *addr);
static inline void pci_st_le16(volatile uint16_t *addr, uint16_t val);
static inline uint32_t pci_ld_le32(volatile uint32_t *addr);
static inline void pci_st_le32(volatile uint32_t *addr, uint32_t val);
static inline uint16_t pci_ld_be16(volatile uint16_t *addr);
static inline void pci_st_be16(volatile uint16_t *addr, uint16_t val);
static inline uint32_t pci_ld_be32(volatile uint32_t *addr);
static inline void pci_st_be32(volatile uint32_t *addr, uint32_t val);
```

In order to support non-standard big-endian PCI bus the above `pci_*` functions is required, `pci_ld_le16 != ld_le16` on big endian PCI buses.

#### 31.2.3.4 Access functions

The PCI Access Library can provide device drivers with function pointers executing the above Configuration, I/O and Memory space accesses. The functions have the same arguments and return values as the above functions.

The pci_access_func() function defined below can be used to get a function pointer of a specific access type.

```
/* Get Read/Write function for accessing a register over PCI Memory Space
 * (non-inline functions).
```

(continues on next page)

(continued from previous page)

```
 *
 * Arguments
 *  wr              0(Read), 1(Write)
 *  size            1(Byte), 2(Word), 4(Double Word)
 *  func            Where function pointer will be stored
 *  endian          PCI_LITTLE_ENDIAN or PCI_BIG_ENDIAN
 *  type            1(I/O), 3(REG over MEM), 4(CFG)
 *
 * Return
 *  0               Found function
 *  others          No such function defined by host driver or BSP
*/
int pci_access_func(int wr, int size, void **func, int endian, int type);
```

PCI device drivers may be written to support run-time detection of endianess, this is mosly for debugging or for development systems. When the product is finally deployed macros switch to using the inline functions instead which have been configured for the correct endianness.

#### 31.2.3.5 PCI address translation

When PCI addresses, both I/O and memory space, is not mapped 1:1 address translation before access is needed. If drivers read the PCI resources directly using configuration space routines or in the device tree, the addresses given are PCI addresses. The below functions can be used to translate PCI addresses into CPU accessible addresses or vice versa, translation may be different for different PCI spaces/regions.

```
/* Translate PCI address into CPU accessible address */
static inline int pci_pci2cpu(uint32_t *address, int type);

/* Translate CPU accessible address into PCI address (for DMA) */
static inline int pci_cpu2pci(uint32_t *address, int type);
```

### 31.2.4 PCI Interrupt

The PCI specification defines four different interrupt lines INTA#..INTD#, the interrupts are low level sensitive which make it possible to support multiple interrupt sources on the same interrupt line. Since the lines are level sensitive the interrupt sources must be acknowledged before clearing the interrupt contoller, or the interrupt controller must be masked. The BSP must provide a routine for clearing/acknowledging the interrupt controller, it is up to the interrupt service routine to acknowledge the interrupt source.

The PCI Library relies on the BSP for implementing shared interrupt handling through the BSP_PCI_shared_interrupt_* functions/macros, they must be defined when including bsp.h.

PCI device drivers may use the pci_interrupt_* routines in order to call the BSP specific functions in a platform independent way. The PCI interrupt interface has been made similar to the RTEMS IRQ extension so that a BSP can use the standard RTEMS interrupt functions directly.

### 31.2.5 PCI Shell command

The RTEMS shell has a PCI command 'pci' which makes it possible to read/write configuration space, print the current PCI configuration and print out a configuration C-file for the static or peripheral library.

CHAPTER

**THIRTYTWO**

# STACK BOUNDS CHECKER

## 32.1 Introduction

The stack bounds checker is an RTEMS support component that determines if a task has overrun its run-time stack. The routines provided by the stack bounds checker manager are:

- *rtems_stack_checker_is_blown* (page 885) - Has the Current Task Blown its Stack
- *rtems_stack_checker_report_usage* (page 885) - Report Task Stack Usage

## 32.2 Background

### 32.2.1 Task Stack

Each task in a system has a fixed size stack associated with it. This stack is allocated when the task is created. As the task executes, the stack is used to contain parameters, return addresses, saved registers, and local variables. The amount of stack space required by a task is dependent on the exact set of routines used. The peak stack usage reflects the worst case of subroutine pushing information on the stack. For example, if a subroutine allocates a local buffer of 1024 bytes, then this data must be accounted for in the stack of every task that invokes that routine.

Recursive routines make calculating peak stack usage difficult, if not impossible. Each call to the recursive routine consumes *n* bytes of stack space. If the routine recursives 1000 times, then `1000 * n` bytes of stack space are required.

### 32.2.2 Execution

The stack bounds checker operates as a set of task extensions. At task creation time, the task's stack is filled with a pattern to indicate the stack is unused. As the task executes, it will overwrite this pattern in memory. At each task switch, the stack bounds checker's task switch extension is executed. This extension checks that:

- the last `n` bytes of the task's stack have not been overwritten. If this pattern has been damaged, it indicates that at some point since this task was context switch to the CPU, it has used too much stack space.
- the current stack pointer of the task is not within the address range allocated for use as the task's stack.

If either of these conditions is detected, then a blown stack error is reported using the `printk` routine.

The number of bytes checked for an overwrite is processor family dependent. The minimum stack frame per subroutine call varies widely between processor families. On CISC families like the Motorola MC68xxx and Intel ix86, all that is needed is a return address. On more complex RISC processors, the minimum stack frame per subroutine call may include space to save a significant number of registers.

Another processor dependent feature that must be taken into account by the stack bounds checker is the direction that the stack grows. On some processor families, the stack grows up or to higher addresses as the task executes. On other families, it grows down to lower addresses. The stack bounds checker implementation uses the stack description definitions provided by every RTEMS port to get for this information.

## 32.3 Operations

### 32.3.1 Initializing the Stack Bounds Checker

The stack checker is initialized automatically when its task create extension runs for the first time.

The application must include the stack bounds checker extension set in its set of Initial Extensions. This set of extensions is defined as `STACK_CHECKER_EXTENSION`. If using `<rtems/confdefs.h>` for Configuration Table generation, then all that is necessary is to define the macro `CONFIGURE_STACK_CHECKER_ENABLED` before including `<rtems/confdefs.h>` as shown below:

```
#define CONFIGURE_STACK_CHECKER_ENABLED
...
#include <rtems/confdefs.h>
```

### 32.3.2 Checking for Blown Task Stack

The application may check whether the stack pointer of currently executing task is within proper bounds at any time by calling the `rtems_stack_checker_is_blown` method. This method return `FALSE` if the task is operating within its stack bounds and has not damaged its pattern area.

### 32.3.3 Reporting Task Stack Usage

The application may dynamically report the stack usage for every task in the system by calling the `rtems_stack_checker_report_usage` routine. This routine prints a table with the peak usage and stack size of every task in the system. The following is an example of the report generated:

```
ID       NAME        LOW         HIGH      AVAILABLE       USED
0x04010001  IDLE  0x003e8a60  0x003e9667       2952        200
0x08010002  TA1   0x003e5750  0x003e7b57       9096       1168
0x08010003  TA2   0x003e31c8  0x003e55cf       9096       1168
0x08010004  TA3   0x003e0c40  0x003e3047       9096       1104
0xffffffff  INTR  0x003ecfc0  0x003effbf      12160        128
```

Notice the last line. The task id is `0xffffffff` and its name is `INTR`. This is not actually a task, it is the interrupt stack.

### 32.3.4 When a Task Overflows the Stack

When the stack bounds checker determines that a stack overflow has occurred, it will attempt to print a message using `printk` identifying the task and then shut the system down. If the stack overflow has caused corruption, then it is possible that the message cannot be printed.

The following is an example of the output generated:

```
BLOWN STACK!!! Offending task(0x3eb360): id=0x08010002; name=0x54413120
stack covers range 0x003e5750 - 0x003e7b57 (9224 bytes)
Damaged pattern begins at 0x003e5758 and is 128 bytes long
```

The above includes the task id and a pointer to the task control block as well as enough information so one can look at the task's stack and see what was happening.

## 32.4 Routines

This section details the stack bounds checker's routines. A subsection is dedicated to each of routines and describes the calling sequence, related constants, usage, and status codes.

### 32.4.1 STACK_CHECKER_IS_BLOWN - Has Current Task Blown Its Stack

**CALLING SEQUENCE:**

```
bool rtems_stack_checker_is_blown( void );
```

**STATUS CODES:**

| | |
|---|---|
| TRUE | Stack is operating within its stack limits |
| FALSE | Current stack pointer is outside allocated area |

**DESCRIPTION:**
This method is used to determine if the current stack pointer of the currently executing task is within bounds.

**NOTES:**
This method checks the current stack pointer against the high and low addresses of the stack memory allocated when the task was created and it looks for damage to the high water mark pattern for the worst case usage of the task being called.

### 32.4.2 STACK_CHECKER_REPORT_USAGE - Report Task Stack Usage

**CALLING SEQUENCE:**

```
void rtems_stack_checker_report_usage( void );
```

**STATUS CODES:**
NONE

**DESCRIPTION:**
This routine prints a table with the peak stack usage and stack space allocation of every task in the system.

**NOTES:**
NONE

CHAPTER

**THIRTYTHREE**

# CPU USAGE STATISTICS

## 33.1 Introduction

The CPU usage statistics manager is an RTEMS support component that provides a convenient way to manipulate the CPU usage information associated with each task. The routines provided by the CPU usage statistics manager are:

- *rtems_cpu_usage_report* (page 892) - Report CPU Usage Statistics
- *rtems_cpu_usage_reset* (page 893) - Reset CPU Usage Statistics

## 33.2 Background

When analyzing and debugging real-time applications, it is important to be able to know how much CPU time each task in the system consumes. This support component provides a mechanism to easily obtain this information with little burden placed on the target.

The raw data is gathered as part of performing a context switch. RTEMS keeps track of how many clock ticks have occurred which the task being switched out has been executing. If the task has been running less than 1 clock tick, then for the purposes of the statistics, it is assumed to have executed 1 clock tick. This results in some inaccuracy but the alternative is for the task to have appeared to execute 0 clock ticks.

RTEMS versions newer than the 4.7 release series, support the ability to obtain timestamps with nanosecond granularity if the BSP provides support. It is a desirable enhancement to change the way the usage data is gathered to take advantage of this recently added capability. Please consider sponsoring the core RTEMS development team to add this capability.

## 33.3 Operations

### 33.3.1 Report CPU Usage Statistics

The application may dynamically report the CPU usage for every task in the system by calling the `rtems_cpu_usage_report` routine. This routine prints a table with the following information per task:

- task id
- task name
- number of clock ticks executed
- percentage of time consumed by this task

The following is an example of the report generated:

```
+-----------------------------------------------------------------------------+
|CPU USAGE BY THREAD                                                          |
+-----------+---------------------------------------+-------------------------+
|ID         | NAME                                  | SECONDS       | PERCENT |
+-----------+---------------------------------------+---------------+---------+
|0x04010001 | IDLE                                  |             0 |   0.000 |
+-----------+---------------------------------------+---------------+---------+
|0x08010002 | TA1                                   |          1203 |   0.748 |
+-----------+---------------------------------------+---------------+---------+
|0x08010003 | TA2                                   |           203 |   0.126 |
+-----------+---------------------------------------+---------------+---------+
|0x08010004 | TA3                                   |           202 |   0.126 |
+-----------+---------------------------------------+---------------+---------+
|TICKS SINCE LAST SYSTEM RESET:                                          1600 |
|TOTAL UNITS:                                                            1608 |
+-----------------------------------------------------------------------------+
```

Notice that the `TOTAL UNITS` is greater than the ticks per reset. This is an artifact of the way in which RTEMS keeps track of CPU usage. When a task is context switched into the CPU, the number of clock ticks it has executed is incremented. While the task is executing, this number is incremented on each clock tick. Otherwise, if a task begins and completes execution between successive clock ticks, there would be no way to tell that it executed at all.

Another thing to keep in mind when looking at idle time, is that many systems - especially during debug - have a task providing some type of debug interface. It is usually fine to think of the total idle time as being the sum of the `IDLE` task and a debug task that will not be included in a production build of an application.

### 33.3.2 Reset CPU Usage Statistics

Invoking the `rtems_cpu_usage_reset` routine resets the CPU usage statistics for all tasks in the system.

## 33.4 Directives

This section details the CPU usage statistics manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

### 33.4.1 cpu_usage_report - Report CPU Usage Statistics

**CALLING SEQUENCE:**

```
void rtems_cpu_usage_report( void );
```

**STATUS CODES:**
NONE

**DESCRIPTION:**
This routine prints out a table detailing the CPU usage statistics for all tasks in the system.

**NOTES:**
The table is printed using the `printk` routine.

### 33.4.2 cpu_usage_reset - Reset CPU Usage Statistics

**CALLING SEQUENCE:**

```
void rtems_cpu_usage_reset( void );
```

**STATUS CODES:**
NONE

**DESCRIPTION:**
This routine re-initializes the CPU usage statistics for all tasks in the system to their initial state. The initial state is that a task has not executed and thus has consumed no CPU time.

**NOTES:**
NONE

CHAPTER

**THIRTYFOUR**

# OBJECT SERVICES

## 34.1 Introduction

RTEMS provides a collection of services to assist in the management and usage of the objects created and utilized via other managers. These services assist in the manipulation of RTEMS objects independent of the API used to create them. The directives provided by the Object Services are:

- *rtems_build_id()* (page 901) - Builds the object identifier from the API, class, MPCI node, and index components.
- *rtems_build_name()* (page 902) - Builds the object name composed of the four characters.
- *rtems_object_get_classic_name()* (page 903) - Gets the object name associated with the object identifier.
- *rtems_object_get_name()* (page 904) - Gets the object name associated with the object identifier as a string.
- *rtems_object_set_name()* (page 905) - Sets the object name of the object associated with the object identifier.
- *rtems_object_id_get_api()* (page 906) - Gets the API component of the object identifier.
- *rtems_object_id_get_class()* (page 907) - Gets the class component of the object identifier.
- *rtems_object_id_get_node()* (page 908) - Gets the MPCI node component of the object identifier.
- *rtems_object_id_get_index()* (page 909) - Gets the index component of the object identifier.
- *rtems_object_id_api_minimum()* (page 910) - Gets the lowest valid value for the API component of an object identifier.
- *rtems_object_id_api_maximum()* (page 911) - Gets the highest valid value for the API component of an object identifier.
- *rtems_object_api_minimum_class()* (page 912) - Gets the lowest valid class value of the object API.
- *rtems_object_api_maximum_class()* (page 913) - Gets the highest valid class value of the object API.
- *rtems_object_get_api_name()* (page 914) - Gets a descriptive name of the object API.
- *rtems_object_get_api_class_name()* (page 915) - Gets a descriptive name of the object class of the object API.
- *rtems_object_get_class_information()* (page 916) - Gets the object class information of the object class of the object API.
- *rtems_object_get_local_node()* (page 917) - Gets the local MPCI node number.
- *RTEMS_OBJECT_ID_INITIAL()* (page 918) - Builds the object identifier with the lowest index from the API, class, and MPCI node components.

## 34.2 Background

### 34.2.1 APIs

RTEMS implements multiple APIs including an Internal API, the Classic API, and the POSIX API. These APIs share the common foundation of SuperCore objects and thus share object management code. This includes a common scheme for object Ids and for managing object names whether those names be in the thirty-two bit form used by the Classic API or C strings.

The object Id contains a field indicating the API that an object instance is associated with. This field holds a numerically small non-zero integer.

### 34.2.2 Object Classes

Each API consists of a collection of managers. Each manager is responsible for instances of a particular object class. Classic API Tasks and POSIX Mutexes example classes.

The object Id contains a field indicating the class that an object instance is associated with. This field holds a numerically small non-zero integer. In all APIs, a class value of one is reserved for tasks or threads.

### 34.2.3 Object Names

Every RTEMS object which has an Id may also have a name associated with it. Depending on the API, names may be either thirty-two bit integers as in the Classic API or strings as in the POSIX API.

Some objects have Ids but do not have a defined way to associate a name with them. For example, POSIX threads have Ids but per POSIX do not have names. In RTEMS, objects not defined to have thirty-two bit names may have string names assigned to them via the `rtems_object_set_name` service. The original impetus in providing this service was so the normally anonymous POSIX threads could have a user defined name in CPU Usage Reports.

## 34.3 Operations

### 34.3.1 Decomposing and Recomposing an Object Id

Services are provided to decompose an object Id into its subordinate components. The following services are used to do this:

- `rtems_object_id_get_api`
- `rtems_object_id_get_class`
- `rtems_object_id_get_node`
- `rtems_object_id_get_index`

The following C language example illustrates the decomposition of an Id and printing the values.

```
void printObjectId(rtems_id id)
{
    printf(
        "API=%d Class=%" PRIu32 " Node=%" PRIu32 " Index=%" PRIu16 "\n",
        rtems_object_id_get_api(id),
        rtems_object_id_get_class(id),
        rtems_object_id_get_node(id),
        rtems_object_id_get_index(id)
    );
}
```

This prints the components of the Ids as integers.

It is also possible to construct an arbitrary Id using the `rtems_build_id` service. The following C language example illustrates how to construct the "next Id."

```
rtems_id nextObjectId(rtems_id id)
{
    return rtems_build_id(
                rtems_object_id_get_api(id),
                rtems_object_id_get_class(id),
                rtems_object_id_get_node(id),
                rtems_object_id_get_index(id) + 1
            );
}
```

Note that this Id may not be valid in this system or associated with an allocated object.

### 34.3.2 Printing an Object Id

RTEMS also provides services to associate the API and Class portions of an Object Id with strings. This allows the application developer to provide more information about an object in diagnostic messages.

In the following C language example, an Id is decomposed into its constituent parts and "pretty-printed."

```c
void prettyPrintObjectId(rtems_id id)
{
    int tmpAPI;
    uint32_t tmpClass;

    tmpAPI   = rtems_object_id_get_api(id),
    tmpClass = rtems_object_id_get_class(id),

    printf(
        "API=%s Class=%s Node=%" PRIu32 " Index=%" PRIu16 "\n",
        rtems_object_get_api_name(tmpAPI),
        rtems_object_get_api_class_name(tmpAPI, tmpClass),
        rtems_object_id_get_node(id),
        rtems_object_id_get_index(id)
    );
}
```

## 34.4 Directives

This section details the directives of the Object Services. A subsection is dedicated to each of this manager's directives and lists the calling sequence, parameters, description, return values, and notes of the directive.

### 34.4.1 rtems_build_id()

Builds the object identifier from the API, class, MPCI node, and index components.

CALLING SEQUENCE:

```
rtems_id rtems_build_id(
  uint32_t api,
  uint32_t the_class,
  uint32_t node,
  uint32_t index
);
```

PARAMETERS:

**api**
This parameter is the API of the object identifier to build.

**the_class**
This parameter is the class of the object identifier to build.

**node**
This parameter is the MPCI node of the object identifier to build.

**index**
This parameter is the index of the object identifier to build.

RETURN VALUES:

Returns the object identifier built from the API, class, MPCI node, and index components.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.2 rtems_build_name()

Builds the object name composed of the four characters.

CALLING SEQUENCE:

```
rtems_name rtems_build_name( char c1, char c2, char c3, char c4 );
```

PARAMETERS:

**c1**
This parameter is the first character of the name.

**c2**
This parameter is the second character of the name.

**c3**
This parameter is the third character of the name.

**c4**
This parameter is the fourth character of the name.

DESCRIPTION:

This directive takes the four characters provided as arguments and composes a 32-bit object name with `c1` in the most significant 8-bits and `c4` in the least significant 8-bits.

RETURN VALUES:

Returns the object name composed of the four characters.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.3 rtems_object_get_classic_name()

Gets the object name associated with the object identifier.

CALLING SEQUENCE:

```
rtems_status_code rtems_object_get_classic_name(
  rtems_id    id,
  rtems_name *name
);
```

PARAMETERS:

**id**
This parameter is the object identifier to get the name.

**name**
This parameter is the pointer to an *rtems_name* (page 48) object. When the directive call is successful, the object name associated with the object identifier will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `name` parameter was NULL.

**RTEMS_INVALID_ID**
There was no object information available for the object identifier.

**RTEMS_INVALID_ID**
The object name associated with the object identifier was a string.

**RTEMS_INVALID_ID**
There was no object associated with the object identifier.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.4 rtems_object_get_name()

Gets the object name associated with the object identifier as a string.

CALLING SEQUENCE:

```
char *rtems_object_get_name( rtems_id id, size_t length, char *name );
```

PARAMETERS:

**id**
This parameter is the object identifier to get the name.

**length**
This parameter is the buffer length in bytes.

**name**
This parameter is the pointer to a buffer of the specified length.

DESCRIPTION:

The object name is stored in the name buffer. If the name buffer length is greater than zero, then the stored object name will be `NUL` terminated. The stored object name may be truncated to fit the length. There is no indication if a truncation occurred. Every attempt is made to return name as a printable string even if the object has the Classic API 32-bit integer style name.

RETURN VALUES:

**NULL**
The `length` parameter was 0.

**NULL**
The `name` parameter was NULL.

**NULL**
There was no object information available for the object identifier.

**NULL**
There was no object associated with the object identifier.

Returns the `name` parameter value, if there is an object name associated with the object identifier.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.5 rtems_object_set_name()

Sets the object name of the object associated with the object identifier.

CALLING SEQUENCE:

```
rtems_status_code rtems_object_set_name( rtems_id id, const char *name );
```

PARAMETERS:

**id**
This parameter is the object identifier of the object to set the name.

**name**
This parameter is the object name to set.

DESCRIPTION:

This directive will set the object name based upon the user string.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `name` parameter was NULL.

**RTEMS_INVALID_ID**
There was no object information available for the object identifier.

**RTEMS_INVALID_ID**
There was no object associated with the object identifier.

**RTEMS_NO_MEMORY**
There was no memory available to duplicate the name.

NOTES:

This directive can be used to set the name of objects which do not have a naming scheme per their API.

If the object specified by `id` is of a class that has a string name, this directive will free the existing name to the RTEMS Workspace and allocate enough memory from the RTEMS Workspace to make a copy of the string located at `name`.

If the object specified by `id` is of a class that has a 32-bit integer style name, then the first four characters in `name` will be used to construct the name.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within device driver initialization context.
- The directive may be called from within task context.
- The directive may obtain and release the object allocator mutex. This may cause the calling task to be preempted.

### 34.4.6 rtems_object_id_get_api()

Gets the API component of the object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_get_api( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the object identifier with the API component to get.

RETURN VALUES:

Returns the API component of the object identifier.

NOTES:

This directive does not validate the object identifier provided in `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.7 rtems_object_id_get_class()

Gets the class component of the object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_get_class( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the object identifier with the class component to get.

RETURN VALUES:

Returns the class component of the object identifier.

NOTES:

This directive does not validate the object identifier provided in `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.8 rtems_object_id_get_node()

Gets the MPCI node component of the object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_get_node( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the object identifier with the MPCI node component to get.

RETURN VALUES:

Returns the MPCI node component of the object identifier.

NOTES:

This directive does not validate the object identifier provided in `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.9 rtems_object_id_get_index()

Gets the index component of the object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_get_index( rtems_id id );
```

PARAMETERS:

**id**
This parameter is the object identifier with the index component to get.

RETURN VALUES:

Returns the index component of the object identifier.

