



RTEMS C User Documentation

Release 4.11.3

©Copyright 2016, RTEMS Project (built 15th February 2018)

CONTENTS

I	RTEMS C User's Guide	1
1	Preface	5
2	Overview	9
2.1	Introduction	10
2.2	Real-time Application Systems	11
2.3	Real-time Executive	12
2.4	RTEMS Application Architecture	13
2.5	RTEMS Internal Architecture	14
2.6	User Customization and Extensibility	15
2.7	Portability	16
2.8	Memory Requirements	17
2.9	Audience	18
2.10	Conventions	19
2.11	Manual Organization	20
3	Key Concepts	23
3.1	Introduction	24
3.2	Objects	25
3.2.1	Object Names	25
3.2.2	Object IDs	25
3.2.2.1	Thirty-Two Object ID Format	25
3.2.2.2	Sixteen Bit Object ID Format	26
3.2.3	Object ID Description	26
3.3	Communication and Synchronization	27
3.4	Time	28
3.5	Memory Management	29
4	RTEMS Data Types	31
4.1	Introduction	32
4.2	List of Data Types	33
5	Scheduling Concepts	37
5.1	Introduction	38
5.2	Scheduling Algorithms	39
5.2.1	Priority Scheduling	39
5.2.2	Deterministic Priority Scheduler	39
5.2.3	Simple Priority Scheduler	40

5.2.4	Simple SMP Priority Scheduler	40
5.2.5	Earliest Deadline First Scheduler	40
5.2.6	Constant Bandwidth Server Scheduling (CBS)	40
5.3	Scheduling Modification Mechanisms	42
5.3.1	Task Priority and Scheduling	42
5.3.2	Preemption	42
5.3.3	Timeslicing	42
5.3.4	Manual Round-Robin	42
5.4	Dispatching Tasks	43
5.5	Task State Transitions	44
6	Initialization Manager	47
6.1	Introduction	48
6.2	Background	49
6.2.1	Initialization Tasks	49
6.2.2	System Initialization	49
6.2.3	The Idle Task	49
6.2.4	Initialization Manager Failure	49
6.3	Operations	51
6.3.1	Initializing RTEMS	51
6.3.2	Shutting Down RTEMS	52
6.4	Directives	53
6.4.1	INITIALIZE_EXECUTIVE - Initialize RTEMS	54
6.4.2	SHUTDOWN_EXECUTIVE - Shutdown RTEMS	55
7	Task Manager	57
7.1	Introduction	58
7.2	Background	59
7.2.1	Task Definition	59
7.2.2	Task Control Block	59
7.2.3	Task States	59
7.2.4	Task Priority	59
7.2.5	Task Mode	60
7.2.6	Accessing Task Arguments	60
7.2.7	Floating Point Considerations	61
7.2.8	Per Task Variables	61
7.2.9	Building a Task Attribute Set	62
7.2.10	Building a Mode and Mask	62
7.3	Operations	64
7.3.1	Creating Tasks	64
7.3.2	Obtaining Task IDs	64
7.3.3	Starting and Restarting Tasks	64
7.3.4	Suspending and Resuming Tasks	64
7.3.5	Delaying the Currently Executing Task	64
7.3.6	Changing Task Priority	65
7.3.7	Changing Task Mode	65
7.3.8	Notepad Locations	65
7.3.9	Task Deletion	65
7.3.10	Transition Advice for Obsolete Directives	65
7.3.10.1	Notepads	65
7.4	Directives	67
7.4.1	TASK_CREATE - Create a task	68