NOTES:

This directive does not validate the object identifier provided in `id`.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.10 rtems_object_id_api_minimum()

Gets the lowest valid value for the API component of an object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_api_minimum( void );
```

RETURN VALUES:

Returns the lowest valid value for the API component of an object identifier.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.11 rtems_object_id_api_maximum()

Gets the highest valid value for the API component of an object identifier.

CALLING SEQUENCE:

```
int rtems_object_id_api_maximum( void );
```

RETURN VALUES:

Returns the highest valid value for the API component of an object identifier.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive is implemented by a macro and may be called from within C/C++ constant expressions. In addition, a function implementation of the directive exists for bindings to other programming languages.
- The directive will not cause the calling task to be preempted.

### 34.4.12 rtems_object_api_minimum_class()

Gets the lowest valid class value of the object API.

CALLING SEQUENCE:

```
int rtems_object_api_minimum_class( int api );
```

PARAMETERS:

**api**
This parameter is the object API to get the lowest valid class value.

RETURN VALUES:

**-1**
The object API was invalid.

Returns the lowest valid class value of the object API.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.13 rtems_object_api_maximum_class()

Gets the highest valid class value of the object API.

CALLING SEQUENCE:

```
int rtems_object_api_maximum_class( int api );
```

PARAMETERS:

**api**
This parameter is the object API to get the highest valid class value.

RETURN VALUES:

**0**
The object API was invalid.

Returns the highest valid class value of the object API.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.14 rtems_object_get_api_name()

Gets a descriptive name of the object API.

CALLING SEQUENCE:

```
const char *rtems_object_get_api_name( int api );
```

PARAMETERS:

**api**
This parameter is the object API to get the name.

RETURN VALUES:

**"BAD API"**
The API was invalid.

Returns a descriptive name of the API, if the API was valid.

NOTES:

The string returned is from constant space. Do not modify or free it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.15 rtems_object_get_api_class_name()

Gets a descriptive name of the object class of the object API.

CALLING SEQUENCE:

```
const char *rtems_object_get_api_class_name( int the_api, int the_class );
```

PARAMETERS:

**the_api**
This parameter is the object API of the object class.

**the_class**
This parameter is the object class of the object API to get the name.

RETURN VALUES:

**"BAD API"**
The API was invalid.

**"BAD CLASS"**
The class of the API was invalid.

Returns a descriptive name of the class of the API, if the class of the API and the API were valid.

NOTES:

The string returned is from constant space. Do not modify or free it.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.16 rtems_object_get_class_information()

Gets the object class information of the object class of the object API.

CALLING SEQUENCE:

```
rtems_status_code rtems_object_get_class_information(
  int                                 the_api,
  int                                 the_class,
  rtems_object_api_class_information *info
);
```

PARAMETERS:

**the_api**
This parameter is the object API of the object class.

**the_class**
This parameter is the object class of the object API to get the class information.

**info**
This parameter is the pointer to an *rtems_object_api_class_information* (page 48) object. When the directive call is successful, the object class information of the class of the API will be stored in this object.

RETURN VALUES:

**RTEMS_SUCCESSFUL**
The requested operation was successful.

**RTEMS_INVALID_ADDRESS**
The `info` parameter was NULL.

**RTEMS_INVALID_NUMBER**
The class of the API or the API was invalid.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.17 rtems_object_get_local_node()

Gets the local MPCI node number.

CALLING SEQUENCE:

```
uint16_t rtems_object_get_local_node( void );
```

RETURN VALUES:

Returns the local MPCI node number.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

### 34.4.18 RTEMS_OBJECT_ID_INITIAL()

Builds the object identifier with the lowest index from the API, class, and MPCI node components.

CALLING SEQUENCE:

```
rtems_id RTEMS_OBJECT_ID_INITIAL(
  uint32_t api,
  uint32_t class,
  uint32_t node
);
```

PARAMETERS:

**api**
This parameter is the API of the object identifier to build.

**class**
This parameter is the class of the object identifier to build.

**node**
This parameter is the MPCI node of the object identifier to build.

RETURN VALUES:

Returns the object identifier with the lowest index built from the API, class, and MPCI node components.

CONSTRAINTS:

The following constraints apply to this directive:

- The directive may be called from within any runtime context.
- The directive will not cause the calling task to be preempted.

CHAPTER

**THIRTYFIVE**

# CHAINS

## 35.1 Introduction

The Chains API is an interface to the Super Core (score) chain implementation. The Super Core uses chains for all list type functions. This includes wait queues and task queues. The Chains API provided by RTEMS is:

- *rtems_chain_initialize* (page 925) - initialize the chain with nodes
- *rtems_chain_initialize_empty* (page 926) - initialize the chain as empty
- *rtems_chain_is_null_node* (page 927) - Is the node NULL ?
- *rtems_chain_head* (page 928) - Return the chain's head
- *rtems_chain_tail* (page 929) - Return the chain's tail
- *rtems_chain_are_nodes_equal* (page 930) - Are the node's equal ?
- *rtems_chain_is_empty* (page 931) - Is the chain empty ?
- *rtems_chain_is_first* (page 932) - Is the Node the first in the chain ?
- *rtems_chain_is_last* (page 933) - Is the Node the last in the chain ?
- *rtems_chain_has_only_one_node* (page 934) - Does the node have one node ?
- *rtems_chain_node_count_unprotected* (page 935) - Returns the node count of the chain (unprotected)
- *rtems_chain_is_head* (page 936) - Is the node the head ?
- *rtems_chain_is_tail* (page 937) - Is the node the tail ?
- *rtems_chain_extract* (page 938) - Extract the node from the chain
- *rtems_chain_extract_unprotected* (page 939) - Extract the node from the chain (unprotected)
- *rtems_chain_get* (page 940) - Return the first node on the chain
- *rtems_chain_get_unprotected* (page 941) - Return the first node on the chain (unprotected)
- *rtems_chain_insert* (page 942) - Insert the node into the chain
- *rtems_chain_insert_unprotected* (page 943) - Insert the node into the chain (unprotected)
- *rtems_chain_append* (page 944) - Append the node to chain
- *rtems_chain_append_unprotected* (page 945) - Append the node to chain (unprotected)
- *rtems_chain_prepend* (page 946) - Prepend the node to the end of the chain
- *rtems_chain_prepend_unprotected* (page 947) - Prepend the node to chain (unprotected)

## 35.2 Background

The Chains API maps to the Super Core Chains API. Chains are implemented as a double linked list of nodes anchored to a control node. The list starts at the control node and is terminated at the control node. A node has previous and next pointers. Being a double linked list nodes can be inserted and removed without the need to travse the chain.

Chains have a small memory footprint and can be used in interrupt service routines and are thread safe in a multi-threaded environment. The directives list which operations disable interrupts.

Chains are very useful in Board Support packages and applications.

### 35.2.1 Nodes

A chain is made up from nodes that orginate from a chain control object. A node is of type `rtems_chain_node`. The node is designed to be part of a user data structure and a cast is used to move from the node address to the user data structure address. For example:

```
typedef struct foo
{
    rtems_chain_node node;
    int              bar;
} foo;
```

creates a type `foo` that can be placed on a chain. To get the foo structure from the list you perform the following:

```
foo* get_foo(rtems_chain_control* control)
{
    return (foo*) rtems_chain_get(control);
}
```

The node is placed at the start of the user's structure to allow the node address on the chain to be easly cast to the user's structure address.

### 35.2.2 Controls

A chain is anchored with a control object. Chain control provide the user with access to the nodes on the chain. The control is head of the node.

```
[Control]
first ------------------------->
permanent_null <---------------- [NODE]
last -------------------------->
```

The implementation does not require special checks for manipulating the first and last nodes on the chain. To accomplish this the `rtems_chain_control` structure is treated as two overlapping `rtems_chain_node` structures. The permanent head of the chain overlays a node structure on the first and `permanent_null` fields. The `permanent_tail` of the chain overlays a node structure on the `permanent_null` and `last` elements of the structure.

## 35.3 Operations

### 35.3.1 Multi-threading

Chains are designed to be used in a multi-threading environment. The directives list which operations mask interrupts. Chains supports tasks and interrupt service routines appending and extracting nodes with out the need for extra locks. Chains how-ever cannot insure the integrity of a chain for all operations. This is the responsibility of the user. For example an interrupt service routine extracting nodes while a task is iterating over the chain can have unpredictable results.

### 35.3.2 Creating a Chain

To create a chain you need to declare a chain control then add nodes to the control. Consider a user structure and chain control:

```
typedef struct foo
{
    rtems_chain_node node;
    char*            data;
} foo;
rtems_chain_control chain;
```

Add nodes with the following code:

```
rtems_chain_initialize_empty (&chain);

for (i = 0; i < count; i++)
{
    foo* bar = malloc (sizeof (foo));
    if (!bar)
        return -1;
    bar->data = malloc (size);
    rtems_chain_append (&chain, &bar->node);
}
```

The chain is initialized and the nodes allocated and appended to the chain. This is an example of a pool of buffers.

### 35.3.3 Iterating a Chain

Iterating a chain is a common function. The example shows how to iterate the buffer pool chain created in the last section to find buffers starting with a specific string. If the buffer is located it is extracted from the chain and placed on another chain:

```
void foobar (const char*          match,
             rtems_chain_control* chain,
             rtems_chain_control* out)
{
    rtems_chain_node* node;
    foo*              bar;

```

(continues on next page)

(continued from previous page)

```
    rtems_chain_initialize_empty (out);

    node = rtems_chain_first (chain);

    while (!rtems_chain_is_tail (chain, node))
    {
        bar = (foo*) node;
        rtems_chain_node* next_node = rtems_chain_next(node);
        if (strcmp (match, bar->data) == 0)
        {
            rtems_chain_extract (node);
            rtems_chain_append (out, node);
        }
        node = next_node;
    }
}
```

## 35.4 Directives

The section details the Chains directives.

### 35.4.1 Initialize Chain With Nodes

**CALLING SEQUENCE:**

```
void rtems_chain_initialize(
    rtems_chain_control *the_chain,
    void                *starting_address,
    size_t               number_nodes,
    size_t               node_size
)
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This function take in a pointer to a chain control and initializes it to contain a set of chain nodes. The chain will contain `number_nodes` chain nodes from the memory pointed to by `start_address`. Each node is assumed to be `node_size` bytes.

**NOTES:**
This call will discard any nodes on the chain.

This call does NOT inititialize any user data on each node.

### 35.4.2 Initialize Empty

**CALLING SEQUENCE:**

```
void rtems_chain_initialize_empty(
    rtems_chain_control *the_chain
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This function take in a pointer to a chain control and initializes it to empty.

**NOTES:**
This call will discard any nodes on the chain.

### 35.4.3 Is Null Node ?

**CALLING SEQUENCE:**

```
bool rtems_chain_is_null_node(
    const rtems_chain_node *the_node
);
```

**RETURNS:**
Returns `true` is the node point is NULL and `false` if the node is not NULL.

**DESCRIPTION:**
Tests the node to see if it is a NULL returning `true` if a null.

### 35.4.4 Head

**CALLING SEQUENCE:**

```
rtems_chain_node *rtems_chain_head(
    rtems_chain_control *the_chain
)
```

**RETURNS:**
Returns the permanent head node of the chain.

**DESCRIPTION:**
This function returns a pointer to the first node on the chain.

### 35.4.5 Tail

**CALLING SEQUENCE:**

```
rtems_chain_node *rtems_chain_tail(
    rtems_chain_control *the_chain
);
```

**RETURNS:**
Returns the permanent tail node of the chain.

**DESCRIPTION:**
This function returns a pointer to the last node on the chain.

### 35.4.6 Are Two Nodes Equal ?

**CALLING SEQUENCE:**

```
bool rtems_chain_are_nodes_equal(
    const rtems_chain_node *left,
    const rtems_chain_node *right
);
```

**RETURNS:**

This function returns `true` if the left node and the right node are equal, and `false` otherwise.

**DESCRIPTION:**

This function returns `true` if the left node and the right node are equal, and `false` otherwise.

### 35.4.7 Is the Chain Empty

**CALLING SEQUENCE:**

```
bool rtems_chain_is_empty(
    rtems_chain_control *the_chain
);
```

**RETURNS:**
This function returns `true` if there a no nodes on the chain and `false` otherwise.

**DESCRIPTION:**
This function returns `true` if there a no nodes on the chain and `false` otherwise.

### 35.4.8 Is this the First Node on the Chain ?

**CALLING SEQUENCE:**

```
bool rtems_chain_is_first(
    const rtems_chain_node *the_node
);
```

**RETURNS:**
This function returns `true` if the node is the first node on a chain and `false` otherwise.

**DESCRIPTION:**
This function returns `true` if the node is the first node on a chain and `false` otherwise.

### 35.4.9 Is this the Last Node on the Chain ?

**CALLING SEQUENCE:**

```
bool rtems_chain_is_last(
    const rtems_chain_node *the_node
);
```

**RETURNS:**
This function returns `true` if the node is the last node on a chain and `false` otherwise.

**DESCRIPTION:**
This function returns `true` if the node is the last node on a chain and `false` otherwise.

### 35.4.10 Does this Chain have only One Node ?

**CALLING SEQUENCE:**

```
bool rtems_chain_has_only_one_node(
    const rtems_chain_control *the_chain
);
```

**RETURNS:**
This function returns `true` if there is only one node on the chain and `false` otherwise.

**DESCRIPTION:**
This function returns `true` if there is only one node on the chain and `false` otherwise.

### 35.4.11 Returns the node count of the chain (unprotected)

**CALLING SEQUENCE:**

```
size_t rtems_chain_node_count_unprotected(
    const rtems_chain_control *the_chain
);
```

**RETURNS:**
This function returns the node count of the chain.

**DESCRIPTION:**
This function returns the node count of the chain.

### 35.4.12 Is this Node the Chain Head ?

**CALLING SEQUENCE:**

```
bool rtems_chain_is_head(
    rtems_chain_control    *the_chain,
    rtems_const chain_node *the_node
);
```

**RETURNS:**

This function returns `true` if the node is the head of the chain and `false` otherwise.

**DESCRIPTION:**

This function returns `true` if the node is the head of the chain and `false` otherwise.

### 35.4.13 Is this Node the Chain Tail ?

**CALLING SEQUENCE:**

```
bool rtems_chain_is_tail(
    rtems_chain_control    *the_chain,
    const rtems_chain_node *the_node
)
```

**RETURNS:**

This function returns `true` if the node is the tail of the chain and `false` otherwise.

**DESCRIPTION:**

This function returns `true` if the node is the tail of the chain and `false` otherwise.

### 35.4.14 Extract a Node

**CALLING SEQUENCE:**

```
void rtems_chain_extract(
    rtems_chain_node *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine extracts the node from the chain on which it resides.

**NOTES:**
Interrupts are disabled while extracting the node to ensure the atomicity of the operation.

Use `rtems_chain_extract_unprotected` to avoid disabling of interrupts.

### 35.4.15 Extract a Node (unprotected)

**CALLING SEQUENCE:**

```
void rtems_chain_extract_unprotected(
    rtems_chain_node *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine extracts the node from the chain on which it resides.

**NOTES:**
The function does nothing to ensure the atomicity of the operation.

### 35.4.16 Get the First Node

**CALLING SEQUENCE:**

```
rtems_chain_node *rtems_chain_get(
    rtems_chain_control *the_chain
);
```

**RETURNS:**

Returns a pointer a node. If a node was removed, then a pointer to that node is returned. If the chain was empty, then `NULL` is returned.

**DESCRIPTION:**

This function removes the first node from the chain and returns a pointer to that node. If the chain is empty, then `NULL` is returned.

**NOTES:**

Interrupts are disabled while obtaining the node to ensure the atomicity of the operation.

Use `rtems_chain_get_unprotected()` to avoid disabling of interrupts.

### 35.4.17 Get the First Node (unprotected)

**CALLING SEQUENCE:**

```
rtems_chain_node *rtems_chain_get_unprotected(
    rtems_chain_control *the_chain
);
```

**RETURNS:**
A pointer to the former first node is returned.

**DESCRIPTION:**
Removes the first node from the chain and returns a pointer to it. In case the chain was empty, then the results are unpredictable.

**NOTES:**
The function does nothing to ensure the atomicity of the operation.

### 35.4.18 Insert a Node

**CALLING SEQUENCE:**

```
void rtems_chain_insert(
    rtems_chain_node *after_node,
    rtems_chain_node *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine inserts a node on a chain immediately following the specified node.

**NOTES:**
Interrupts are disabled during the insert to ensure the atomicity of the operation.

Use `rtems_chain_insert_unprotected()` to avoid disabling of interrupts.

### 35.4.19 Insert a Node (unprotected)

**CALLING SEQUENCE:**

```
void rtems_chain_insert_unprotected(
    rtems_chain_node *after_node,
    rtems_chain_node *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine inserts a node on a chain immediately following the specified node.

**NOTES:**
The function does nothing to ensure the atomicity of the operation.

### 35.4.20 Append a Node

**CALLING SEQUENCE:**

```
void rtems_chain_append(
    rtems_chain_control *the_chain,
    rtems_chain_node    *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine appends a node to the end of a chain.

**NOTES:**
Interrupts are disabled during the append to ensure the atomicity of the operation.

Use `rtems_chain_append_unprotected` to avoid disabling of interrupts.

### 35.4.21 Append a Node (unprotected)

**CALLING SEQUENCE:**

```
void rtems_chain_append_unprotected(
    rtems_chain_control *the_chain,
    rtems_chain_node    *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine appends a node to the end of a chain.

**NOTES:**
The function does nothing to ensure the atomicity of the operation.

### 35.4.22 Prepend a Node

**CALLING SEQUENCE:**

```
void rtems_chain_prepend(
    rtems_chain_control *the_chain,
    rtems_chain_node    *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine prepends a node to the front of the chain.

**NOTES:**
Interrupts are disabled during the prepend to ensure the atomicity of the operation.

Use `rtems_chain_prepend_unprotected` to avoid disabling of interrupts.

### 35.4.23 Prepend a Node (unprotected)

**CALLING SEQUENCE:**

```
void rtems_chain_prepend_unprotected(
    rtems_chain_control *the_chain,
    rtems_chain_node    *the_node
);
```

**RETURNS:**
Returns nothing.

**DESCRIPTION:**
This routine prepends a node to the front of the chain.

**NOTES:**
The function does nothing to ensure the atomicity of the operation.

CHAPTER

**THIRTYSIX**

# RED-BLACK TREES

## 36.1 Introduction

The Red-Black Tree API is an interface to the SuperCore (score) rbtree implementation. Within RTEMS, red-black trees are used when a binary search tree is needed, including dynamic priority thread queues and non-contiguous heap memory. The Red-Black Tree API provided by RTEMS is:

- `rtems_rtems_rbtree_node` - Red-Black Tree node embedded in another struct
- `rtems_rtems_rbtree_control` - Red-Black Tree control node for an entire tree
- `rtems_rtems_rbtree_initialize` - initialize the red-black tree with nodes
- `rtems_rtems_rbtree_initialize_empty` - initialize the red-black tree as empty
- `rtems_rtems_rbtree_set_off_tree` - Clear a node's links
- `rtems_rtems_rbtree_root` - Return the red-black tree's root node
- `rtems_rtems_rbtree_min` - Return the red-black tree's minimum node
- `rtems_rtems_rbtree_max` - Return the red-black tree's maximum node
- `rtems_rtems_rbtree_left` - Return a node's left child node
- `rtems_rtems_rbtree_right` - Return a node's right child node
- `rtems_rtems_rbtree_parent` - Return a node's parent node
- `rtems_rtems_rbtree_are_nodes_equal` - Are the node's equal ?
- `rtems_rtems_rbtree_is_empty` - Is the red-black tree empty ?
- `rtems_rtems_rbtree_is_min` - Is the Node the minimum in the red-black tree ?
- `rtems_rtems_rbtree_is_max` - Is the Node the maximum in the red-black tree ?
- `rtems_rtems_rbtree_is_root` - Is the Node the root of the red-black tree ?
- `rtems_rtems_rbtree_find` - Find the node with a matching key in the red-black tree
- `rtems_rtems_rbtree_predecessor` - Return the in-order predecessor of a node.
- `rtems_rtems_rbtree_successor` - Return the in-order successor of a node.
- `rtems_rtems_rbtree_extract` - Remove the node from the red-black tree
- `rtems_rtems_rbtree_get_min` - Remove the minimum node from the red-black tree
- `rtems_rtems_rbtree_get_max` - Remove the maximum node from the red-black tree
- `rtems_rtems_rbtree_peek_min` - Returns the minimum node from the red-black tree
- `rtems_rtems_rbtree_peek_max` - Returns the maximum node from the red-black tree
- `rtems_rtems_rbtree_insert` - Add the node to the red-black tree

## 36.2 Background

The Red-Black Trees API is a thin layer above the SuperCore Red-Black Trees implementation. A Red-Black Tree is defined by a control node with pointers to the root, minimum, and maximum nodes in the tree. Each node in the tree consists of a parent pointer, two children pointers, and a color attribute. A tree is parameterized as either unique, meaning identical keys are rejected, or not, in which case duplicate keys are allowed.

Users must provide a comparison functor that gets passed to functions that need to compare nodes. In addition, no internal synchronization is offered within the red-black tree implementation, thus users must ensure at most one thread accesses a red-black tree instance at a time.

### 36.2.1 Nodes

A red-black tree is made up from nodes that orginate from a red-black tree control object. A node is of type `rtems_rtems_rbtree_node`. The node is designed to be part of a user data structure. To obtain the encapsulating structure users can use the `RTEMS_CONTAINER_OF` macro. The node can be placed anywhere within the user's structure and the macro will calculate the structure's address from the node's address.

### 36.2.2 Controls

A red-black tree is rooted with a control object. Red-Black Tree control provide the user with access to the nodes on the red-black tree. The implementation does not require special checks for manipulating the root of the red-black tree. To accomplish this the `rtems_rtems_rbtree_control` structure is treated as a `rtems_rtems_rbtree_node` structure with a `NULL` parent and left child pointing to the root.

## 36.3 Operations

Examples for using the red-black trees can be found in the `testsuites/sptests/sprbtree01/init.c` file.

## 36.4 Directives

### 36.4.1 Documentation for the Red-Black Tree Directives

Source documentation for the Red-Black Tree API can be found in the generated Doxygen output for `cpukit/sapi`.

CHAPTER

**THIRTYSEVEN**

# TIMESPEC HELPERS

## 37.1 Introduction

The Timespec helpers manager provides directives to assist in manipulating instances of the POSIX `struct timespec` structure.

The directives provided by the timespec helpers manager are:

- *rtems_timespec_set* (page 960) - Set timespec's value
- *rtems_timespec_zero* (page 961) - Zero timespec's value
- *rtems_timespec_is_valid* (page 962) - Check if timespec is valid
- *rtems_timespec_add_to* (page 963) - Add two timespecs
- *rtems_timespec_subtract* (page 964) - Subtract two timespecs
- *rtems_timespec_divide* (page 965) - Divide two timespecs
- *rtems_timespec_divide_by_integer* (page 966) - Divide timespec by integer
- *rtems_timespec_less_than* (page 967) - Less than operator
- *rtems_timespec_greater_than* (page 968) - Greater than operator
- *rtems_timespec_equal_to* (page 969) - Check if two timespecs are equal
- *rtems_timespec_get_seconds* (page 970) - Obtain seconds portion of timespec
- *rtems_timespec_get_nanoseconds* (page 971) - Obtain nanoseconds portion of timespec
- *rtems_timespec_to_ticks* (page 972) - Convert timespec to number of ticks
- *rtems_timespec_from_ticks* (page 973) - Convert ticks to timespec

## 37.2 Background

### 37.2.1 Time Storage Conventions

Time can be stored in many ways. One of them is the `struct timespec` format which is a structure that consists of the fields `tv_sec` to represent seconds and `tv_nsec` to represent nanoseconds. The``struct timeval`` structure is simular and consists of seconds (stored in `tv_sec`) and microseconds (stored in `tv_usec`). Either``struct timespec`` or `struct timeval` can be used to represent elapsed time, time of executing some operations, or time of day.

## 37.3 Operations

### 37.3.1 Set and Obtain Timespec Value

A user may write a specific time by passing the desired seconds and nanoseconds values and the destination `struct timespec` using the `rtems_timespec_set` directive.

The `rtems_timespec_zero` directive is used to zero the seconds and nanoseconds portions of a `struct timespec` instance.

Users may obtain the seconds or nanoseconds portions of a `struct timespec` instance with the `rtems_timespec_get_seconds` or `rtems_timespec_get_nanoseconds` methods, respectively.

### 37.3.2 Timespec Math

A user can perform multiple operations on `struct timespec` instances. The helpers in this manager assist in adding, subtracting, and performing divison on `struct timespec` instances.

- Adding two `struct timespec` can be done using the `rtems_timespec_add_to` directive. This directive is used mainly to calculate total amount of time consumed by multiple operations.
- The `rtems_timespec_subtract` is used to subtract two `struct timespecs` instances and determine the elapsed time between those two points in time.
- The `rtems_timespec_divide` is used to use to divide one `struct timespec` instance by another. This calculates the percentage with a precision to three decimal points.
- The `rtems_timespec_divide_by_integer` is used to divide a `struct timespec` instance by an integer. It is commonly used in benchmark calculations to dividing duration by the number of iterations performed.

### 37.3.3 Comparing struct timespec Instances

A user can compare two `struct timespec` instances using the `rtems_timespec_less_than`, `rtems_timespec_greater_than` or `rtems_timespec_equal_to` routines.

### 37.3.4 Conversions and Validity Check

Conversion to and from clock ticks may be performed by using the `rtems_timespec_to_ticks` and `rtems_timespec_from_ticks` directives.

User can also check validity of timespec with `rtems_timespec_is_valid` routine.

## 37.4 Directives

This section details the Timespec Helpers manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

### 37.4.1 TIMESPEC_SET - Set struct timespec Instance

**CALLING SEQUENCE:**

```
void rtems_timespec_set(
    struct timespec *time,
    time_t           seconds,
    uint32_t         nanoseconds
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This directive sets the `struct timespec` *time* to the desired `seconds` and `nanoseconds` values.

**NOTES:**

This method does NOT check if `nanoseconds` is less than the maximum number of nanoseconds in a second.

### 37.4.2 TIMESPEC_ZERO - Zero struct timespec Instance

**CALLING SEQUENCE:**

```
void rtems_timespec_zero(
    struct timespec *time
);
```

**DIRECTIVE STATUS CODES:**
NONE

**DESCRIPTION:**
This routine sets the contents of the `struct timespec` instance `time` to zero.

**NOTES:**
NONE

### 37.4.3 TIMESPEC_IS_VALID - Check validity of a struct timespec instance

**CALLING SEQUENCE:**

```
bool rtems_timespec_is_valid(
    const struct timespec *time
);
```

**DIRECTIVE STATUS CODES:**

This method returns `true` if the instance is valid, and `false` otherwise.

**DESCRIPTION:**

This routine check validity of a `struct timespec` instance. It checks if the nanoseconds portion of the `struct timespec` instanceis in allowed range (less than the maximum number of nanoseconds per second).

**NOTES:**

NONE

### 37.4.4 TIMESPEC_ADD_TO - Add Two struct timespec Instances

**CALLING SEQUENCE:**

```
uint32_t rtems_timespec_add_to(
    struct timespec       *time,
    const struct timespec *add
);
```

**DIRECTIVE STATUS CODES:**

The method returns the number of seconds `time` increased by.

**DESCRIPTION:**

This routine adds two `struct timespec` instances. The second argument is added to the first. The parameter `time` is the base time to which the `add` parameter is added.

**NOTES:**

NONE

### 37.4.5 TIMESPEC_SUBTRACT - Subtract Two struct timespec Instances

**CALLING SEQUENCE:**

```
void rtems_timespec_subtract(
    const struct timespec *start,
    const struct timespec *end,
    struct timespec       *result
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This routine subtracts `start` from `end` saves the difference in `result`. The primary use of this directive is to calculate elapsed time.

**NOTES:**

It is possible to subtract when `end` is less than `start` and it produce negative `result`. When doing this you should be careful and remember that only the seconds portion of a `struct timespec` instance is signed, which means that nanoseconds portion is always increasing. Due to that when your timespec has seconds = -1 and nanoseconds = 500,000,000 it means that result is -0.5 second, NOT the expected -1.5!

### 37.4.6 TIMESPEC_DIVIDE - Divide Two struct timespec Instances

**CALLING SEQUENCE:**

```
void rtems_timespec_divide(
    const struct timespec *lhs,
    const struct timespec *rhs,
    uint32_t              *ival_percentage,
    uint32_t              *fval_percentage
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This routine divides the `struct timespec` instance `lhs` by the `struct timespec` instance `rhs`. The result is returned in the `ival_percentage` and `fval_percentage`, representing the integer and fractional results of the division respectively.

The `ival_percentage` is integer value of calculated percentage and `fval_percentage` is fractional part of calculated percentage.

**NOTES:**

The intended use is calculating percentges to three decimal points.

When dividing by zero, this routine return both `ival_percentage` and `fval_percentage` equal zero.

The division is performed using exclusively integer operations.

### 37.4.7 TIMESPEC_DIVIDE_BY_INTEGER - Divide a struct timespec Instance by an Integer

**CALLING SEQUENCE:**

```
int rtems_timespec_divide_by_integer(
    const struct timespec *time,
    uint32_t               iterations,
    struct timespec       *result
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This routine divides the `struct timespec` instance `time` by the integer value `iterations`. The result is saved in `result`.

**NOTES:**

The expected use is to assist in benchmark calculations where you typically divide a duration (`time`) by a number of iterations what gives average time.

### 37.4.8 TIMESPEC_LESS_THAN - Less than operator

**CALLING SEQUENCE:**

```
bool rtems_timespec_less_than(
    const struct timespec *lhs,
    const struct timespec *rhs
);
```

**DIRECTIVE STATUS CODES:**

This method returns `struct true` if `lhs` is less than `rhs` and `struct false` otherwise.

**DESCRIPTION:**

This method is the less than operator for `struct timespec` instances. The first parameter is the left hand side and the second is the right hand side of the comparison.

**NOTES:**

NONE

### 37.4.9 TIMESPEC_GREATER_THAN - Greater than operator

**CALLING SEQUENCE:**

```
bool rtems_timespec_greater_than(
    const struct timespec *_lhs,
    const struct timespec *_rhs
);
```

**DIRECTIVE STATUS CODES:**

This method returns `struct true` if `lhs` is greater than `rhs` and `struct false` otherwise.

**DESCRIPTION:**

This method is greater than operator for `struct timespec` instances.

**NOTES:**

NONE

### 37.4.10 TIMESPEC_EQUAL_TO - Check equality of timespecs

**CALLING SEQUENCE:**

```
bool rtems_timespec_equal_to(
    const struct timespec *lhs,
    const struct timespec *rhs
);
```

**DIRECTIVE STATUS CODES:**

This method returns `struct true` if `lhs` is equal to `rhs` and `struct false` otherwise.

**DESCRIPTION:**

This method is equality operator for `struct timespec` instances.

**NOTES:**

NONE

### 37.4.11 TIMESPEC_GET_SECONDS - Get Seconds Portion of struct timespec Instance

**CALLING SEQUENCE:**

```
time_t rtems_timespec_get_seconds(
    struct timespec *time
);
```

**DIRECTIVE STATUS CODES:**

This method returns the seconds portion of the specified `struct timespec` instance.

**DESCRIPTION:**

This method returns the seconds portion of the specified `struct timespec` instance `time`.

**NOTES:**

NONE

### 37.4.12 TIMESPEC_GET_NANOSECONDS - Get Nanoseconds Portion of the struct timespec Instance

**CALLING SEQUENCE:**

```
uint32_t rtems_timespec_get_nanoseconds(
    struct timespec *_time
);
```

**DIRECTIVE STATUS CODES:**

This method returns the nanoseconds portion of the specified `struct timespec` instance.

**DESCRIPTION:**

This method returns the nanoseconds portion of the specified timespec which is pointed by `_time`.

**NOTES:**

NONE

### 37.4.13 TIMESPEC_TO_TICKS - Convert struct timespec Instance to Ticks

**CALLING SEQUENCE:**

```
uint32_t rtems_timespec_to_ticks(
    const struct timespec *time
);
```

**DIRECTIVE STATUS CODES:**
This directive returns the number of ticks computed.

**DESCRIPTION:**
This directive converts the `time` timespec to the corresponding number of clock ticks.

**NOTES:**
NONE

### 37.4.14 TIMESPEC_FROM_TICKS - Convert Ticks to struct timespec Representation

**CALLING SEQUENCE:**

```
void rtems_timespec_from_ticks(
    uint32_t         ticks,
    struct timespec *time
);
```

**DIRECTIVE STATUS CODES:**

NONE

**DESCRIPTION:**

This routine converts the `ticks` to the corresponding `struct timespec` representation and stores it in `time`.

**NOTES:**

NONE

CHAPTER

**THIRTYEIGHT**

# CONSTANT BANDWIDTH SERVER SCHEDULER API

## 38.1 Introduction

Unlike simple schedulers, the Constant Bandwidth Server (CBS) requires a special API for tasks to indicate their scheduling parameters. The directives provided by the CBS API are:

- *rtems_cbs_initialize* (page 982) - Initialize the CBS library
- *rtems_cbs_cleanup* (page 983) - Cleanup the CBS library
- *rtems_cbs_create_server* (page 984) - Create a new bandwidth server
- *rtems_cbs_attach_thread* (page 985) - Attach a thread to server
- *rtems_cbs_detach_thread* (page 986) - Detach a thread from server
- *rtems_cbs_destroy_server* (page 987) - Destroy a bandwidth server
- *rtems_cbs_get_server_id* (page 988) - Get an ID of a server
- *rtems_cbs_get_parameters* (page 989) - Get scheduling parameters of a server
- *rtems_cbs_set_parameters* (page 990) - Set scheduling parameters of a server
- *rtems_cbs_get_execution_time* (page 991) - Get elapsed execution time
- *rtems_cbs_get_remaining_budget* (page 992) - Get remainig execution time
- *rtems_cbs_get_approved_budget* (page 993) - Get scheduler approved execution time

## 38.2 Background

### 38.2.1 Constant Bandwidth Server Definitions

The Constant Bandwidth Server API enables tasks to communicate with the scheduler and indicate its scheduling parameters. The scheduler has to be set up first (by defining `CONFIGURE_SCHEDULER_CBS` macro).

The difference to a plain EDF is the presence of servers. It is a budget aware extention of the EDF scheduler, therefore, tasks attached to servers behave in a similar way as with EDF unless they exceed their budget.

The intention of servers is reservation of a certain computation time (budget) of the processor for all subsequent periods. The structure `rtems_cbs_parameters` determines the behavior of a server. It contains `deadline` which is equal to period, and `budget` which is the time the server is allowed to spend on CPU per each period. The ratio between those two parameters yields the maximum percentage of the CPU the server can use (bandwidth). Moreover, thanks to this limitation the overall utilization of CPU is under control, and the sum of bandwidths of all servers in the system yields the overall reserved portion of processor. The rest is still available for ordinary tasks that are not attached to any server.

In order to make the server effective to the executing tasks, tasks have to be attached to the servers. The `rtems_cbs_server_id` is a type denoting an id of a server and `rtems_id` a type for id of tasks. .. index:: CBS periodic tasks

### 38.2.2 Handling Periodic Tasks

Each task's execution begins with a default background priority (see the chapter Scheduling Concepts to understand the concept of priorities in EDF). Once you decide the tasks should start periodic execution, you have two possibilities. Either you use only the Rate Monotonic manager which takes care of periodic behavior, or you declare deadline and budget using the CBS API in which case these properties are constant for all subsequent periods, unless you change them using the CBS API again. Task now only has to indicate and end of each period using `rtems_rate_monotonic_period`. .. index:: CBS overrun handler

### 38.2.3 Registering a Callback Function

In case tasks attached to servers are not aware of their execution time and happen to exceed it, the scheduler does not guarantee execution any more and pulls the priority of the task to background, which would possibly lead to immediate preemption (if there is at least one ready task with a higher pirority). However, the task is not blocked but a callback function is invoked. The callback function (`rtems_cbs_budget_overrun`) might be optionally registered upon a server creation (`rtems_cbs_create_server`).

This enables the user to define what should happen in case of budget overrun. There is obviously no space for huge operations because the priority is down and not real time any more, however, you still can at least in release resources for other tasks, restart the task or log an error information. Since the routine is called directly from kernel, use `printk()` instead of `printf()`.

The calling convention of the callback function is:

```
void overrun_handler(
    rtems_cbs_server_id server_id
);
```

### 38.2.4 Limitations

When using this scheduler you have to keep in mind several things:

- it_limitations
- In the current implementation it is possible to attach only a single task to each server.
- If you have a task attached to a server and you voluntatily block it in the beginning of its execution, its priority will be probably pulled to background upon unblock, thus not guaranteed deadline any more. This is because you are effectively raising computation time of the task. When unbocking, you should be always sure that the ratio between remaining computation time and remaining deadline is not higher that the utilization you have agreed with the scheduler.

## 38.3 Operations

### 38.3.1 Setting up a server

The directive `rtems_cbs_create_server` is used to create a new server that is characterized by `rtems_cbs_parameters`. You also might want to register the `rtems_cbs_budget_overrun` callback routine. After this step tasks can be attached to the server. The directive `rtems_cbs_set_parameters` can change the scheduling parameters to avoid destroying and creating a new server again.

### 38.3.2 Attaching Task to a Server

If a task is attached to a server using `rtems_cbs_attach_thread`, the task's computation time per period is limited by the server and the deadline (period) of task is equal to deadline of the server which means if you conclude a period using `rate_monotonic_period`, the length of next period is always determined by the server's property.

The task has a guaranteed bandwidth given by the server but should not exceed it, otherwise the priority is pulled to background until the start of next period and the `rtems_cbs_budget_overrun` callback function is invoked.

When attaching a task to server, the preemptability flag of the task is raised, otherwise it would not be possible to control the execution of the task.

### 38.3.3 Detaching Task from a Server

The directive `rtems_cbs_detach_thread` is just an inverse operation to the previous one, the task continues its execution with the initial priority.

Preemptability of the task is restored to the initial value.

### 38.3.4 Examples

The following example presents a simple common use of the API.

You can see the initialization and cleanup call here, if there are multiple tasks in the system, it is obvious that the initialization should be called before creating the task.

Notice also that in this case we decided to register an overrun handler, instead of which there could be `NULL`. This handler just prints a message to terminal, what else may be done here depends on a specific application.

During the periodic execution, remaining budget should be watched to avoid overrun.

```
void overrun_handler (
    rtems_cbs_server_id server_id
)
{
    printk( "Budget overrun, fixing the task\n" );
    return;
}

rtems_task Tasks_Periodic(
    rtems_task_argument argument
)
```

(continues on next page)

(continued from previous page)

```
{
    rtems_id             rmid;
    rtems_cbs_server_id  server_id;
    rtems_cbs_parameters params;

    params.deadline = 10;
    params.budget = 4;

    rtems_cbs_initialize();
    rtems_cbs_create_server( &params, &overrun_handler, &server_id );
    rtems_cbs_attach_thread( server_id, RTEMS_SELF );
    rtems_rate_monotonic_create( argument, &rmid );

    while ( 1 ) {
        if (rtems_rate_monotonic_period(rmid, params.deadline) == RTEMS_TIMEOUT)
            break;
        /* Perform some periodic action */
    }

    rtems_rate_monotonic_delete( rmid );
    rtems_cbs_cleanup();
    exit( 1 );
}
```

## 38.4 Directives

This section details the Constant Bandwidth Server's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

### 38.4.1 CBS_INITIALIZE - Initialize the CBS library

**CALLING SEQUENCE:**

```
int rtems_cbs_initialize( void );
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful initialization |
| RTEMS_CBS_ERROR_NO_MEMORY | not enough memory for data |

**DESCRIPTION:**

This routine initializes the library in terms of allocating necessary memory for the servers. In case not enough memory is available in the system, RTEMS_CBS_ERROR_NO_MEMORY is returned, otherwise RTEMS_CBS_OK.

**NOTES:**

Additional memory per each server is allocated upon invocation of rtems_cbs_create_server.

Tasks in the system are not influenced, they still keep executing with their initial parameters.

### 38.4.2 CBS_CLEANUP - Cleanup the CBS library

**CALLING SEQUENCE:**

```
int rtems_cbs_cleanup( void );
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | always successful |

**DESCRIPTION:**

This routine detaches all tasks from their servers, destroys all servers and returns memory back to the system.

**NOTES:**

All tasks continue executing with their initial priorities.

### 38.4.3 CBS_CREATE_SERVER - Create a new bandwidth server

**CALLING SEQUENCE:**

```
int rtems_cbs_create_server (
    rtems_cbs_parameters     *params,
    rtems_cbs_budget_overrun  budget_overrun_callback,
    rtems_cbs_server_id      *server_id
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successfully created |
| RTEMS_CBS_ERROR_NO_MEMORY | not enough memory for data |
| RTEMS_CBS_ERROR_FULL | maximum servers exceeded |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |

**DESCRIPTION:**

This routine prepares an instance of a constant bandwidth server. The input parameter `rtems_cbs_parameters` specifies scheduling parameters of the server (period and budget). If these are not valid, `RTEMS_CBS_ERROR_INVALID_PARAMETER` is returned. The `budget_overrun_callback` is an optional callback function, which is invoked in case the server's budget within one period is exceeded. Output parameter `server_id` becomes an id of the newly created server. If there is not enough memory, the `RTEMS_CBS_ERROR_NO_MEMORY` is returned. If the maximum server count in the system is exceeded, `RTEMS_CBS_ERROR_FULL` is returned.

**NOTES:**

No task execution is being influenced so far.

### 38.4.4 CBS_ATTACH_THREAD - Attach a thread to server

**CALLING SEQUENCE:**

```
int rtems_cbs_attach_thread (
    rtems_cbs_server_id server_id,
    rtems_id            task_id
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successfully attached |
| RTEMS_CBS_ERROR_FULL | server maximum tasks exceeded |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

Attaches a task (`task_id`) to a server (`server_id`). The server has to be previously created. Now, the task starts to be scheduled according to the server parameters and not using initial priority. This implementation allows only one task per server, if the user tries to bind another task to the same server, `RTEMS_CBS_ERROR_FULL` is returned.