7.4.2	TASK_IDENT - Get ID of a task	70
7.4.3	TASK_SELF - Obtain ID of caller	71
7.4.4	TASK_START - Start a task	72
7.4.5	TASK_RESTART - Restart a task	73
7.4.6	TASK_DELETE - Delete a task	74
7.4.7	TASK_SUSPEND - Suspend a task	75
7.4.8	TASK_RESUME - Resume a task	76
7.4.9	TASK_IS_SUSPENDED - Determine if a task is Suspended	77
7.4.10	TASK_SET_PRIORITY - Set task priority	78
7.4.11	TASK_MODE - Change the current task mode	79
7.4.12	TASK_GET_NOTE - Get task notepad entry	80
7.4.13	TASK_SET_NOTE - Set task notepad entry	81
7.4.14	TASK_WAKE_AFTER - Wake up after interval	82
7.4.15	TASK_WAKE_WHEN - Wake up when specified	83
7.4.16	ITERATE_OVER_ALL_THREADS - Iterate Over Tasks	84
7.4.17	TASK_VARIABLE_ADD - Associate per task variable	85
7.4.18	TASK_VARIABLE_GET - Obtain value of a per task variable	86
7.4.19	TASK_VARIABLE_DELETE - Remove per task variable	87
8	Interrupt Manager	89
8.1	Introduction	90
8.2	Background	91
8.2.1	Processing an Interrupt	91
8.2.2	RTEMS Interrupt Levels	91
8.2.3	Disabling of Interrupts by RTEMS	91
8.3	Operations	93
8.3.1	Establishing an ISR	93
8.3.2	Directives Allowed from an ISR	93
8.4	Directives	95
8.4.1	INTERRUPT_CATCH - Establish an ISR	96
8.4.2	INTERRUPT_DISABLE - Disable Interrupts	97
8.4.3	INTERRUPT_ENABLE - Enable Interrupts	98
8.4.4	INTERRUPT_FLASH - Flash Interrupts	99
8.4.5	INTERRUPT_LOCAL_DISABLE - Disable Interrupts on Current Processor	100
8.4.6	INTERRUPT_LOCAL_ENABLE - Enable Interrupts on Current Processor	101
8.4.7	INTERRUPT_LOCK_INITIALIZE - Initialize an ISR Lock	102
8.4.8	INTERRUPT_LOCK_ACQUIRE - Acquire an ISR Lock	103
8.4.9	INTERRUPT_LOCK_RELEASE - Release an ISR Lock	104
8.4.10	INTERRUPT_LOCK_ACQUIRE_ISR - Acquire an ISR Lock from ISR	105
8.4.11	INTERRUPT_LOCK_RELEASE_ISR - Release an ISR Lock from ISR	106
8.4.12	INTERRUPT_IS_IN_PROGRESS - Is an ISR in Progress	107
9	Clock Manager	109
9.1	Introduction	110
9.2	Background	111
9.2.1	Required Support	111
9.2.2	Time and Date Data Structures	111
9.2.3	Clock Tick and Timeslicing	111
9.2.4	Delays	111
9.2.5	Timeouts	111
9.3	Operations	113
9.3.1	Announcing a Tick	113

9.3.2	Setting the Time	113
9.3.3	Obtaining the Time	113
9.4	Directives	114
9.4.1	CLOCK_SET - Set date and time	115
9.4.2	CLOCK_GET - Get date and time information	116
9.4.3	CLOCK_GET_TOD - Get date and time in TOD format	117
9.4.4	CLOCK_GET_TOD_TIMEVAL - Get date and time in timeval format	118
9.4.5	CLOCK_GET_SECONDS_SINCE_EPOCH - Get seconds since epoch	119
9.4.6	CLOCK_GET_TICKS_PER_SECOND - Get ticks per second	120
9.4.7	CLOCK_GET_TICKS_SINCE_BOOT - Get current ticks counter value	121
9.4.8	CLOCK_TICK_LATER - Get tick value in the future	122
9.4.9	CLOCK_TICK_LATER_USEC - Get tick value in the future in microseconds	123
9.4.10	CLOCK_TICK_BEFORE - Is tick value is before a point in time	124
9.4.11	CLOCK_GET_UPTIME - Get the time since boot	125
9.4.12	CLOCK_GET_UPTIME_TIMEVAL - Get the time since boot in timeval format	126
9.4.13	CLOCK_GET_UPTIME_SECONDS - Get the seconds since boot	127
9.4.14	CLOCK_GET_UPTIME_NANOSECONDS - Get the nanoseconds since boot	128
10	Timer Manager	129
10.1	Introduction	130
10.2	Background	131
10.2.1	Required Support	131
10.2.2	Timers	131
10.2.3	Timer Server	131
10.2.4	Timer Service Routines	131
10.3	Operations	132
10.3.1	Creating a Timer	132
10.3.2	Obtaining Timer IDs	132
10.3.3	Initiating an Interval Timer	132
10.3.4	Initiating a Time of Day Timer	132
10.3.5	Canceling a Timer	132
10.3.6	Resetting a Timer	132
10.3.7	Initiating the Timer Server	132
10.3.8	Deleting a Timer	132
10.4	Directives	133
10.4.1	TIMER_CREATE - Create a timer	134
10.4.2	TIMER_IDENT - Get ID of a timer	135
10.4.3	TIMER_CANCEL - Cancel a timer	136
10.4.4	TIMER_DELETE - Delete a timer	137
10.4.5	TIMER_FIRE_AFTER - Fire timer after interval	138
10.4.6	TIMER_FIRE_WHEN - Fire timer when specified	139
10.4.7	TIMER_INITIATE_SERVER - Initiate server for task-based timers	140
10.4.8	TIMER_SERVER_FIRE_AFTER - Fire task-based timer after interval	141
10.4.9	TIMER_SERVER_FIRE_WHEN - Fire task-based timer when specified	142
10.4.10	TIMER_RESET - Reset an interval timer	143
11	Rate Monotonic Manager	145
11.1	Introduction	146
11.2	Background	147
11.2.1	Rate Monotonic Manager Required Support	147
11.2.2	Period Statistics	147