**NOTES:**

Tasks attached to servers become preemptible.

### 38.4.5 CBS_DETACH_THREAD - Detach a thread from server

**CALLING SEQUENCE:**

```
int rtems_cbs_detach_thread (
    rtems_cbs_server_id server_id,
    rtems_id            task_id
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successfully detached |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive detaches a thread from server. The task continues its execution with initial priority.

**NOTES:**

The server can be reused for any other task.

### 38.4.6 CBS_DESTROY_SERVER - Destroy a bandwidth server

**CALLING SEQUENCE:**

```
int rtems_cbs_destroy_server (
    rtems_cbs_server_id server_id
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successfully destroyed |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive destroys a server. If any task was attached to the server, the task is detached and continues its execution according to EDF rules with initial properties.

**NOTES:**

This again enables one more task to be created.

### 38.4.7 CBS_GET_SERVER_ID - Get an ID of a server

**CALLING SEQUENCE:**

```
int rtems_cbs_get_server_id (
    rtems_id             task_id,
    rtems_cbs_server_id *server_id
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive returns an id of server belonging to a given task.

### 38.4.8 CBS_GET_PARAMETERS - Get scheduling parameters of a server

**CALLING SEQUENCE:**

```
rtems_cbs_get_parameters (
    rtems_cbs_server_id   server_id,
    rtems_cbs_parameters *params
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive returns a structure with current scheduling parameters of a given server (period and execution time).

**NOTES:**

It makes no difference if any task is assigned or not.

### 38.4.9 CBS_SET_PARAMETERS - Set scheduling parameters

**CALLING SEQUENCE:**

```
int rtems_cbs_set_parameters (
    rtems_cbs_server_id   server_id,
    rtems_cbs_parameters *params
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive sets new scheduling parameters to the server. This operation can be performed regardless of whether a task is assigned or not. If a task is assigned, the parameters become effective imediately, therefore it is recommended to apply the change between two subsequent periods.

**NOTES:**

There is an upper limit on both period and budget equal to (2^31)-1 ticks.

### 38.4.10 CBS_GET_EXECUTION_TIME - Get elapsed execution time

**CALLING SEQUENCE:**

```
int rtems_cbs_get_execution_time (
    rtems_cbs_server_id    server_id,
    time_t                 *exec_time,
    time_t                 *abs_time
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This routine returns consumed execution time (exec_time) of a server during the current period.

**NOTES:**

Absolute time (abs_time) not supported now.

### 38.4.11 CBS_GET_REMAINING_BUDGET - Get remaining execution time

**CALLING SEQUENCE:**

```
int rtems_cbs_get_remaining_budget (
    rtems_cbs_server_id  server_id,
    time_t              *remaining_budget
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive returns remaining execution time of a given server for current period.

**NOTES:**

If the execution time approaches zero, the assigned task should finish computations of the current period.

### 38.4.12 CBS_GET_APPROVED_BUDGET - Get scheduler approved execution time

**CALLING SEQUENCE:**

```
int rtems_cbs_get_approved_budget (
    rtems_cbs_server_id  server_id,
    time_t              *appr_budget
);
```

**DIRECTIVE STATUS CODES:**

| | |
|---|---|
| RTEMS_CBS_OK | successful |
| RTEMS_CBS_ERROR_INVALID_PARAMETER | invalid input argument |
| RTEMS_CBS_ERROR_NOSERVER | server is not valid |

**DESCRIPTION:**

This directive returns server's approved budget for subsequent periods.

CHAPTER

**THIRTYNINE**

# ADA SUPPORT

## 39.1 Introduction

RTEMS has long had support for the Ada programming language by supporting the GNU Ada Compiler (GNAT). There are two primary components to this support:

- Ada Programming Language Support
- Classic API Ada Bindings

## 39.2 Ada Programming Language Support

The Ada programming natively supports multi-threaded programming with its own tasking and concurrency model. Native Ada multi-threaded applications should work using GNAT/RTEMS with no changes.

The application developer will have to account for the specific requirements of the GNAT Run-Time when configuring RTEMS. There are example Ada programs with RTEMS configuration and startup sequences.

## 39.3 Classic API Ada Bindings

An Ada language binding exists for a subset of the RTEMS Classic API. In the early 1990's, there were C and Ada implementations of RTEMS which were functionally equivalent. The source structure was as similar as possible. In fact, the top level `c/` directory at one point had a sibling `ada/`. The current Ada language bindings and test code was derived from that Ada implementation.

The Ada binding specifically excludes some methods which are either not safe or not intended for use from Ada programs. However, methods are generally only added to this binding when a user makes a requests. Thus some methods that could be supported are not. If in doubt, ask about a methods and contribute bindings.

The bindings are located in the `c/src/ada` directory of the RTEMS source tree. The tests are in `c/src/ada-tests`. The bindings following a simple pattern to map the C Classic API calls into Ada subprograms. The following rules are used:

- All RTEMS interfaces are in the RTEMS Ada package. The rtems_ and RTEMS_ prefixes in the C version of the Classic API thus correspond to "RTEMS." in Ada symbol nomenclature. For example, `rtems_task_create()` in C is `RTEMS.Task_Create()` in Ada.
- Classic API directives tend to return an `rtems_status_code`. Some directives also have an output parameter such as an object id on a create operation. Ada subprograms are either pure functions with only a single return value or subprograms. For consistency, the returned status code is always the last parameter of the Ada calling sequence.

Caution should be exercised when writing programs which mix Ada tasks, Classic API tasks, and POSIX API threads. Ada tasks use a priority numbering scheme defined by the Ada programming language. Each Ada task is implemented in GNAT/RTEMS as a single POSIX thread. Thus Ada task priorities must be mapped onto POSIX thread priorities. Complicating matters, Classic API tasks and POSIX API threads use different numbering schemes for priority. Low numbers are high priority in the Classic API while indicating low priority in the POSIX threads API. Experience writing mixed threading model programs teaches that creating a table of the priorities used in the application with the value in all tasking models used is helpful.

The GNAT run-time uses a priority ceiling mutex to protect its data structures. The priority ceiling value is one priority more important than the most important Ada task priority (in POSIX API terms). Do not invoke any services implemented in Ada from a thread or task which is of greater priority. This will result in a priority ceiling violation error and lead to a failure in the Ada run-time.

Exercise extreme caution when considering writing code in Ada which will execute in the context of an interrupt handler. Hardware interrupts are processed outside the context of any thread in RTEMS and this can lead to violating assumptions in the GNAT run-time. Specifically a priority ceiling mutex should never be used from an ISR and it is difficult to predict when the Ada compiler or run-time will use a mutex.

RTEMS has two capabilities which can assist in avoiding this problem. The Classic API Timer Manager allows the creation of Timer Service Routines which execute in the context of a task rather than the clock tick Interrupt Service Routine. Similarly, there is support for Interrupt Tasks which is a mechanism to defer the processing of the event from the hardware interrupt level to a thread.

CHAPTER

**FORTY**

# LINKER SETS

## 40.1 Introduction

Linker sets are a flexible means to create arrays of items out of a set of object files at link-time. For example it is possible to define an item *I* of type *T* in object file *A* and an item *J* of type *T* in object file *B* to be a member of a linker set *S*. The linker will then collect these two items *I* and *J* and place them in consecutive memory locations, so that they can be accessed like a normal array defined in one object file. The size of a linker set is defined by its begin and end markers. A linker set may be empty. It should only contain items of the same type.

The following macros are provided to create, populate and use linker sets.

- *RTEMS_LINKER_SET_BEGIN* (page 1004) - Designator of the linker set begin marker
- *RTEMS_LINKER_SET_END* (page 1005) - Designator of the linker set end marker
- *RTEMS_LINKER_SET_SIZE* (page 1006) - The linker set size in characters
- *RTEMS_LINKER_SET_ITEM_COUNT* (page 1007) - The linker set item count
- *RTEMS_LINKER_SET_IS_EMPTY* (page 1008) - Is the linker set empty?
- *RTEMS_LINKER_SET_FOREACH* (page 1009) - Iterate through the linker set items
- *RTEMS_LINKER_ROSET_DECLARE* (page 1010) - Declares a read-only linker set
- *RTEMS_LINKER_ROSET* (page 1011) - Defines a read-only linker set
- *RTEMS_LINKER_ROSET_ITEM_DECLARE* (page 1012) - Declares a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM_ORDERED_DECLARE* (page 1013) - Declares an ordered read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM_REFERENCE* (page 1014) - References a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM* (page 1015) - Defines a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM_ORDERED* (page 1016) - Defines an ordered read-only linker set item
- *RTEMS_LINKER_ROSET_CONTENT* (page 1017) - Marks a declaration as a read-only linker set content
- *RTEMS_LINKER_RWSET_DECLARE* (page 1018) - Declares a read-write linker set
- *RTEMS_LINKER_RWSET* (page 1019) - Defines a read-write linker set
- *RTEMS_LINKER_RWSET_ITEM_DECLARE* (page 1020) - Declares a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM_ORDERED_DECLARE* (page 1021) - Declares an ordered read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM_REFERENCE* (page 1022) - References a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM* (page 1023) - Defines a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM_ORDERED* (page 1024) - Defines an ordered read-write linker set item

- *RTEMS_LINKER_RWSET_CONTENT* (page 1025) - Marks a declaration as a read-write linker set content

## 40.2 Background

Linker sets are used not only in RTEMS, but also for example in Linux, in FreeBSD, for the GNU C constructor extension and for global C++ constructors. They provide a space efficient and flexible means to initialize modules. A linker set consists of

- dedicated input sections for the linker (e.g. `.ctors` and `.ctors.*` in the case of global constructors),
- a begin marker (e.g. provided by `crtbegin.o`, and
- an end marker (e.g. provided by `crtend.o`).

A module may place a certain data item into the dedicated input section. The linker will collect all such data items in this section and create a begin and end marker. The initialization code can then use the begin and end markers to find all the collected data items (e.g. pointers to initialization functions).

In the linker command file of the GNU linker we need the following output section descriptions.

```
/* To be placed in a read-only memory region */
.rtemsroset : {
  KEEP (*(SORT(.rtemsroset.*)))
}
/* To be placed in a read-write memory region */
.rtemsrwset : {
  KEEP (*(SORT(.rtemsrwset.*)))
}
```

The `KEEP()` ensures that a garbage collection by the linker will not discard the content of this section. This would normally be the case since the linker set items are not referenced directly. The `SORT()` directive sorts the input sections lexicographically. Please note the lexicographical order of the `.begin`, `.content` and `.end` section name parts in the RTEMS linker sets macros which ensures that the position of the begin and end markers are right.

So, what is the benefit of using linker sets to initialize modules? They can be used to initialize and include only those RTEMS managers and other components which are used by the application. For example, in case an application uses message queues, it must call `rtems_message_queue_create()`. In the module implementing this function, we can place a linker set item and register the message queue handler constructor. Otherwise, in case the application does not use message queues, there will be no reference to the `rtems_message_queue_create()` function and the constructor is not registered, thus nothing of the message queue handler will be in the final executable.

For an example see test program `sptests/splinkersets01`.

## 40.3 Directives

### 40.3.1 RTEMS_LINKER_SET_BEGIN - Designator of the linker set begin marker

**CALLING SEQUENCE:**

```
type *begin = RTEMS_LINKER_SET_BEGIN( set );
```

**DESCRIPTION:**

This macro generates the designator of the begin marker of the linker set identified by `set`. The item at the begin marker address is the first member of the linker set if it exists, e.g. the linker set is not empty. A linker set is empty, if and only if the begin and end markers have the same address.

The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set.

**NOTE:**

The compiler may try to be smart. In general it will not work to assign linker set begin and end addresses to pointer variables and treat them like ordinary pointers. The compiler may exploit the fact that actually two distinct objects are involved and use this to optimize. To avoid trouble use *RTEMS_LINKER_SET_SIZE - The linker set size in characters* (page 1006), *RTEMS_LINKER_SET_ITEM_COUNT - The linker set item count* (page 1007), *RTEMS_LINKER_SET_IS_EMPTY - Is the linker set empty?* (page 1008) and *RTEMS_LINKER_SET_FOREACH - Iterate through the linker set items* (page 1009).

### 40.3.2 RTEMS_LINKER_SET_END - Designator of the linker set end marker

**CALLING SEQUENCE:**

```
type *end = RTEMS_LINKER_SET_END( set );
```

**DESCRIPTION:**

This macro generates the designator of the end marker of the linker set identified by `set`. The item at the end marker address is not a member of the linker set. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set.

### 40.3.3 RTEMS_LINKER_SET_SIZE - The linker set size in characters

**CALLING SEQUENCE:**

```
size_t size = RTEMS_LINKER_SET_SIZE( set );
```

**DESCRIPTION:**

This macro returns the size of the linker set identified by `set` in characters. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set.

### 40.3.4 RTEMS_LINKER_SET_ITEM_COUNT - The linker set item count

**CALLING SEQUENCE:**

```
size_t item_count = RTEMS_LINKER_SET_ITEM_COUNT( set );
```

**DESCRIPTION:**

This macro returns the item count of the linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set.

### 40.3.5 RTEMS_LINKER_SET_IS_EMPTY - Is the linker set empty?

**CALLING SEQUENCE:**

```
bool is_empty = RTEMS_LINKER_SET_IS_EMPTY( set );
```

**DESCRIPTION:**

This macro returns true if the linker set identified by `set` is empty, otherwise returns false. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set.

### 40.3.6 RTEMS_LINKER_SET_FOREACH - Iterate through the linker set items

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET( myset, int );

int count( void )
{
  int *item;
  int n;

  n = 0;
  RTEMS_LINKER_SET_FOREACH( myset, item ) {
    n += *item;
  }

  return n;
}
```

**DESCRIPTION:**

This macro generates a for loop statement which iterates through each item of a linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `item` parameter must be a pointer to an item of the linker set. It iterates through all items of the linker set from begin to end.

### 40.3.7 RTEMS_LINKER_ROSET_DECLARE - Declares a read-only linker set

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_DECLARE( set, type );
```

**DESCRIPTION:**

This macro generates declarations for the begin and end markers of a read-only linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set.

### 40.3.8 RTEMS_LINKER_ROSET - Defines a read-only linker set

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET( set, type );
```

**DESCRIPTION:**

This macro generates definitions for the begin and end markers of a read-only linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set.

### 40.3.9 RTEMS_LINKER_ROSET_ITEM_DECLARE - Declares a read-only linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_ITEM_DECLARE( set, type, item );
```

**DESCRIPTION:**

This macro generates a declaration of an item contained in the read-only linker set identified by `set`. For a description of the `set`, `type`, and `item` parameters see *RTEMS_LINKER_ROSET_ITEM - Defines a read-only linker set item* (page 1015).

### 40.3.10 RTEMS_LINKER_ROSET_ITEM_ORDERED_DECLARE - Declares an ordered read-only linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_ITEM_ORDERED_DECLARE( set, type, item, order );
```

**DESCRIPTION:**

This macro generates a declaration of an ordered item contained in the read-only linker set identified by `set`. For a description of the `set`, `type`, `item`, and `order` parameters see *RTEMS_LINKER_ROSET_ITEM_ORDERED - Defines an ordered read-only linker set item* (page 1016).

### 40.3.11 RTEMS_LINKER_ROSET_ITEM_REFERENCE - References a read-only linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_ITEM_REFERENCE( set, type, item );
```

**DESCRIPTION:**

This macro generates a reference to an item contained in the read-only linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set.

### 40.3.12 RTEMS_LINKER_ROSET_ITEM - Defines a read-only linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_ITEM( set, type, item );
```

**DESCRIPTION:**

This macro generates a definition of an item contained in the read-only linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set.

### 40.3.13 RTEMS_LINKER_ROSET_ITEM_ORDERED - Defines an ordered read-only linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_ITEM_ORDERED( set, type, item, order );
```

**DESCRIPTION:**

This macro generates a definition of an ordered item contained in the read-only linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set. The `order` parameter must be a valid linker input section name part on which macro expansion is performed. The items are lexicographically ordered according to the `order` parameter within a linker set. Ordered items are placed before unordered items in the linker set.

**NOTES:**

To be resilient to typos in the order parameter, it is recommended to use the following construct in macros defining items for a particular linker set (see enum in `XYZ_ITEM()`).

```c
#include <rtems/linkersets.h>

typedef struct {
  int foo;
} xyz_item;

/* The XYZ-order defines */
#define XYZ_ORDER_FIRST 0x00001000
#define XYZ_ORDER_AND_SO_ON 0x00002000

/* Defines an ordered XYZ-item */
#define XYZ_ITEM( item, order ) \
  enum { xyz_##item = order }; \
  RTEMS_LINKER_ROSET_ITEM_ORDERED( \
    xyz, const xyz_item *, item, order \
  ) = { &item }

/* Example item */
static const xyz_item some_item = { 123 };
XYZ_ITEM( some_item, XYZ_ORDER_FIRST );
```

### 40.3.14 RTEMS_LINKER_ROSET_CONTENT - Marks a declaration as a read-only linker set content

**CALLING SEQUENCE:**

```
RTEMS_LINKER_ROSET_CONTENT( set, decl );
```

**DESCRIPTION:**

This macro marks a declaration as a read-only linker set content. The linker set is identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `decl` parameter must be an arbitrary variable declaration.

### 40.3.15 RTEMS_LINKER_RWSET_DECLARE - Declares a read-write linker set

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_DECLARE( set, type );
```

**DESCRIPTION:**

This macro generates declarations for the begin and end markers of a read-write linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set.

### 40.3.16 RTEMS_LINKER_RWSET - Defines a read-write linker set

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET( set, type );
```

**DESCRIPTION:**

This macro generates definitions for the begin and end markers of a read-write linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set.

### 40.3.17 RTEMS_LINKER_RWSET_ITEM_DECLARE - Declares a read-write linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_ITEM_DECLARE( set, type, item );
```

**DESCRIPTION:**

This macro generates a declaration of an item contained in the read-write linker set identified by `set`. For a description of the `set`, `type`, and `item` parameters see *RTEMS_LINKER_RWSET_ITEM - Defines a read-write linker set item* (page 1023).

### 40.3.18 RTEMS_LINKER_RWSET_ITEM_ORDERED_DECLARE - Declares an ordered read-write linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_ITEM_ORDERED_DECLARE( set, type, item, order );
```

**DESCRIPTION:**

This macro generates a declaration of an ordered item contained in the read-write linker set identified by `set`. For a description of the `set`, `type`, `item`, and `order` parameters see *RTEMS_LINKER_RWSET_ITEM_ORDERED - Defines an ordered read-write linker set item* (page 1024).

### 40.3.19 RTEMS_LINKER_RWSET_ITEM_REFERENCE - References a read-write linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_ITEM_REFERENCE( set, type, item );
```

**DESCRIPTION:**

This macro generates a reference to an item contained in the read-write linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set.

### 40.3.20 RTEMS_LINKER_RWSET_ITEM - Defines a read-write linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_ITEM( set, type, item );
```

**DESCRIPTION:**

This macro generates a definition of an item contained in the read-write linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set.

### 40.3.21 RTEMS_LINKER_RWSET_ITEM_ORDERED - Defines an ordered read-write linker set item

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_ITEM_ORDERED( set, type, item, order );
```

**DESCRIPTION:**

This macro generates a definition of an ordered item contained in the read-write linker set identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `type` parameter defines the type of the linker set items. The type must be the same for all macro invocations of a particular linker set. The `item` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies an item in the linker set. The `order` parameter must be a valid linker input section name part on which macro expansion is performed. The items are lexicographically ordered according to the `order` parameter within a linker set. Ordered items are placed before unordered items in the linker set.

**NOTES:**

To be resilient to typos in the order parameter, it is recommended to use the following construct in macros defining items for a particular linker set (see enum in `XYZ_ITEM()`).

```c
#include <rtems/linkersets.h>

typedef struct {
  int foo;
} xyz_item;

/* The XYZ-order defines */
#define XYZ_ORDER_FIRST 0x00001000
#define XYZ_ORDER_AND_SO_ON 0x00002000

/* Defines an ordered XYZ-item */
#define XYZ_ITEM( item, order ) \
  enum { xyz_##item = order }; \
  RTEMS_LINKER_RWSET_ITEM_ORDERED( \
    xyz, const xyz_item *, item, order \
  ) = { &item }

/* Example item */
static const xyz_item some_item = { 123 };
XYZ_ITEM( some_item, XYZ_ORDER_FIRST );
```

### 40.3.22 RTEMS_LINKER_RWSET_CONTENT - Marks a declaration as a read-write linker set content

**CALLING SEQUENCE:**

```
RTEMS_LINKER_RWSET_CONTENT( set, decl );
```

**DESCRIPTION:**

This macro marks a declaration as a read-write linker set content. The linker set is identified by `set`. The `set` parameter itself must be a valid C designator on which no macro expansion is performed. It uniquely identifies the linker set. The `decl` parameter must be an arbitrary variable declaration.

CHAPTER

**FORTYONE**

# DIRECTIVE STATUS CODES

## 41.1 Introduction

The directive status code directives are:

- *rtems_status_text* (page 1030) - Return the name for the status code

## 41.2 Directives

The directives are:

| | |
|---|---|
| RTEMS_SUCCESSFUL | successful completion |
| RTEMS_TASK_EXITTED | returned from a task |
| RTEMS_MP_NOT_CONFIGURED | multiprocessing not configured |
| RTEMS_INVALID_NAME | invalid object name |
| RTEMS_INVALID_ID | invalid object id |
| RTEMS_TOO_MANY | too many |
| RTEMS_TIMEOUT | timed out waiting |
| RTEMS_OBJECT_WAS_DELETED | object was deleted while waiting |
| RTEMS_INVALID_SIZE | invalid specified size |
| RTEMS_INVALID_ADDRESS | invalid address specified |
| RTEMS_INVALID_NUMBER | number was invalid |
| RTEMS_NOT_DEFINED | item not initialized |
| RTEMS_RESOURCE_IN_USE | resources outstanding |
| RTEMS_UNSATISFIED | request not satisfied |
| RTEMS_INCORRECT_STATE | task is in wrong state |
| RTEMS_ALREADY_SUSPENDED | task already in state |
| RTEMS_ILLEGAL_ON_SELF | illegal for calling task |
| RTEMS_ILLEGAL_ON_REMOTE_OBJECT | illegal for remote object |
| RTEMS_CALLED_FROM_ISR | invalid environment |
| RTEMS_INVALID_PRIORITY | invalid task priority |
| RTEMS_INVALID_CLOCK | invalid time buffer |
| RTEMS_INVALID_NODE | invalid node id |
| RTEMS_NOT_CONFIGURED | directive not configured |
| RTEMS_NOT_OWNER_OF_RESOURCE | not owner of resource |
| RTEMS_NOT_IMPLEMENTED | directive not implemented or feature not available in configuration |
| RTEMS_INTERNAL_ERROR | RTEMS inconsistency detected |
| RTEMS_NO_MEMORY | could not get enough memory |
| RTEMS_IO_ERROR | driver I/O error |
| RTEMS_INTERRUPTED | returned by driver to indicate interrupted operation |

### 41.2.1 STATUS_TEXT - Returns the enumeration name for a status code

**CALLING SEQUENCE:**

```
const char *rtems_status_text(
    rtems_status_code code
);
```

**DIRECTIVE STATUS CODES**

The status code enumeration name or "?" in case the status code is invalid.

**DESCRIPTION:**

Returns the enumeration name for the specified status code.

CHAPTER

**FORTYTWO**

# EXAMPLE APPLICATION

```c
/*
*  This file contains an example of a simple RTEMS
*  application.  It instantiates the RTEMS Configuration
*  Information using confdef.h and contains two tasks:
*  a user initialization task and a simple task.
*/

#include <rtems.h>

rtems_task user_application(rtems_task_argument argument);

rtems_task init_task(
    rtems_task_argument ignored
)
{
    rtems_id          tid;
    rtems_status_code status;
    rtems_name        name;

    name = rtems_build_name( 'A', 'P', 'P', '1' )

    status = rtems_task_create(
        name, 1, RTEMS_MINIMUM_STACK_SIZE,
        RTEMS_NO_PREEMPT, RTEMS_FLOATING_POINT, &tid
    );
    if ( status != RTEMS_SUCCESSFUL ) {
        printf( "rtems_task_create failed with status of %d.\n", status );
        exit( 1 );
    }

    status = rtems_task_start( tid, user_application, 0 );
    if ( status != RTEMS_SUCCESSFUL ) {
        printf( "rtems_task_start failed with status of %d.\n", status );
        exit( 1 );
    }

    status = rtems_task_delete( SELF );    /* should not return */
```

(continues on next page)

(continued from previous page)

```

    printf( "rtems_task_delete returned with status of %d.\n", status );
    exit( 1 );
}

rtems_task user_application(rtems_task_argument argument)
{
    /* application specific initialization goes here */
    while ( 1 )  {              /* infinite loop */
        /*  APPLICATION CODE GOES HERE
         *
         *  This code will typically include at least one
         *  directive which causes the calling task to
         *  give up the processor.
         */
    }
}

/* The Console Driver supplies Standard I/O. */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
/* The Clock Driver supplies the clock tick. */
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_TASKS 2
#define CONFIGURE_INIT_TASK_NAME rtems_build_name( 'E', 'X', 'A', 'M' )
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

CHAPTER

**FORTYTHREE**

# GLOSSARY

**ABI**
This term is an acronym for Application Binary Interface.

**active**
A term used to describe an object which has been created by an application.

**AMP**
This term is an acronym for Asymmetric Multiprocessing.

**APA**
This term is an acronym for Arbitrary Processor Affinity. APA schedulers allow a thread to have an arbitrary affinity to a processor set, rather than a restricted mapping to only one processor of the set or the ability to run on all processors of the set.

It has two variants, *Weak APA* and *Strong APA*.

**aperiodic task**
A task which must execute only at irregular intervals and has only a soft deadline.

**API**
This term is an acronym for Application Programming Interface.

**application**
In this document, software which makes use of RTEMS.

**ASR**
This term is an acronym for *Asynchronous Signal Routine*.

**assembler language**
The assembler language is a programming language which can be translated very easily into machine code and data. For this project assembler languages are restricted to languages accepted by the *GNU* assembler program for the target architectures.

**asynchronous**
Not related in order or timing to other occurrences in the system.

**Asynchronous Signal Routine**
Similar to a hardware interrupt except that it is associated with a task and is run in the context of a task. The directives provided by the signal manager are used to service signals.

**atomic operations**
Atomic operations are defined in terms of *C11*.

**awakened**
A term used to describe a task that has been unblocked and may be scheduled to the CPU.

**BCB**
This term is an acronym for Barrier Control Block.

**big endian**
A data representation scheme in which the bytes composing a numeric value are arranged such that the most significant byte is at the lowest address.

**bit-mapped**
A data encoding scheme in which each bit in a variable is used to represent something different. This makes for compact data representation.

**block**
A physically contiguous area of memory.

**blocked task**
The task state entered by a task which has been previously started and cannot continue execution until the reason for waiting has been satisfied. Blocked tasks are not an element of the set of ready tasks of a scheduler instance.

**Board Support Package**
A collection of device initialization and control routines specific to a particular type of board or collection of boards.

**broadcast**
To simultaneously send a message to a logical set of destinations.

**BSP**
This term is an acronym for *Board Support Package*.

**buffer**
A fixed length block of memory allocated from a partition.

**C language**
The C language for this project is defined in terms of *C11*.

**C++11**
The standard ISO/IEC 14882:2011.

**C++14**
The standard ISO/IEC 14882:2014.

**C++17**
The standard ISO/IEC 14882:2017.

**C++20**
The standard ISO/IEC 14882:2020.

**C11**
The standard ISO/IEC 9899:2011.

**C17**
The standard ISO/IEC 9899:2018.

**calling convention**
The processor and compiler dependent rules which define the mechanism used to invoke subroutines in a high-level language. These rules define the passing of arguments, the call and return mechanism, and the register set which must be preserved.

**CCB**
This term is an acronym for Change Control Board.

**Central Processing Unit**
This term is equivalent to the terms processor and microprocessor.

**chain**
A data structure which allows for efficient dynamic addition and removal of elements. It differs from an array in that it is not limited to a predefined size.

**Clock Driver**
The Clock Driver is a driver which provides the *clock tick* and a time counter. The time counter is used to drive the *CLOCK_REALTIME* and *CLOCK_MONOTONIC*. The Clock Driver can be initialized by the application with the *CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER* (page 596) and *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration options.

**clock tick**
The clock tick is a coarse time measure provided by RTEMS. The *Clock Driver* emits clock ticks at rate specified by the *CONFIGURE_MICROSECONDS_PER_TICK* (page 583) application configuration option. In contrast to *CLOCK_REALTIME* and *CLOCK_MONOTONIC*, the clock tick rate is not affected by incremental adjustments.

**CLOCK_MONOTONIC**
The CLOCK_MONOTONIC is a clock provided by RTEMS which measures the time since an unspecified starting point. In contrast to *CLOCK_REALTIME*, this clock cannot be set. It may be affected by incremental adjustments for example carried out by the *NTP* or the use of a *PPS* signal. See also *CLOCK_REALTIME*, *clock tick*, and *Clock Driver*.

**CLOCK_REALTIME**
The CLOCK_REALTIME is a clock provided by RTEMS which measures the real time (also known as wall-clock time). It is defined by *POSIX*. In particular, every day is treated as if it contains exactly 86400 seconds and leap seconds are ignored. This clock can be set by the application which may result in time jumps. It may be affected by incremental adjustments for example carried out by the *NTP* or the use of a *PPS* signal. RTEMS can represent time points of this clock in nanoseconds ranging from 1988-01-01T00:00:00.000000000Z to 2514-05-31T01:53:03.999999999Z. See also *CLOCK_MONOTONIC*, *clock tick*, and *Clock Driver*.

**cluster**
We have clustered scheduling in case the set of processors of a system is partitioned into non-empty pairwise disjoint subsets. These subsets are called clusters. Clusters with a cardinality of one are partitions. Each cluster is owned by exactly one scheduler instance.

**coalesce**
The process of merging adjacent holes into a single larger hole. Sometimes this process is referred to as garbage collection.

**Configuration Table**
A table which contains information used to tailor RTEMS for a particular application.

**context**
All of the processor registers and operating system data structures associated with a task.

**context switch**
Alternate term for task switch. Taking control of the processor from one task and transferring it to another task.

**control block**
A data structure used by the executive to define and control an object.

**core**
When used in this manual, this term refers to the internal executive utility functions. In the interest of application portability, the core of the executive should not be used directly by applications.

**CPU**
This term is an acronym for *Central Processing Unit*.

**critical section**
A section of code which must be executed indivisibly.

**CRT**
This term is an acronym for Cathode Ray Tube. Normally used in reference to the man-machine interface.

**current priority**
The current priority of a *task* is the *task priority* with respect to the *home scheduler* of the task. It is an aggregation of the *real priority* and temporary priority adjustments due to locking protocols, the rate-monotonic period objects on some schedulers such as EDF, and the *POSIX* sporadic server. The current priority is an *eligible priority*.

**deadline**
A fixed time limit by which a task must have completed a set of actions. Beyond this point, the results are of reduced value and may even be considered useless or harmful.

**device**
A peripheral used by the application that requires special operation software. See also device driver.

**device driver**
Control software for special peripheral devices used by the application.

**Device Driver Table**
A table which contains the entry points for each of the configured device drivers.

**directives**
RTEMS' provided routines that provide support mechanisms for real-time applications.

**dispatch**
The act of loading a task's context onto the CPU and transferring control of the CPU to that task.

**Doorstop**
Doorstop is a requirements management tool.

**dormant**
The state entered by a task after it is created and before it has been started.

**DPCB**
This term is an acronym for Dual-Ported Memory Control Block.

**dual-ported**
A term used to describe memory which can be accessed at two different addresses.

**dynamic extension sets**
The dynamic extension sets are a list of *user extensions*. The list is defined by the system services used by the application and directive calls such as *rtems_extension_create()* (page 550). See also *initial extension sets*.

**EARS**
This term is an acronym for Easy Approach to Requirements Syntax.

**EDF**
This term is an acronym for Earliest Deadline First.

**ELF**
This term is an acronym for Executable and Linkable Format.

**eligible priority**
An eligible priority of a *task* is the *task priority* with respect to the corresponding *eligible scheduler* of the task. An eligible priority is either the *current priority* or a *helping priority* of a task.

**eligible scheduler**
An eligible scheduler of a *task* is a *scheduler* which can be used by the task to allocate a processor for the task.

**embedded**
An application that is delivered as a hidden part of a larger system. For example, the software in a fuel-injection control system is an embedded application found in many late-model automobiles.

**entry point**
The address at which a function or task begins to execute. In C, the entry point of a function is the function's name.

**envelope**
A buffer provided by the MPCI layer to RTEMS which is used to pass messages between nodes in a multiprocessor system. It typically contains routing information needed by the MPCI. The contents of an envelope are referred to as a packet.

**error code**
This term has the same meaning as *status code*.

**ESCB**
This term is an acronym for Extension Set Control Block.

**events**
A method for task communication and synchronization. The directives provided by the event manager are used to service events.

**exception**
A synonym for interrupt.

**executing task**
The task state entered by a task after it has been given control of the processor. In SMP configurations, a task may be registered as executing on more than one processor for short time frames during task migration. Blocked tasks can be executing until they issue a thread dispatch.

**executive**
In this document, this term is used to referred to RTEMS. Commonly, an executive is a small real-time operating system used in embedded systems.

**exported**
An object known by all nodes in a multiprocessor system. An object created with the GLOBAL attribute will be exported.

**extension forward order**
The *user extensions* may be invoked in extension forward order. In forward order, all user extensions of the *initial extension sets* are invoked before all user extensions of the *dynamic extension sets*. In the initial extension sets the order is defined by the table index. The user extension with the lowest table index is invoked first. In the dynamic extension sets the order is defined by the registration order. The first registered user extension is invoked first. See also *extension reverse order*.

**extension reverse order**
The *user extensions* may be invoked in extension reverse order. In reverse order, all user extensions of the *dynamic extension sets* are invoked before all user extensions of the *initial extension sets*. In the dynamic extension sets the order is defined by the registration order. The last registered user extension is invoked first. In the initial extension sets the order is defined by the table index. The user extension with the highest table index is invoked first. See also *extension forward order*.

**external address**
The address used to access dual-ported memory by all the nodes in a system which do not own the memory.

**FIFO**
This term is an acronym for *First In First Out*.

**First In First Out**
A discipline for manipulating entries in a data structure. See also *Last In First Out*.

**floating point coprocessor**
A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.

**formal model**
A model of a computing component (hardware or software) that has a mathematically based *semantics*.

**freed**
A resource that has been released by the application to RTEMS.

**Futex**
This term is an abbreviation for Fast User-Space Locking. The futex support in RTEMS is provided for the barriers of the *OpenMP* library provided by *GCC*. It could be used to implement high performance *SMP* synchronization primitives which offer random-fairness.

**GCC**
This term is an acronym for GNU Compiler Collection.

**global**
An object that has been created with the GLOBAL attribute and exported to all nodes in a multiprocessor system.

**global construction**
In the global construction, the C++ global constructors and constructor functions are invoked. See *Global Construction* (page 92).

**GNAT**
*GNAT* is the *GNU* compiler for Ada, integrated into the *GCC*.

**GNU**
This term is an acronym for GNU's Not Unix.

**GPL**
This term is an acronym for GNU General Public License.

**GPLv2**
This term is an acronym for GNU General Public License Version 2.

**GPLv3**
This term is an acronym for GNU General Public License Version 3.

**GR712RC**
The GR712RC is a *system-on-chip* containing two processors of the *SPARC target architecture*.

**GR740**
The GR740 is a *system-on-chip* containing four processors of the *SPARC target architecture*.

**handler**
The equivalent of a manager, except that it is internal to RTEMS and forms part of the core. A handler is a collection of routines which provide a related set of functions. For example, there is a handler used by RTEMS to manage all objects.

**hard real-time system**
A real-time system in which a missed deadline causes the worked performed to have no value or to result in a catastrophic effect on the integrity of the system.

**heap**
A data structure used to dynamically allocate and deallocate variable sized blocks of memory.

**heir task**
A task is an heir if it is registered as an heir in a processor of the system. A task can be the heir on at most one processor in the system. In case the executing and heir tasks differ on a processor and a thread dispatch is marked as necessary, then the next thread dispatch will make the heir task the executing task.

**helping priority**
A helping priority of a *task* is the *task priority* with respect to the corresponding *helping scheduler* of the task. A helping priority is an *eligible priority*.

**helping scheduler**
A helping scheduler of a *task* is a *scheduler* which is a *eligible scheduler* and which is not the *home scheduler* of the task.

**heterogeneous**
A multiprocessor computer system composed of dissimilar processors.

**higher priority**
A *task* `H` has a higher *priority* than a task `L`, if task `H` is more important than task `L`.

**home scheduler**
The home scheduler of a *task* is a *scheduler* which is an *eligible scheduler* and which is assigned to the task during its initialization or explicitly via a directive call such as *rtems_task_set_scheduler()* (page 137).

**homogeneous**
A multiprocessor computer system composed of a single type of processor.

**I/O**
This term is an acronym for Input/Output.

**ID**
An RTEMS assigned identification tag used to access an active object.

**IDLE task**
A special low priority task which assumes control of the CPU when no other task is able to execute.

**ineligible scheduler**
An ineligible scheduler of a *task* is a *scheduler* which is not an *eligible scheduler*.

**initial extension sets**
The initial extension sets are a table of *user extensions*. The table is defined by the application configuration for example through the *CONFIGURE_INITIAL_EXTENSIONS* (page 573) application configuration option. The initial extension sets cannot be altered during runtime through directive calls. See also *dynamic extension sets*.

**interface**
A specification of the methodology used to connect multiple independent subsystems.

**internal address**
The address used to access dual-ported memory by the node which owns the memory.

**interrupt**
A hardware facility that causes the CPU to suspend execution, save its status, and transfer control to a specific location.

**interrupt level**
A mask used to by the CPU to determine which pending interrupts should be serviced. If a pending interrupt is below the current interrupt level, then the CPU does not recognize that interrupt.

**interrupt service**
An *interrupt service* consists of an *Interrupt Service Routine* which is called with a user provided argument upon reception of an interrupt service request. The routine is invoked in interrupt context. Interrupt service requests may have a priority and an affinity to a set of processors. An *interrupt service* is a *software component*.

**Interrupt Service Routine**
An ISR is invoked by the CPU to process a pending interrupt.

**ISR**
This term is an acronym for *Interrupt Service Routine*.

**ISVV**
This term is an acronym for Independent Software Verification and Validation.

**kernel**
In this document, this term is used as a synonym for executive.

**Last In First Out**
A discipline for manipulating entries in a data structure. See also *First In First Out*.

**LIFO**
This term is an acronym for *Last In First Out*.

**Linear Temporal Logic**
This is a logic that states properties about (possibly infinite) sequences of states.

**list**
A data structure which allows for dynamic addition and removal of entries. It is not statically limited to a particular size.

**little endian**
A data representation scheme in which the bytes composing a numeric value are arranged such that the least significant byte is at the lowest address.

**LLVM**
This term is an acronym for Low Level Virtual Machine. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

**local**
An object which was created with the LOCAL attribute and is accessible only on the node it was created and resides upon. In a single processor configuration, all objects are local.

**local operation**
The manipulation of an object which resides on the same node as the calling task.

**logical address**
An address used by an application. In a system without memory management, logical addresses will equal physical addresses.

**loosely-coupled**
A multiprocessor configuration where shared memory is not used for communication.

**lower priority**
A *task* `L` has a lower *priority* than a task `H`, if task `L` is less important than task `H`.

**LTL**
This term is an acronym for *Linear Temporal Logic*.

**major number**
The index of a device driver in the Device Driver Table.

**manager**
A group of related RTEMS' directives which provide access and control over resources.

**MCS**
This term is an acronym for Mellor-Crummey Scott.

**memory pool**
Used interchangeably with heap.

**message**
A sixteen byte entity used to communicate between tasks. Messages are sent to message queues and stored in message buffers.

**message buffer**
A block of memory used to store messages.

**message queue**
An RTEMS object used to synchronize and communicate between tasks by transporting messages between sending and receiving tasks.

**Message Queue Control Block**
A data structure associated with each message queue used by RTEMS to manage that message queue.

**minor number**
A numeric value passed to a device driver, the exact usage of which is driver dependent.

**mode**
An entry in a task's control block that is used to determine if the task allows preemption, timeslicing, processing of signals, and the interrupt disable level used by the task.

**MPCI**
This term is an acronym for *Multiprocessor Communications Interface Layer*.

**MrsP**
This term is an acronym for Multiprocessor Resource-Sharing Protocol.

**multiprocessing**
The simultaneous execution of two or more processes by a multiple processor computer system.

**multiprocessor**
A computer with multiple CPUs available for executing applications.

**Multiprocessor Communications Interface Layer**
A set of user-provided routines which enable the nodes in a multiprocessor system to communicate with one another.

**Multiprocessor Configuration Table**
The data structure defining the characteristics of the multiprocessor target system with which RTEMS will communicate.

**multitasking**
The alternation of execution amongst a group of processes on a single CPU. A scheduling algorithm is used to determine which process executes at which time.

**mutual exclusion**
A term used to describe the act of preventing other tasks from accessing a resource simultaneously.

**nested**
A term used to describe an ASR that occurs during another ASR or an ISR that occurs during another ISR.

**node**
A term used to reference a processor running RTEMS in a multiprocessor system.

**non-existent**
The state occupied by an uncreated or deleted task.

**NTP**
This term is an acronym for Network Time Protocol.

**NUMA**
This term is an acronym for Non-Uniform Memory Access.

**numeric coprocessor**
A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.

**OBC**
This term is an acronym for On-Board Computer.

**object**
In this document, this term is used to refer collectively to tasks, timers, message queues, partitions, regions, semaphores, ports, and rate monotonic periods. All RTEMS objects have IDs and user-assigned names.

**object-oriented**
A term used to describe systems with common mechanisms for utilizing a variety of entities. Object-oriented systems shield the application from implementation details.

**OMIP**
This term is an acronym for O(m) Independence-Preserving Protocol. OMIP is a generalization of the *priority inheritance* locking protocol to clustered scheduling. The `m` denotes the number of processors in the system.

**OpenMP**
This term is an acronym for Open Multi-Processing.

**operating system**
The software which controls all the computer's resources and provides the base upon which application programs can be written.

**overhead**
The portion of the CPUs processing power consumed by the operating system.

**packet**
A buffer which contains the messages passed between nodes in a multiprocessor system. A packet is the contents of an envelope.

**partition**
This term has two definitions:

1. A partition is an RTEMS object which is used to allocate and deallocate fixed size blocks of memory from an dynamically specified area of memory.
2. A *cluster* with a cardinality of one is a partition.

**Partition Control Block**
A data structure associated with each partition used by RTEMS to manage that partition.

**PCB**
This term is an acronym for Period Control Block.

**pending**
A term used to describe a task blocked waiting for an event, message, semaphore, or signal.

**periodic task**
A task which must execute at regular intervals and comply with a hard deadline.

**physical address**
The actual hardware address of a resource.

**poll**
A mechanism used to determine if an event has occurred by periodically checking for a particular status. Typical events include arrival of data, completion of an action, and errors.

**pool**
A collection from which resources are allocated.

**portability**
A term used to describe the ease with which software can be rehosted on another computer.

**POSIX**
This term is an acronym for Portable Operating System Interface.

**posting**
The act of sending an event, message, semaphore, or signal to a task.

**PPS**
This term is an acronym for Pulse-Per-Second.

**preempt**
The act of forcing a task to relinquish the processor and dispatching to another task.

**priority**
The priority is a mechanism used to represent the relative importance of an element in a set of items.

For example, *RTEMS* uses *task priorities* to determine which *task* should execute on a processor. In RTEMS, priorities are represented by non-negative integers.

For the Classic *API*, if a numerical priority value `A` is greater than a numerical priority value `B`, then `A` expresses a *lower priority* than `B`. If a numerical priority value `C` is less than a numerical priority value `D`, then `C` expresses a *higher priority* than `D`.

For the *POSIX* API, if a numerical priority value `R` is less than a numerical priority value `S`, then `R` expresses a lower priority than `S`. If a numerical priority value `T` is greater than a numerical priority value `U`, then `T` expresses a higher priority than `U`.

**priority boosting**
A simple approach to extend the priority inheritance protocol for clustered scheduling is priority boosting. In case a mutex is owned by a task of another cluster, then the priority of the owner task is raised to an artificially high priority. This approach is not used in RTEMS, see also *OMIP*.

**priority inheritance**
An algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource. This avoids the problem of priority inversion.

**priority inversion**
A form of indefinite postponement which occurs when a high priority tasks requests access to shared resource currently allocated to low priority task. The high priority task must block until the low priority task releases the resource.

**processor utilization**
The percentage of processor time used by a task or a set of tasks.

**proxy**
An RTEMS control structure used to represent, on a remote node, a task which must block as part of a remote operation.

**Proxy Control Block**
A data structure associated with each proxy used by RTEMS to manage that proxy.

**PTCB**
This term is an acronym for *Partition Control Block*.

**PXCB**
This term is an acronym for *Proxy Control Block*.

**QCB**
This term is an acronym for *Message Queue Control Block*.

**quantum**
The application defined unit of time in which the processor is allocated.

**queue**
Alternate term for message queue.

**ready task**
A task occupies this state when it is available to be given control of a processor. A ready task has no processor assigned. The scheduler decided that other tasks are currently more important. A task that is ready to execute and has a processor assigned is called scheduled.

**real priority**
Each *task* has exactly one real priority. The real priority is always with respect to the *home scheduler* of a task. It is defined during task initialization. It may be changed by directives such as *rtems_task_set_priority()* (page 128) and *rtems_task_set_scheduler()* (page 137). The real priority is the foundation of the *current priority*.

**real-time**
A term used to describe systems which are characterized by requiring deterministic response times to external stimuli. The external stimuli require that the response occur at a precise time or the response is incorrect.

**reentrant**
A term used to describe routines which do not modify themselves or global variables.

**refinement**
A *refinement* is a relationship between a specification and its implementation as code.

**region**
An RTEMS object which is used to allocate and deallocate variable size blocks of memory from a dynamically specified area of memory.

**Region Control Block**
A data structure associated with each region used by RTEMS to manage that region.

**registers**
Registers are locations physically located within a component, typically used for device control or general purpose storage.

**reification**
Another term used to denote *refinement*.

**remote**
Any object that does not reside on the local node.

**remote operation**
The manipulation of an object which does not reside on the same node as the calling task.

**ReqIF**
This term is an acronym for Requirements Interchange Format.

**resource**
A hardware or software entity to which access must be controlled.

**resume**
Removing a task from the suspend state. If the task's state is ready following a call to the

`rtems_task_resume` directive, then the task is available for scheduling.

**return code**
This term has the same meaning as *status code*.

**return value**
The value returned by a function. A return value may be a *status code*.

**RNCB**
This term is an acronym for *Region Control Block*.

**round-robin**
A task scheduling discipline in which tasks of equal priority are executed in the order in which they are made ready.

**RS-232**
A standard for serial communications.

**RTEMS**
This term is an acronym for Real-Time Executive for Multiprocessor Systems.

**RTEMS epoch**
The RTEMS epoch is a point in time. It is 1988-01-01T00:00:00Z in ISO 8601 time format.

**running**
The state of a rate monotonic timer while it is being used to delineate a period. The timer exits this state by either expiring or being canceled.

**scenario**
In the context of formal verification, in a setting that involves many concurrent tasks that interleave in arbitrary ways, a scenario describes a single specific possible interleaving. One interpretation of the behaviour of a concurrent system is the set of all its scenarios.

**schedulable**
A set of tasks which can be guaranteed to meet their deadlines based upon a specific scheduling algorithm.

**schedule**
The process of choosing which task should next enter the executing state.

**scheduled task**
A task is scheduled if it is allowed to execute and has a processor assigned. Such a task executes currently on a processor or is about to start execution. A task about to start execution it is an heir task on exactly one processor in the system.

**scheduler**
A scheduler or scheduling algorithm allocates processors to a subset of its set of ready tasks. So it manages access to the processor resource. Various algorithms exist to choose the tasks allowed to use a processor out of the set of ready tasks. One method is to assign each task a priority number and assign the tasks with the lowest priority number to one processor of the set of processors owned by a scheduler instance.

A scheduler is either an *eligible scheduler* or a *ineligible scheduler* for a task. An *eligible scheduler* is either the *home scheduler* or a *helping scheduler* for a task.

**scheduler instance**
A scheduler instance is a scheduling algorithm with a corresponding context to store its internal state. Each processor in the system is owned by at most one scheduler instance. The

processor to scheduler instance assignment is determined at application configuration time. See *Configuring a System* (page 555).

**segments**
Variable sized memory blocks allocated from a region.

**semantics**
This term refers to the meaning of text or utterances in some language. In a software engineering context these will be programming, modelling or specification languages.

**semaphore**
An RTEMS object which is used to synchronize tasks and provide mutually exclusive access to resources.

**Semaphore Control Block**
A data structure associated with each semaphore used by RTEMS to manage that semaphore.

**shared memory**
Memory which is accessible by multiple nodes in a multiprocessor system.

**signal**
An RTEMS provided mechanism to communicate asynchronously with a task. Upon reception of a signal, the ASR of the receiving task will be invoked.

**signal set**
A thirty-two bit entity which is used to represent a task's collection of pending signals and the signals sent to a task.

**SIS**
This term is an acronym for Simple Instruction Simulator. The SIS is a *SPARC* V7/V8 and RISC-V RV32IMACFD *target architecture* simulator.

**SMCB**
This term is an acronym for *Semaphore Control Block*.

**SMP**
This term is an acronym for Symmetric Multiprocessing.

**SMP barriers**
The SMP barriers ensure that a defined set of independent threads of execution on a set of processors reaches a common synchronization point in time. They are implemented using atomic operations. Currently a sense barrier is used in RTEMS.

**SMP locks**
The SMP locks ensure mutual exclusion on the lowest level and are a replacement for the sections of disabled interrupts. Interrupts are usually disabled while holding an SMP lock. They are implemented using atomic operations. Currently a ticket lock is used in RTEMS.

**soft real-time system**
A real-time system in which a missed deadline does not compromise the integrity of the system.

**software component**
This term is defined by ECSS-E-ST-40C 3.2.28 as a "part of a software system". For this project a *software component* shall be any of the following items and nothing else:

- *software unit*
- explicitly defined *ELF* symbol in a *source code* file

- *assembler language* data in a source code file
- *C language* object with static storage duration
- C language object with thread-local storage duration
- *thread*
- *interrupt service*
- collection of *software components* (this is a software architecture element)

Please note that explicitly defined ELF symbols and assembler language data are considered a software component only if they are defined in a *source code* file. For example, this rules out symbols and data generated as side-effects by the toolchain (compiler, assembler, linker) such as jump tables, linker trampolines, exception frame information, etc.

**software item**
This term has the same meaning as *software product*.

**software product**
The *software product* is the *RTEMS* real-time operating system.

**software unit**
This term is defined by ECSS-E-ST-40C 3.2.24 as a "separately compilable piece of source code". For this project a *software unit* shall be any of the following items and nothing else:

- *assembler language* function in a *source code* file
- *C language* function (external and internal linkage)

A *software unit* is a *software component*.

**source code**
This project uses the *source code* definition of the Linux Information Project: "Source code (also referred to as source or code) is the version of software as it is originally written (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters)."

**SPARC**
This term is an acronym for Scalable Processor ARChitecture. See also *target architecture*.

**sporadic task**
A task which executes at irregular intervals and must comply with a hard deadline. A minimum period of time between successive iterations of the task can be guaranteed.

**stack**
A data structure that is managed using a Last In First Out (LIFO) discipline. Each task has a stack associated with it which is used to store return information and local variables.

**status code**
A status code indicates the completion status of an operation. For example most RTEMS directives return a status code through the *return value* to indicate a successful operation or error conditions.

**Strong APA**
Strong APA is a specialization of *APA*. Schedulers which implement strong APA recursively searches for a processor in the *thread*'s affinity set, whenever a thread becomes ready for execution, followed by the processors in the affinity set of threads that are assigned the processor present in the ready thread's affinity set. This is done to find a thread to processor mapping

that does not violate the priority ordering and to provide a thread to processor mapping with a higher total priority of the threads allocated a processor. Similar analysis is done when a thread blocks. See also [CGB14].

**suspend**
A term used to describe a task that is not competing for the CPU because it has had a `rtems_task_suspend` directive.

**synchronous**
Related in order or timing to other occurrences in the system.

**system call**
In this document, this is used as an alternate term for directive.

**system-on-chip**
This project uses the system on a chip definition of Wikipedia: "A system on a chip or system-on-chip is an integrated circuit that integrates most or all components of a computer or other electronic system."

Systems on a chip are *target* systems for applications using *RTEMS*.

**target**
The system on which the application will ultimately execute.

**target architecture**
The target architecture is the instruction set architecture (ISA) of the *target*. Some RTEMS features depend on the target architecture. For the details consult the *RTEMS CPU Architecture Supplement*.

**TAS**
This term is an acronym for Test-And-Set.

**task**
This project uses the thread definition of Wikipedia: "a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system."

It consists normally of a set of registers and a stack. The scheduler assigns processors to a subset of the ready tasks. The terms task and *thread* are synonym in RTEMS. The term task is used throughout the Classic API, however, internally in the operating system implementation and the POSIX API the term thread is used.

A *task* is a *software component*.

**Task Control Block**
A data structure associated with each task used by RTEMS to manage that task.

**task entry**
The task entry is invoked to execute the task's job. Before the task entry is invoked, the thread begin *user extensions* run in the context of the task. After the return of the task entry, the thread exitted user extensions run in the context of the task. The first user initialization task performs the *global construction* after running the thread begin extensions and before the task entry is invoked. See also *rtems_task_start()* (page 118).

**task migration**
Task migration happens in case a task stops execution on one processor and resumes execution on another processor.

**task priority**
A task *priority* of a *task* determines its importance relative to other tasks.

The scheduler use task priorities to determine which *ready task* gets a processor allocated, see *scheduled task*. The *eligible priorities* of a task define the position of the task in a *wait queue* which uses the priority discipline. Each task has at least the *real priority*.

Task priorities are used in *wait queues* which use the priority discipline to determine the dequeueing order of tasks.

**task processor affinity**
The set of processors on which a task is allowed to execute.

**task switch**
Alternate terminology for context switch. Taking control of the processor from one task and given to another.

**TCB**
This term is an acronym for *Task Control Block*.

**thread**
This term has the same meaning as *task*.

**thread dispatch**
The thread dispatch transfers control of the processor from the currently executing thread to the heir thread of the processor.

**tick**
The basic unit of time used by RTEMS. It is a user-configurable number of microseconds. The current tick expires when a clock tick directive is invoked.

**tightly-coupled**
A multiprocessor configuration system which communicates via shared memory.

**timeout**
An argument provided to a number of directives which determines the maximum length of time an application task is willing to wait to acquire the resource if it is not immediately available.

**timer**
An RTEMS object used to invoke subprograms at a later time.

**Timer Control Block**
A data structure associated with each timer used by RTEMS to manage that timer.

**timeslice**
The application defined unit of time in which the processor is allocated.

**timeslicing**
A task scheduling discipline in which tasks of equal priority are executed for a specific period of time before being preempted by another task.

**TLS**
This term is an acronym for Thread-Local Storage [Dre13]. TLS is available in *C11* and *C++11*. The support for TLS depends on the CPU port [RTE].

**TMCB**
This term is an acronym for *Timer Control Block*.

**transient overload**
A temporary rise in system activity which may cause deadlines to be missed. Rate Monotonic Scheduling can be used to determine if all deadlines will be met under transient overload.

**TTAS**
This term is an acronym for Test and Test-And-Set.

**Unix epoch**
The Unix epoch is a point in time. It is 1970-01-01T00:00:00Z in ISO 8601 time format.

**User Extension Table**
A table which contains the entry points for each user extensions.

**user extensions**
User extensions are software routines provided by the application to enhance the functionality of RTEMS. An active user extension is either in the *initial extension sets* or the *dynamic extension sets*. User extensions are invoked in *extension forward order* or *extension reverse order*.

**User Initialization Tasks Table**
A table which contains the information needed to create and start each of the user initialization tasks.

**user-provided**
These terms are used to designate any software routines which must be written by the application designer.

**user-supplied**
This term has the same meaning as *user-provided*.

**vector**
Memory pointers used by the processor to fetch the address of routines which will handle various exceptions and interrupts.

**wait queue**
The list of tasks blocked pending the release of a particular resource. Message queues, regions, and semaphores have a wait queue associated with them.

**Weak APA**
Weak APA is a specialization of *APA*. This refers to Linux's push and pull implementation of APA model. When a *thread* becomes ready for execution, it is allocated a processor if there is an idle processor, or a processor executing a lower priority thread in its affinity set. Unlike *Strong APA*, no thread is migrated from its processor to find a thread to processor mapping. See also [CGB14].

**YAML**
This term is an acronym for YAML Ain't Markup Language.

**yield**
When a task voluntarily releases control of the processor.

# BIBLIOGRAPHY


[RTE] RTEMS CPU Architecture Supplement. URL: https://docs.rtems.org/docs/main/cpu-supplement.pdf.

[BBB+13] Dave Banham, Andrew Banks, Mark Bradbury, Paul Burden, Mark Dawson-Butterworth, Mike Hennell, Chris Hills, Steve Montgomery, Chris Tapp, and Liz Whiting. *MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems*. MISRA Limited, March 2013. ISBN 978-1906400101.

[Boe12] Hans-J. Boehm. Can Seqlocks Get Along With Programming Language Memory Models? Technical Report, HP Laboratories, June 2012. HPL-2012-68. URL: http://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf.

[Bra11] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. URL: http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf.

[Bra13] Björn B. Brandenburg. A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, 292–302. 2013. URL: http://www.mpi-sws.org/~bbb/papers/pdf/ecrts13b.pdf.

[Bur91] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6:116–128, 1991.

[BW01] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, November 2001. ISBN 978-0321417459.

[BW13] A. Burns and A. J. Wellings. A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*. 2013. URL: http://www-users.cs.york.ac.uk/~burns/MRSPpaper.pdf.

[CBHM15] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the Implementation of MrsP. In *Reliable Software Technologies - Ada-Europe 2015*, 179–195. 2015.

[CGB14] Felipe Cerqueira, Arpan Gujarati, and Björn B. Brandenburg. Linux's Processor Affinity API, Refined: Shifting Real-Time Tasks towards Higher Schedulability. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014)*. 2014. URL: http://www.mpi-sws.org/~bbb/papers/pdf/rtss14f.pdf.

[CvdBruggenC16] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Overrun Handling for Mixed-Criticality Support in RTEMS. In *Mixed Criticality Systems - WMC 2016*, 13–14. 2016. URL: http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2016-wmc.pdf.

[CMV14] Davide Compagnin, Enrico Mezzetti, and Tullio Vardanega. Putting RUN into practice: implementation and evaluation. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*. 2014.

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[Dre13] Ulrich Drepper. *ELF Handling For Thread-Local Storage*. 2013. URL: http://www.akkadia.org/drepper/tls.pdf.

[FRK02] Hubertus Franke, Rusty Russel, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium 2002*, 479–495. 2002. URL: https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[GN06] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium*, 333–346. 2006. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf.

[GCB13] Arpan Gujarati, Felipe Cerqueira, and Björn B. Brandenburg. Schedulability Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*. 2013. URL: https://people.mpi-sws.org/~bbb/papers/pdf/ecrts13a-rev1.pdf.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real-Time Systems Symposium*, 166–171. 1989.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[LLF+16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: a Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. 2016. URL: https://hal.archives-ouvertes.fr/hal-01295194/document.

[Mot88] Motorola. *Real Time Executive Interface Definition*. Motorola Inc., Microcomputer Division and Software Components Group, Inc., January 1988. DRAFT 2.1. URL: https://ftp.rtems.org/pub/rtems/publications/RTEID-ORKID/RTEID-2.1/RTEID-2_1.pdf.

[SG90] Lui Sha and J. B. Goodenough. Real-time scheduling theory and Ada. *Computer*, 23:53–62, 1990.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[VC95] G. Varghese and A. Costello. Redesigning the BSD callout and timer facilities. Technical Report, Washington University in St. Louis, November 1995. WUCS-95-23. URL: http://web.mit.edu/afs.new/sipb/user/daveg/ATHENA/Info/wucs-95-23.ps.

[VL87] G. Varghese and T. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. 1987. URL: http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf.

[VIT90] VITA. *Open Real-Time Kernel Interface Definition*. VITA, the VMEbus International Trade Association, August 1990. Draft 2.1. URL: https://ftp.rtems.org/pub/rtems/publications/RTEID-ORKID/ORKID-2.1/ORKID-2_1.pdf.

[Wil12] Anthony Williams. *C++ Concurrency in Action - Practical Multithreading*. Manning Publications Co, 2012. ISBN 978-1933988771.

# INDEX

## Symbols


/dev/null, 600
/dev/zero, 607
_Terminate, 517


## A


ABI, **1031**
active, **1031**
Ada, 993
add memory, 442
add memory to a region, 442
AMP, **1031**
announce, 526
announce fatal error, 526
APA, **1031**
aperiodic task, 302, **1031**
API, **1031**
application, **1031**
Application architecture, 10
ASR, 405, **1031**
ASR mode, 405
ASR vs. ISR, 405
assembler language, **1031**
asynchronous, **1031**
Asynchronous Signal Routine, **1031**
asynchronous signal routine, 405
atomic operations, **1031**
attach a thread to server, 985
awakened, **1031**


## B


barrier, 353
BCB, **1032**
big endian, **1032**
binary semaphores, 326
bit-mapped, **1032**
block, **1032**
blocked task, **1032**
Board Support Package, **1032**
Board Support Packages, 532
broadcast, **1032**
broadcast message to a queue, 385
BSP, 532, **1032**
BSP_output_char_function_type, 37
BSP_polling_getchar_function_type, 37
BSPs, 532
buffer, **1032**
buffers, 417
building, 103, 104, 395, 405, 417, 433


## C


C language, **1032**
C Program Heap, 587
C++11, **1032**
C++14, **1032**
C++17, **1032**
C++20, **1032**
C11, **1032**
C17, **1032**
cache, 494
calling convention, **1032**
cancel a period, 315
cancel a timer, 284
cbs, 973
CBS limitations, 977
CBS parameters, 976
CCB, **1032**
Central Processing Unit, **1033**
chain, **1033**
chain append a node, 944
chain append a node unprotected, 945
chain extract a node, 938
chain extract a node unprotected, 939
chain get first node, 940, 941
chain get head, 928
chain get tail, 929
chain initialize, 925
chain initialize empty, 926
chain insert a node, 942
chain insert a node unprotected, 943
chain is chain empty, 931
chain is node null, 927

chain is node the first, 932
chain is node the head, 936
chain is node the last, 933
chain is node the tail, 937
chain iterate, 922
chain only one node, 934, 935
chains, 918
chare are nodes equal, 930
cleanup the CBS library, 983
clear C Program Heap, 591
clear RTEMS Workspace, 591
clock, 233
Clock Driver, **1033**
clock tick, **1033**
clock tick quantum, 583
CLOCK_MONOTONIC, **1033**
CLOCK_REALTIME, **1033**
close a device, 480
cluster, **1033**
coalesce, **1033**
communication and synchronization, 27
conclude current period, 317
Configuration Table, **1033**
configure message queue buffer memory, 581
CONFIGURE_APPLICATION_DISABLE_FILESYSTEM, 656
CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER, 593
CONFIGURE_APPLICATION_EXTRA_DRIVERS, 594
CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER, 595
CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER, 596
CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER, 597
CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER, 598
CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER, 599
CONFIGURE_APPLICATION_NEEDS_LIBBLOCK, 685
CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER, 600
CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER, 601
CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER, 602
CONFIGURE_APPLICATION_NEEDS_SIMPLE_TASK_CONSOLE_DRIVER, 603
CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER, 604
CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER, 605
CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER, 606
CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER, 607
CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS, 608
CONFIGURE_ATA_DRIVER_TASK_PRIORITY, 609
CONFIGURE_BDBUF_BUFFER_COUNT, 781
CONFIGURE_BDBUF_BUFFER_MAX_SIZE, 686
CONFIGURE_BDBUF_BUFFER_MIN_SIZE, 687
CONFIGURE_BDBUF_BUFFER_SIZE, 781
CONFIGURE_BDBUF_CACHE_MEMORY_SIZE, 688
CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS, 689
CONFIGURE_BDBUF_MAX_WRITE_BLOCKS, 690
CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY, 691
CONFIGURE_BDBUF_TASK_STACK_SIZE, 692
CONFIGURE_CBS_MAXIMUM_SERVERS, 711
CONFIGURE_DIRTY_MEMORY, 567
CONFIGURE_DISABLE_BSP_SETTINGS, 568
CONFIGURE_DISABLE_CLASSIC_API_NOTEPADS, 781
CONFIGURE_DISABLE_NEWLIB_REENTRANCY, 569
CONFIGURE_ENABLE_GO, 781
CONFIGURE_EXCEPTION_TO_SIGNAL_MAPPING, 610
CONFIGURE_EXECUTIVE_RAM_SIZE, 570
CONFIGURE_EXTRA_MPCI_RECEIVE_SERVER_STACK, 733
CONFIGURE_EXTRA_TASK_STACKS, 571
CONFIGURE_FILESYSTEM_ALL, 657
CONFIGURE_FILESYSTEM_DOSFS, 658
CONFIGURE_FILESYSTEM_FTPFS, 659
CONFIGURE_FILESYSTEM_IMFS, 660
CONFIGURE_FILESYSTEM_JFFS2, 661
CONFIGURE_FILESYSTEM_NFS, 662
CONFIGURE_FILESYSTEM_RFS, 663
CONFIGURE_FILESYSTEM_TFTPFS, 664
CONFIGURE_GNAT_RTEMS, 781
CONFIGURE_HAS_OWN_BDBUF_TABLE, 781
CONFIGURE_HAS_OWN_CONFIGURATION_TABLE, 781
CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE, 781
CONFIGURE_HAS_OWN_INIT_TASK_TABLE, 781
CONFIGURE_HAS_OWN_MOUNT_TABLE, 781
CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE, 781
CONFIGURE_IDLE_TASK_BODY, 706

CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION, 707
CONFIGURE_IDLE_TASK_STACK_SIZE, 708
CONFIGURE_IDLE_TASK_STORAGE_SIZE, 709
CONFIGURE_IMFS_DISABLE_CHMOD, 665
CONFIGURE_IMFS_DISABLE_CHOWN, 666
CONFIGURE_IMFS_DISABLE_LINK, 667
CONFIGURE_IMFS_DISABLE_MKNOD, 668
CONFIGURE_IMFS_DISABLE_MKNOD_DEVICE, 669
CONFIGURE_IMFS_DISABLE_MKNOD_FILE, 670
CONFIGURE_IMFS_DISABLE_MOUNT, 671
CONFIGURE_IMFS_DISABLE_READDIR, 672
CONFIGURE_IMFS_DISABLE_READLINK, 673
CONFIGURE_IMFS_DISABLE_RENAME, 674
CONFIGURE_IMFS_DISABLE_RMNOD, 675
CONFIGURE_IMFS_DISABLE_SYMLINK, 676
CONFIGURE_IMFS_DISABLE_UNMOUNT, 677
CONFIGURE_IMFS_DISABLE_UTIME, 678
CONFIGURE_IMFS_ENABLE_MKFIFO, 679
CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK, 680
CONFIGURE_INIT, 572
CONFIGURE_INIT_TASK_ARGUMENTS, 626
CONFIGURE_INIT_TASK_ATTRIBUTES, 627
CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE, 628
CONFIGURE_INIT_TASK_ENTRY_POINT, 629
CONFIGURE_INIT_TASK_INITIAL_MODES, 630
CONFIGURE_INIT_TASK_NAME, 631
CONFIGURE_INIT_TASK_PRIORITY, 632
CONFIGURE_INIT_TASK_STACK_SIZE, 633
CONFIGURE_INITIAL_EXTENSIONS, 573
CONFIGURE_INTERRUPT_STACK_SIZE, 574
CONFIGURE_JFFS2_DELAYED_WRITE_TASK_PRIORITY, 681
CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS, 781
CONFIGURE_MALLOC_DIRTY, 575
CONFIGURE_MAXIMUM_ADA_TASKS, 782
CONFIGURE_MAXIMUM_BARRIERS, 614
CONFIGURE_MAXIMUM_DEVICES, 782
CONFIGURE_MAXIMUM_DRIVERS, 611
CONFIGURE_MAXIMUM_FAKE_ADA_TASKS, 782
CONFIGURE_MAXIMUM_FILE_DESCRIPTORS, 576
CONFIGURE_MAXIMUM_GO_CHANNELS, 782
CONFIGURE_MAXIMUM_GOROUTINES, 782
CONFIGURE_MAXIMUM_MESSAGE_QUEUES, 615
CONFIGURE_MAXIMUM_MRSP_SEMAPHORES, 782
CONFIGURE_MAXIMUM_PARTITIONS, 616
CONFIGURE_MAXIMUM_PERIODS, 617
CONFIGURE_MAXIMUM_PORTS, 618
CONFIGURE_MAXIMUM_POSIX_BARRIERS, 782
CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES, 782
CONFIGURE_MAXIMUM_POSIX_KEY_VALUE_PAIRS, 637
CONFIGURE_MAXIMUM_POSIX_KEYS, 636
CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTORS, 782
CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES, 638
CONFIGURE_MAXIMUM_POSIX_MUTEXES, 782
CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS, 639
CONFIGURE_MAXIMUM_POSIX_RWLOCKS, 782
CONFIGURE_MAXIMUM_POSIX_SEMAPHORES, 640
CONFIGURE_MAXIMUM_POSIX_SHMS, 641
CONFIGURE_MAXIMUM_POSIX_SPINLOCKS, 783
CONFIGURE_MAXIMUM_POSIX_THREADS, 642
CONFIGURE_MAXIMUM_POSIX_TIMERS, 643
CONFIGURE_MAXIMUM_PRIORITY, 712
CONFIGURE_MAXIMUM_PROCESSORS, 577
CONFIGURE_MAXIMUM_REGIONS, 619
CONFIGURE_MAXIMUM_SEMAPHORES, 620
CONFIGURE_MAXIMUM_TASKS, 621
CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE, 578
CONFIGURE_MAXIMUM_THREAD_NAME_SIZE, 579
CONFIGURE_MAXIMUM_TIMERS, 622
CONFIGURE_MAXIMUM_USER_EXTENSIONS, 623
CONFIGURE_MEMORY_OVERHEAD, 580
CONFIGURE_MESSAGE_BUFFER_MEMORY, 581
CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE, 581
CONFIGURE_MICROSECONDS_PER_TICK, 583
CONFIGURE_MINIMUM_POSIX_THREAD_STACK_SIZE, 644
CONFIGURE_MINIMUM_TASK_STACK_SIZE, 584
CONFIGURE_MINIMUM_TASKS_WITH_USER_PROVIDED_STORAGE, 624
CONFIGURE_MP_APPLICATION, 734
CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS, 735
CONFIGURE_MP_MAXIMUM_NODES, 736
CONFIGURE_MP_MAXIMUM_PROXIES, 737
CONFIGURE_MP_MPCI_TABLE_POINTER, 738
CONFIGURE_MP_NODE_NUMBER, 739
CONFIGURE_NUMBER_OF_TERMIOS_PORTS, 782
CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE, 783
CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT, 646
CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE, 647

CONFIGURE_POSIX_INIT_THREAD_TABLE, 648
CONFIGURE_POSIX_TIMERS_FACE_BEHAVIOR, 731
CONFIGURE_RECORD_EXTENSIONS_ENABLED, 650
CONFIGURE_RECORD_FATAL_DUMP_BASE64, 651
CONFIGURE_RECORD_FATAL_DUMP_BASE64_ZLIB, 652
CONFIGURE_RECORD_INTERRUPTS_ENABLED, 653
CONFIGURE_RECORD_PER_PROCESSOR_ITEMS, 654
CONFIGURE_RTEMS_INIT_TASKS_TABLE, 634
CONFIGURE_SCHEDULER_ASSIGNMENTS, 713
CONFIGURE_SCHEDULER_CBS, 714
CONFIGURE_SCHEDULER_EDF, 715
CONFIGURE_SCHEDULER_EDF_SMP, 716
CONFIGURE_SCHEDULER_NAME, 717
CONFIGURE_SCHEDULER_PRIORITY, 718
CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP, 719
CONFIGURE_SCHEDULER_PRIORITY_SMP, 720
CONFIGURE_SCHEDULER_SIMPLE, 721
CONFIGURE_SCHEDULER_SIMPLE_SMP, 722
CONFIGURE_SCHEDULER_STRONG_APA, 723
CONFIGURE_SCHEDULER_TABLE_ENTRIES, 724
CONFIGURE_SCHEDULER_USER, 725
CONFIGURE_SMP_APPLICATION, 783
CONFIGURE_SMP_MAXIMUM_PROCESSORS, 783
CONFIGURE_STACK_CHECKER_ENABLED, 585
CONFIGURE_SWAPOUT_BLOCK_HOLD, 693
CONFIGURE_SWAPOUT_SWAP_PERIOD, 694
CONFIGURE_SWAPOUT_TASK_PRIORITY, 695
CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY, 697
CONFIGURE_SWAPOUT_WORKER_TASKS, 696
CONFIGURE_TASK_STACK_ALLOCATOR, 699
CONFIGURE_TASK_STACK_ALLOCATOR_AVOIDS_WORK_SPACE, 700
CONFIGURE_TASK_STACK_ALLOCATOR_FOR_IDLE, 701
CONFIGURE_TASK_STACK_ALLOCATOR_INIT, 702
CONFIGURE_TASK_STACK_DEALLOCATOR, 703
CONFIGURE_TASK_STACK_FROM_ALLOCATOR, 704
CONFIGURE_TERMIOS_DISABLED, 783
CONFIGURE_TICKS_PER_TIMESLICE, 586
CONFIGURE_UNIFIED_WORK_AREAS, 587
CONFIGURE_UNLIMITED_ALLOCATION_SIZE, 588
CONFIGURE_UNLIMITED_OBJECTS, 589
CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM, 682
CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM, 683
CONFIGURE_VERBOSE_SYSTEM_INITIALIZATION, 590
CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY, 591
configuring a system, 553
constant bandwidth server scheduling, 69
context, **1033**
context switch, **1033**
control block, **1033**
convert external to internal address, 464
convert internal to external address, 465
core, **1034**
counting semaphores, 326
CPU, **1034**
CPU Usage, 885
create a barrier, 359
create a message queue, 372
create a new bandwidth server, 984
create a partition, 420
create a period, 312
create a port, 460
create a region, 438
create a regulator, 832
create a semaphore, 335
create a task, 109
create a timer, 281
create an extension set, 550
critical section, **1034**
CRT, **1034**
current priority, **1034**
current task mode, 132
current task priority, 128, 130

## D

data types, 34
deadline, **1034**
definition, 99, 302, 395, 417, 433, 456, 532, 845, 846, 848
delay a task for a count of clock ticks, 134
delay a task until a wall time, 135
delays, 239
delete a barrier, 362
delete a message queue, 379
delete a partition, 425
delete a period, 316
delete a port, 463
delete a region, 441
delete a regulator, 834
delete a semaphore, 341
delete a task, 122
delete a timer, 285
delete an extension set, 552

deleting a task, 124
destroy a bandwidth server, 987
detach a thread from server, 986
device, **1034**
device driver, **1034**
device driver interface, 470
Device Driver Table, 469, **1034**
device drivers, 465
device names, 469
directives, **1034**
disable interrupts, 164, 168
disabling interrupts, 158
dispatch, **1034**
dispatching, 65
distributed multiprocessing, 26
Doorstop, **1034**
dormant, **1034**
DPCB, **1034**
dual ported memory, 453, 456
dual-ported, **1034**
dynamic extension sets, **1034**

## E

earliest deadline first scheduling, 68
EARS, **1035**
EDF, **1035**
ELF, **1035**
eligible priority, **1035**
eligible scheduler, **1035**
embedded, **1035**
enable interrupts, 166, 170
entry point, **1035**
envelope, **1035**
error code, **1035**
ESCB, **1035**
establish an ASR, 410
establish an ISR, 162
event condition, 395
event flag, 395
event set, 395
events, 391, **1035**
exception, **1035**
exception frame, 529
executing task, **1035**
executive, **1035**
exported, **1035**
extension forward order, **1036**
extension reverse order, **1036**
external address, **1036**
external addresses, 456

## F

fatal error, 526, 530, 531
fatal error detection, 517
fatal error processing, 517
fatal error user extension, 517
fatal errors, 514
FIFO, **1036**
fire a task-based timer at time of day, 294
fire a timer after an interval, 286
fire a timer at time of day, 288
fire task-based a timer after an interval, 292
First In First Out, **1036**
flash interrupts, 167
floating point, 102
floating point coprocessor, **1036**
flush a semaphore, 348
flush messages on a queue, 390
formal model, **1036**
freed, **1036**
Futex, **1036**

## G

GCC, **1036**
get an ID of a server, 988
get buffer from partition, 427
get class from object ID, 27
get elapsed execution time, 991
get ID of a partition, 423
get index from object ID, 27
get node from object ID, 27
get number of pending messages, 389
get per-task variable, 151
get remaining execution time, 992
get scheduler approved execution time, 993
get scheduling parameters of a server, 989
get segment from region, 444
get size of segment, 452
get statistics of period, 321
get status of period, 319
get task mode, 132
get task notepad entry, 148
get task preemption mode, 132
get task priority, 128, 130
getchark(), 494
global, **1036**
global construction, 92, **1036**
global objects, 845
global objects table, 845
global scope, 26

GNAT, **1036**
GNU, **1036**
GPL, **1037**
GPLv2, **1037**
GPLv3, **1037**
GR712RC, **1037**
GR740, **1037**

## H

handler, **1037**
hard real-time system, **1037**
heap, **1037**
heir task, **1037**
helping priority, **1037**
helping scheduler, **1037**
heterogeneous, **1037**
heterogeneous multiprocessing, 850
higher priority, **1037**
home scheduler, **1037**
homogeneous, **1037**

## I

I/O, **1037**
ID, **1038**
IDLE task, **1038**
IDLE task storage size, 709
immediate ceiling priority protocol, 29
ineligible scheduler, **1038**
initial extension sets, **1038**
initialization tasks, 87
initialize a device driver, 477
initialize RTEMS, 95
initialize the CBS library, 982
initiate the Timer Server, 290
install an ASR, 410
install an ISR, 162
interface, **1038**
internal address, **1038**
internal addresses, 456
Internal Architecture, 11
interrupt, **1038**
interrupt level, 101, **1038**
interrupt levels, 157
interrupt processing, 157
interrupt service, **1038**
Interrupt Service Routine, **1038**
interrupt stack size, 574
interrupts, 152
IO control, 483
IO Manager, 465
is interrupt in progress, 171
ISR, **1038**
ISR vs. ASR, 405
ISVV, **1038**

## K

kernel, **1038**
Kernel Character I/O Support, 483

## L

Last In First Out, **1038**
libpci, 869
LIFO, **1038**
Linear Temporal Logic, **1038**
linkersets, 998
list, **1039**
little endian, **1039**
LLVM, **1039**
local, **1039**
local operation, **1039**
local scope, 26
lock a semaphore, 343
locking protocols, 28
logical address, **1039**
loosely-coupled, **1039**
lower priority, **1039**
LTL, **1039**

## M

major device number, 469
major number, **1039**
manager, **1039**
manual round robin, 65
maximum file descriptors, 576
maximum priority, 712
maximum thread name size, 579
MCS, **1039**
memory for a single message queue's buffers, 581
memory for task tasks, 571
memory management, 33
memory pool, **1039**
message, **1039**
message buffer, **1039**
message queue, **1039**
message queue attributes, 367
Message Queue Control Block, **1039**
message queues, 364
messages, 364
minimum POSIX thread stack size, 644
minimum task stack size, 584
minor device number, 469
minor number, **1040**
mode, **1040**

MPCI, 848, **1040**
MPCI and remote operations, 846
MPCI entry points, 848
MrsP, **1040**
multiprocessing, 26, 841, **1040**
multiprocessing topologies, 844
multiprocessor, **1040**
Multiprocessor Communications Interface Layer, **1040**
Multiprocessor Configuration Table, **1040**
Multiprocessor Resource Sharing Protocol (*MrsP*), 29
multitasking, **1040**
mutual exclusion, 326, **1040**

## N

nested, **1040**
node, **1040**
nodes, 845
non-existent, **1040**
NTP, **1040**
NUMA, **1040**
number of priority levels, 712
numeric coprocessor, **1040**

## O

O(m) Independence-Preserving Protocol (*OMIP*), 30
OBC, **1040**
object, **1041**
object id, 25
object id composition, 25
object manipulation, 893
object name, 25
object-oriented, **1041**
objects, 24
obtain a semaphore, 343
obtain buffer from partition, 427
obtain buffer from regulator, 836
obtain ID of a partition, 423
obtain ID of caller, 117
obtain per-task variable, 151
obtain region information, 449
obtain region information on free blocks, 450
obtain statistics from regulator, 840
obtain statistics of period, 321
obtain status of period, 319
obtain task mode, 132
obtain task priority, 128, 130
obtain the ID of a timer, 283
obtain the time of day, 274
obtaining class from object ID, 27
obtaining index from object ID, 27
obtaining node from object ID, 27
OMIP, **1041**
open a device, 479
OpenMP, **1041**
operating system, **1041**
overhead, **1041**

## P

packet, **1041**
panic, 527
partition, 417, **1041**
partition attribute set, 417
Partition Control Block, **1041**
partitions, 413
PCB, **1041**
PCI, 869
PCI address translation, 878
PCI Interrupt, 878
PCI_LIB_AUTO, 740
PCI_LIB_PERIPHERAL, 740
PCI_LIB_READ, 740
PCI_LIB_STATIC, 740
pending, **1041**
per-task variable, 150, 152
period initiation, 317
period statistics report, 325, 326
periodic task, 302, **1041**
periodic tasks, 297
physical address, **1041**
poll, **1041**
pool, **1041**
portability, **1041**
ports, 453
POSIX, **1042**
posting, **1042**
PPS, **1042**
preempt, **1042**
preemption, 64, 101
prepend node, 946
prepend node unprotected, 947
print period statistics report, 325, 326
printk(), 491
priority, 100, **1042**
priority boosting, **1042**
priority ceiling protocol, 29
priority inheritance, **1042**
priority inheritance protocol, 29
priority inversion, 29, **1042**
priority scheduling, 63

processor utilization, **1042**
proxy, 846, **1042**
Proxy Control Block, **1042**
PTCB, **1042**
put message at front of queue, 383
putk(), 490
PXCB, **1042**

## Q

QCB, **1043**
quantum, **1043**
queue, **1043**

## R

rate mononitonic tasks, 297
Rate Monotonic Scheduling Algorithm, 302
rbtree doc, 953
rbtrees, 947
read from a device, 481
ready task, **1043**
real priority, **1043**
real-time, **1043**
Real-time applications, 8
Real-time executive, 9
receive message from a queue, 387
Red-Black Trees, 947
reentrant, **1043**
refinement, **1043**
region, 433, 442, **1043**
region attribute set, 433
Region Control Block, **1043**
regions, 429
register a device driver, 474
register a device in the file system, 478
registers, **1043**
regulator, 824
reification, **1043**
release a barrier, 364
release a semaphore, 346
release buffer back to regulator, 837
remote, **1043**
remote operation, **1043**
ReqIF, **1043**
reset a timer, 296
reset statistics of all periods, 324
reset statistics of period, 323
resize segment, 447
resource, **1043**
restarting a task, 120
restore interrupt level, 166, 170
resume, **1043**
resuming a task, 126
return buffer to partition, 429
return code, **1044**
return segment to region, 446
return value, **1044**
RMS Algorithm, 302
RMS First Deadline Rule, 304
RMS Processor Utilization Rule, 304
RMS schedulability analysis, 303
RNCB, **1044**
round robin scheduling, 64
round-robin, **1044**
RS-232, **1044**
RTEMS, **1044**
RTEMS Data Types, 34
RTEMS epoch, **1044**
rtems extensions table index, 544
RTEMS Workspace, 587
rtems_api_configuration_table, 37
rtems_asr, 38, 407, 977
rtems_asr_entry, 38
rtems_assert_context, 38
rtems_attribute, 39
rtems_barrier_create(), 359
rtems_barrier_delete(), 362
rtems_barrier_ident(), 26, 361
rtems_barrier_release(), 364
rtems_barrier_wait(), 363
rtems_build_id(), 901
rtems_build_name(), 25, 902
rtems_cache_aligned_malloc(), 514
rtems_cache_disable_data(), 511
rtems_cache_disable_instruction(), 513
rtems_cache_enable_data(), 510
rtems_cache_enable_instruction(), 512
rtems_cache_flush_entire_data(), 507
rtems_cache_flush_multiple_data_lines(), 498
rtems_cache_get_data_cache_size(), 505
rtems_cache_get_data_line_size(), 503
rtems_cache_get_instruction_cache_size(), 506
rtems_cache_get_instruction_line_size(), 504
rtems_cache_get_maximal_line_size(), 502
rtems_cache_instruction_sync_after_code_change(), 501
rtems_cache_invalidate_entire_data(), 508
rtems_cache_invalidate_entire_instruction(), 509
rtems_cache_invalidate_multiple_data_lines(), 499

rtems_cache_invalidate_multiple_instruction_lines(), 500
rtems_cbs_attach_thread(), 985
rtems_cbs_cleanup(), 983
rtems_cbs_create_server(), 984
rtems_cbs_destroy_server(), 987
rtems_cbs_detach_thread(), 986
rtems_cbs_get_approved_budget(), 993
rtems_cbs_get_execution_time(), 991
rtems_cbs_get_parameters(), 989
rtems_cbs_get_remaining_budget(), 992
rtems_cbs_get_server_id(), 988
rtems_cbs_initialize(), 982
rtems_cbs_parameters, 976
rtems_cbs_set_parameters(), 990
rtems_chain_append(), 944
rtems_chain_append_unprotected(), 945
rtems_chain_are_nodes_equal(), 930
rtems_chain_extract(), 938
rtems_chain_extract_unprotected(), 939
rtems_chain_get(), 940
rtems_chain_get_unprotected(), 941
rtems_chain_has_only_one_node(), 934
rtems_chain_head(), 928
rtems_chain_initialize(), 925
rtems_chain_initialize_empty(), 926
rtems_chain_insert(), 942
rtems_chain_insert_unprotected(), 943
rtems_chain_is_empty(), 931
rtems_chain_is_first(), 932
rtems_chain_is_head(), 936
rtems_chain_is_last(), 933
rtems_chain_is_null_node(), 927
rtems_chain_is_tail(), 937
rtems_chain_node_count_unprotected(), 935
rtems_chain_prepend(), 946
rtems_chain_prepend_unprotected(), 947
rtems_chain_tail(), 929
rtems_clock_get(), 274
rtems_clock_get_boot_time(), 260
rtems_clock_get_boot_time_bintime(), 261
rtems_clock_get_boot_time_timeval(), 262
rtems_clock_get_monotonic(), 253
rtems_clock_get_monotonic_bintime(), 254
rtems_clock_get_monotonic_coarse(), 257
rtems_clock_get_monotonic_coarse_bintime(), 258
rtems_clock_get_monotonic_coarse_timeval(), 259
rtems_clock_get_monotonic_sbintime(), 255
rtems_clock_get_monotonic_timeval(), 256
rtems_clock_get_options, 274
rtems_clock_get_realtime(), 247
rtems_clock_get_realtime_bintime(), 248
rtems_clock_get_realtime_coarse(), 250
rtems_clock_get_realtime_coarse_bintime(), 251
rtems_clock_get_realtime_coarse_timeval(), 252
rtems_clock_get_realtime_timeval(), 249
rtems_clock_get_seconds_since_epoch(), 263
rtems_clock_get_ticks_per_second(), 264
rtems_clock_get_ticks_since_boot(), 265
rtems_clock_get_tod(), 245
rtems_clock_get_tod_timeval(), 246
rtems_clock_get_uptime(), 266
rtems_clock_get_uptime_nanoseconds(), 269
rtems_clock_get_uptime_seconds(), 268
rtems_clock_get_uptime_timeval(), 267
rtems_clock_set(), 243
rtems_clock_tick_before(), 272
rtems_clock_tick_later(), 270
rtems_clock_tick_later_usec(), 271
rtems_configuration_get_do_zero_of_workspace(), 747
rtems_configuration_get_idle_task(), 749
rtems_configuration_get_idle_task_stack_size(), 748
rtems_configuration_get_interrupt_stack_size(), 750
rtems_configuration_get_maximum_barriers(), 751
rtems_configuration_get_maximum_extensions(), 752
rtems_configuration_get_maximum_message_queues(), 753
rtems_configuration_get_maximum_partitions(), 754
rtems_configuration_get_maximum_periods(), 755
rtems_configuration_get_maximum_ports(), 756
rtems_configuration_get_maximum_processors(), 757
rtems_configuration_get_maximum_regions(), 758
rtems_configuration_get_maximum_semaphores(), 759
rtems_configuration_get_maximum_tasks(), 760
rtems_configuration_get_maximum_

timers(), 761
rtems_configuration_get_microseconds_per_tick(), 762
rtems_configuration_get_milliseconds_per_tick(), 763
rtems_configuration_get_nanoseconds_per_tick(), 764
rtems_configuration_get_number_of_initial_extensions(), 765
rtems_configuration_get_rtems_api_configuration(), 777
rtems_configuration_get_stack_allocate_for_idle_hook(), 766
rtems_configuration_get_stack_allocate_hook(), 767
rtems_configuration_get_stack_allocate_init_hook(), 768
rtems_configuration_get_stack_allocator_avoids_work_space(), 769
rtems_configuration_get_stack_free_hook(), 770
rtems_configuration_get_stack_space_size(), 771
rtems_configuration_get_ticks_per_timeslice(), 772
rtems_configuration_get_unified_work_area(), 773
rtems_configuration_get_user_extension_table(), 774
rtems_configuration_get_user_multiprocessing_table(), 775
rtems_configuration_get_work_space_size(), 776
rtems_cpu_usage_report(), 892
rtems_cpu_usage_reset(), 893
rtems_device_driver, 39
rtems_device_driver_entry, 39
rtems_device_major_number, 39, 469
rtems_device_minor_number, 39, 469
rtems_driver_address_table, 39
rtems_event_receive(), 401
rtems_event_send(), 399
rtems_event_set, 40, 395
rtems_exception_frame, 40
rtems_exception_frame_print(), 529
rtems_extension_create(), 550
rtems_extension_delete(), 552
rtems_extension_ident(), 26, 553
rtems_extensions_table, 40, 543
rtems_fatal(), 526
rtems_fatal_code, 40
rtems_fatal_error_occurred(), 532
rtems_fatal_extension, 41, 548
rtems_fatal_source, 41
rtems_fatal_source_text(), 530
rtems_get_build_label(), 743
rtems_get_copyright_notice(), 744
rtems_get_target_hash(), 745
rtems_get_version_string(), 746
RTEMS_GLOBAL, 26
rtems_id, 25, 41
rtems_initialization_tasks_table, 41
rtems_initialize_executive(), 95
rtems_internal_error_text(), 531
rtems_interrupt_attributes, 42
rtems_interrupt_catch(), 162
rtems_interrupt_clear(), 200
rtems_interrupt_disable(), 164
rtems_interrupt_enable(), 166
rtems_interrupt_entry, 44
rtems_interrupt_entry_initialize(), 186
RTEMS_INTERRUPT_ENTRY_INITIALIZER(), 185
rtems_interrupt_entry_install(), 187
rtems_interrupt_entry_remove(), 189
rtems_interrupt_flash(), 167
rtems_interrupt_get_affinity(), 204
rtems_interrupt_get_attributes(), 207
rtems_interrupt_get_priority(), 201
rtems_interrupt_handler, 44
rtems_interrupt_handler_install(), 190
rtems_interrupt_handler_iterate(), 208
rtems_interrupt_handler_remove(), 192
rtems_interrupt_is_in_progress(), 171
rtems_interrupt_is_pending(), 196
rtems_interrupt_level, 44
rtems_interrupt_local_disable(), 168
rtems_interrupt_local_enable(), 170
rtems_interrupt_lock, 44
rtems_interrupt_lock_acquire(), 174
rtems_interrupt_lock_acquire_isr(), 177
rtems_interrupt_lock_context, 44
RTEMS_INTERRUPT_LOCK_DECLARE(), 180
RTEMS_INTERRUPT_LOCK_DEFINE(), 181
rtems_interrupt_lock_destroy(), 173
rtems_interrupt_lock_initialize(), 172
RTEMS_INTERRUPT_LOCK_INITIALIZER(), 182
rtems_interrupt_lock_interrupt_disable(), 179
RTEMS_INTERRUPT_LOCK_MEMBER(), 183
RTEMS_INTERRUPT_LOCK_REFERENCE(), 184
rtems_interrupt_lock_release(), 176
rtems_interrupt_lock_release_isr(), 178

rtems_interrupt_per_handler_routine, 45
rtems_interrupt_raise(), 198
rtems_interrupt_raise_on(), 199
rtems_interrupt_server_action, 45
rtems_interrupt_server_action_prepend(), 225
rtems_interrupt_server_config, 45
rtems_interrupt_server_control, 45
rtems_interrupt_server_create(), 212
rtems_interrupt_server_delete(), 218
rtems_interrupt_server_entry, 45
rtems_interrupt_server_entry_destroy(), 227
rtems_interrupt_server_entry_initialize(), 224
rtems_interrupt_server_entry_move(), 229
rtems_interrupt_server_entry_submit(), 228
rtems_interrupt_server_handler_install(), 213
rtems_interrupt_server_handler_iterate(), 222
rtems_interrupt_server_handler_remove(), 215
rtems_interrupt_server_initialize(), 210
rtems_interrupt_server_move(), 221
rtems_interrupt_server_request, 46
rtems_interrupt_server_request_destroy(), 232
rtems_interrupt_server_request_initialize(), 230
rtems_interrupt_server_request_set_vector(), 231
rtems_interrupt_server_request_submit(), 233
rtems_interrupt_server_resume(), 220
rtems_interrupt_server_set_affinity(), 216
rtems_interrupt_server_suspend(), 219
rtems_interrupt_set_affinity(), 205
rtems_interrupt_set_priority(), 202
rtems_interrupt_signal_variant, 46
rtems_interrupt_vector_disable(), 195
rtems_interrupt_vector_enable(), 194
rtems_interrupt_vector_is_enabled(), 193
rtems_interval, 32, 46
rtems_io_close(), 480
rtems_io_control(), 483
rtems_io_initialize(), 477
rtems_io_open(), 479
rtems_io_read(), 481
rtems_io_register_driver(), 474
rtems_io_register_name(), 478
rtems_io_unregister_driver(), 476
rtems_io_write(), 482
rtems_isr, 47, 157
rtems_isr_entry, 47
rtems_iterate_over_all_threads(), 146
RTEMS_LINKER_ROSET, 1010
RTEMS_LINKER_ROSET_CONTENT, 1016
RTEMS_LINKER_ROSET_DECLARE, 1009
RTEMS_LINKER_ROSET_ITEM, 1014
RTEMS_LINKER_ROSET_ITEM_DECLARE, 1011
RTEMS_LINKER_ROSET_ITEM_ORDERED, 1015
RTEMS_LINKER_ROSET_ITEM_ORDERED_DECLARE, 1012
RTEMS_LINKER_ROSET_ITEM_REFERENCE, 1013
RTEMS_LINKER_RWSET, 1018
RTEMS_LINKER_RWSET_CONTENT, 1024
RTEMS_LINKER_RWSET_DECLARE, 1017
RTEMS_LINKER_RWSET_ITEM, 1022
RTEMS_LINKER_RWSET_ITEM_DECLARE, 1019
RTEMS_LINKER_RWSET_ITEM_ORDERED, 1023
RTEMS_LINKER_RWSET_ITEM_ORDERED_DECLARE, 1020
RTEMS_LINKER_RWSET_ITEM_REFERENCE, 1021
RTEMS_LINKER_SET_BEGIN, 1003
RTEMS_LINKER_SET_END, 1004
RTEMS_LINKER_SET_FOREACH, 1008
RTEMS_LINKER_SET_IS_EMPTY, 1007
RTEMS_LINKER_SET_ITEM_COUNT, 1006
RTEMS_LINKER_SET_SIZE, 1005
RTEMS_LOCAL, 26
rtems_message_queue_broadcast(), 385
RTEMS_MESSAGE_QUEUE_BUFFER(), 391
rtems_message_queue_config, 47
rtems_message_queue_construct(), 375
rtems_message_queue_create(), 372
rtems_message_queue_delete(), 379
rtems_message_queue_flush(), 390
rtems_message_queue_get_number_pending(), 389
rtems_message_queue_ident(), 26, 377
rtems_message_queue_receive(), 387
rtems_message_queue_send(), 381
rtems_message_queue_urgent(), 383
rtems_mode, 47
rtems_mp_packet_classes, 47
rtems_mpci_entry, 48, 848
rtems_mpci_get_packet_entry, 48
rtems_mpci_initialization_entry, 48
rtems_mpci_receive_packet_entry, 48

rtems_mpci_return_packet_entry, 48
rtems_mpci_send_packet_entry, 48
rtems_mpci_table, 48
rtems_multiprocessing_announce(), 854
rtems_multiprocessing_table, 48
rtems_name, 25, 48
rtems_object_api_class_information, 48
rtems_object_api_maximum_class(), 913
rtems_object_api_minimum_class(), 912
rtems_object_get_api_class_name(), 915
rtems_object_get_api_name(), 914
rtems_object_get_class_information(), 916
rtems_object_get_classic_name(), 903
rtems_object_get_local_node(), 917
rtems_object_get_name(), 25, 904
rtems_object_id_api_maximum(), 911
rtems_object_id_api_minimum(), 910
rtems_object_id_get_api(), 27, 906
rtems_object_id_get_class(), 27, 907
rtems_object_id_get_index(), 27, 909
rtems_object_id_get_node(), 27, 908
RTEMS_OBJECT_ID_INITIAL(), 918
rtems_object_set_name(), 905
rtems_option, 49
rtems_packet_prefix, 49
rtems_panic(), 527
rtems_partition_create(), 420
rtems_partition_delete(), 425
rtems_partition_get_buffer(), 427
rtems_partition_ident(), 26, 423
rtems_partition_return_buffer(), 429
rtems_port_create(), 460
rtems_port_delete(), 463
rtems_port_external_to_internal(), 464
rtems_port_ident(), 26, 462
rtems_port_internal_to_external(), 465
rtems_printk_printer(), 493
rtems_put_char(), 489
rtems_putc(), 488
rtems_rate_monotonic_cancel(), 315
rtems_rate_monotonic_create(), 312
rtems_rate_monotonic_delete(), 316
rtems_rate_monotonic_get_statistics(), 321
rtems_rate_monotonic_get_status(), 319
rtems_rate_monotonic_ident(), 314
rtems_rate_monotonic_period(), 317
rtems_rate_monotonic_period_states, 49
rtems_rate_monotonic_period_statistics, 49
rtems_rate_monotonic_period_status, 50
rtems_rate_monotonic_report_statistics(), 325
rtems_rate_monotonic_report_statistics_with_plugin(), 326
rtems_rate_monotonic_reset_all_statistics(), 324
rtems_rate_monotonic_reset_statistics(), 323
rtems_region_create(), 438
rtems_region_delete(), 441
rtems_region_extend(), 442
rtems_region_get_free_information(), 450
rtems_region_get_information(), 449
rtems_region_get_segment(), 444
rtems_region_get_segment_size(), 452
rtems_region_ident(), 26, 440
rtems_region_resize_segment(), 447
rtems_region_return_segment(), 446
rtems_regulator_attributes, 50
rtems_regulator_create(), 832
rtems_regulator_delete(), 834
rtems_regulator_deliverer, 51
rtems_regulator_get_statistics(), 840
rtems_regulator_obtain_buffer(), 836
rtems_regulator_release_buffer(), 837
rtems_regulator_send(), 838
rtems_regulator_statistics, 51
rtems_resource_is_unlimited(), 778
rtems_resource_maximum_per_allocation(), 779
rtems_resource_unlimited(), 780
rtems_scheduler_add_processor(), 82
rtems_scheduler_get_maximum_priority(), 76
rtems_scheduler_get_processor(), 79
rtems_scheduler_get_processor_maximum(), 80
rtems_scheduler_get_processor_set(), 81
rtems_scheduler_ident(), 72
rtems_scheduler_ident_by_processor(), 73
rtems_scheduler_ident_by_processor_set(), 74
rtems_scheduler_map_priority_from_posix(), 78
rtems_scheduler_map_priority_to_posix(), 77
rtems_scheduler_remove_processor(), 83
rtems_semaphore_create(), 335
rtems_semaphore_delete(), 341
rtems_semaphore_flush(), 348
rtems_semaphore_ident(), 26, 339

rtems_semaphore_obtain(), 343
rtems_semaphore_release(), 346
rtems_semaphore_set_priority(), 350
rtems_shutdown_executive(), 528
rtems_signal_catch(), 410
rtems_signal_send(), 412
rtems_signal_set, 51, 405
rtems_stack_allocate_hook, 52
rtems_stack_allocate_init_hook, 52
rtems_stack_free_hook, 52
rtems_status_code, 52, 1027
rtems_status_text(), 1028
rtems_task, 53, 102
rtems_task_argument, 54
rtems_task_begin_extension, 54, 546
rtems_task_config, 54
rtems_task_construct(), 113
rtems_task_create(), 109
rtems_task_create_extension, 55
rtems_task_create_extension(), 544
rtems_task_delete(), 122
rtems_task_delete_extension, 56, 547
rtems_task_entry, 56
rtems_task_exit(), 124
rtems_task_exitted_extension, 56, 547
rtems_task_get_affinity(), 139
rtems_task_get_note(), 107, 148
rtems_task_get_priority(), 130
rtems_task_get_scheduler(), 136
rtems_task_ident(), 26, 115
rtems_task_is_suspended(), 127
rtems_task_iterate(), 143
rtems_task_mode, 100
rtems_task_mode(), 132
rtems_task_priority, 57, 100
rtems_task_restart(), 120
rtems_task_restart_extension, 57, 545
rtems_task_resume(), 126
rtems_task_self(), 117
rtems_task_set_affinity(), 141
rtems_task_set_note(), 107, 149
rtems_task_set_priority(), 128
rtems_task_set_scheduler(), 137
rtems_task_start(), 118
rtems_task_start_extension, 57, 545
RTEMS_TASK_STORAGE_SIZE(), 144
rtems_task_suspend(), 125
rtems_task_switch_extension, 57, 546
rtems_task_terminate_extension, 58, 547
rtems_task_variable_add(), 107, 150
rtems_task_variable_delete(), 107, 152
rtems_task_variable_get(), 107, 151
rtems_task_visitor, 58
rtems_task_wake_after(), 134
rtems_task_wake_when(), 135
rtems_tcb, 59
rtems_time_of_day, 32, 59, 238
rtems_timer_cancel(), 284
rtems_timer_create(), 281
rtems_timer_delete(), 285
rtems_timer_fire_after(), 286
rtems_timer_fire_when(), 288
rtems_timer_get_information(), 297
rtems_timer_ident(), 26, 283
rtems_timer_information, 59
rtems_timer_initiate_server(), 290
rtems_timer_reset(), 296
rtems_timer_server_fire_after(), 292
rtems_timer_server_fire_when(), 294
rtems_timer_service_routine, 59, 277
rtems_timer_service_routine_entry, 60
rtems_timespec_add_to(), 963
rtems_timespec_divide(), 965
rtems_timespec_divide_by_integer(), 966
rtems_timespec_equal_to(), 969
rtems_timespec_from_ticks(), 973
rtems_timespec_get_nanoseconds(), 971
rtems_timespec_get_seconds(), 970
rtems_timespec_greater_than(), 968
rtems_timespec_is_valid(), 962
rtems_timespec_less_than(), 967
rtems_timespec_set(), 960
rtems_timespec_subtract(), 964
rtems_timespec_to_ticks(), 972
rtems_timespec_zero(), 961
rtems_vector_number, 60, 157
running, **1044**
runtime driver registration, 470

## S

sbintime_t, 238
scenario, **1044**
schedulable, **1044**
schedule, **1044**
scheduled task, **1044**
scheduler, **1044**
scheduler instance, **1044**
scheduling, 60
scheduling algorithms, 63
scheduling mechanisms, 64
segment, 433
segments, **1045**

semantics, **1045**
semaphore, **1045**
Semaphore Control Block, **1045**
semaphores, 326
send buffer to regulator for delivery, 838
send message to a queue, 381
send signal set, 412
separate work areas, 587
set priority by scheduler for a semaphore, 350
set scheduling parameters, 990
set struct timespec instance, 960
set task mode, 132
set task notepad entry, 149
set task preemption mode, 132
set task priority, 128
shared memory, **1045**
shutdown RTEMS, 528
signal, **1045**
signal set, 405, **1045**
signals, 402
SIS, **1045**
SMCB, **1045**
SMP, 854, **1045**
SMP barriers, **1045**
SMP locks, **1045**
soft real-time system, **1045**
software component, **1045**
software item, **1046**
software product, **1046**
software unit, **1046**
source code, **1046**
SPARC, **1046**
special device services, 483
sporadic task, 302, **1046**
stack, 879, **1046**
Stack Bounds Checker, 879
start current period, 317
start multitasking, 95
starting a task, 118
status code, **1046**
Status Codes, 1024
Strong APA, **1046**
struct bintime, 238
struct timespec, 238
struct timeval, 238
suspend, **1047**
suspending a task, 125
Symmetric Multiprocessing, 854
symmetric multiprocessing (*SMP*), 26
synchronous, **1047**
system call, **1047**
system-on-chip, **1047**

## T

target, **1047**
target architecture, **1047**
TAS, **1047**
task, 99–101, **1047**
task affinity, 858
task arguments, 102
task attributes, 103
Task Control Block, **1047**
task entry, **1047**
task life states, 101
task memory, 99
task migration, 858, **1047**
task mode, 100, 104
task name, 100
task priority, 64, 100, **1048**
task private data, 150, 152
task private variable, 150, 152
task processor affinity, **1048**
task prototype, 102
task scheduling, 60
task stack allocator, 699, 704
task stack allocator for IDLE tasks, 701
task stack deallocator, 703
task state transitions, 65
task states, 100
task switch, **1048**
tasks, 95
TCB, **1048**
TCB extension area, 543
thread, **1048**
thread affinity, 858
thread dispatch, **1048**
thread migration, 858
thread queues, 30
tick, **1048**
tick quantum, 583
ticks per timeslice, 586
tightly-coupled, **1048**
time, 31
timeout, **1048**
timeouts, 239
timer, **1048**
Timer Control Block, **1048**
Timer_Classes, 37
timers, 274
timeslice, **1048**
timeslicing, 64, 101, 238, **1048**

TImespec Helpers, 953
TLS, **1048**
TMCB, **1048**
transient overload, **1049**
TTAS, **1049**

## U

unblock all tasks waiting on a semaphore, 348
unified work areas, 587
uniprocesor, 26
Unix epoch, **1049**
unlock a semaphore, 346
unregister a device driver, 476
user extension set, 543
User Extension Table, **1049**
user extensions, 540, **1049**
User Initialization Tasks Table, **1049**
user-provided, **1049**
user-supplied, **1049**

## V

vector, **1049**
vprintk(), 492

## W

wait at a barrier, 363
wait queue, **1049**
wake up after a count of clock ticks, 134
wake up at a wall time, 135
Weak APA, **1049**
write to a device, 482

## Y

YAML, **1049**
yield, **1049**

## Z

zero C Program Heap, 591
zero RTEMS Workspace, 591