11.2.3	Rate Monotonic Manager Definitions	148
11.2.4	Rate Monotonic Scheduling Algorithm	148
11.2.5	Schedulability Analysis	149
11.2.5.1	Assumptions	149
11.2.5.2	Processor Utilization Rule	149
11.2.5.3	Processor Utilization Rule Example	149
11.2.5.4	First Deadline Rule	150
11.2.5.5	First Deadline Rule Example	150
11.2.5.6	Relaxation of Assumptions	151
11.2.5.7	Further Reading	151
11.3	Operations	152
11.3.1	Creating a Rate Monotonic Period	152
11.3.2	Manipulating a Period	152
11.3.3	Obtaining the Status of a Period	152
11.3.4	Canceling a Period	152
11.3.5	Deleting a Rate Monotonic Period	152
11.3.6	Examples	152
11.3.7	Simple Periodic Task	152
11.3.8	Task with Multiple Periods	153
11.4	Directives	155
11.4.1	RATE_MONOTONIC_CREATE - Create a rate monotonic period	156
11.4.2	RATE_MONOTONIC_IDENT - Get ID of a period	157
11.4.3	RATE_MONOTONIC_CANCEL - Cancel a period	158
11.4.4	RATE_MONOTONIC_DELETE - Delete a rate monotonic period	159
11.4.5	RATE_MONOTONIC_PERIOD - Conclude current/Start next period	160
11.4.6	RATE_MONOTONIC_GET_STATUS - Obtain status from a period	161
11.4.7	RATE_MONOTONIC_GET_STATISTICS - Obtain statistics from a period	162
11.4.8	RATE_MONOTONIC_RESET_STATISTICS - Reset statistics for a period	163
11.4.9	RATE_MONOTONIC_RESET_ALL_STATISTICS - Reset statistics for all periods	164
11.4.10	RATE_MONOTONIC_REPORT_STATISTICS - Print period statistics report	165
12	Semaphore Manager	167
12.1	Introduction	168
12.2	Background	169
12.2.1	Nested Resource Access	169
12.2.2	Priority Inversion	169
12.2.3	Priority Inheritance	169
12.2.4	Priority Ceiling	170
12.2.5	Multiprocessor Resource Sharing Protocol	170
12.2.6	Building a Semaphore Attribute Set	171
12.2.7	Building a SEMAPHORE_OBTAIN Option Set	171
12.3	Operations	173
12.3.1	Creating a Semaphore	173
12.3.2	Obtaining Semaphore IDs	173
12.3.3	Acquiring a Semaphore	173
12.3.4	Releasing a Semaphore	173
12.3.5	Deleting a Semaphore	174
12.4	Directives	175
12.4.1	SEMAPHORE_CREATE - Create a semaphore	176
12.4.2	SEMAPHORE_IDENT - Get ID of a semaphore	178

12.4.3	SEMAPHORE_DELETE - Delete a semaphore	179
12.4.4	SEMAPHORE_OBTAIN - Acquire a semaphore	180
12.4.5	SEMAPHORE_RELEASE - Release a semaphore	181
12.4.6	SEMAPHORE_FLUSH - Unblock all tasks waiting on a semaphore	182
12.4.7	SEMAPHORE_SET_PRIORITY - Set priority by scheduler for a semaphore	183
13	Barrier Manager	185
13.1	Introduction	186
13.2	Background	187
13.2.1	Automatic Versus Manual Barriers	187
13.2.2	Building a Barrier Attribute Set	187
13.3	Operations	188
13.3.1	Creating a Barrier	188
13.3.2	Obtaining Barrier IDs	188
13.3.3	Waiting at a Barrier	188
13.3.4	Releasing a Barrier	188
13.3.5	Deleting a Barrier	188
13.4	Directives	189
13.4.1	BARRIER_CREATE - Create a barrier	190
13.4.2	BARRIER_IDENT - Get ID of a barrier	191
13.4.3	BARRIER_DELETE - Delete a barrier	192
13.4.4	BARRIER_OBTAIN - Acquire a barrier	193
13.4.5	BARRIER_RELEASE - Release a barrier	194
14	Message Manager	195
14.1	Introduction	196
14.2	Background	197
14.2.1	Messages	197
14.2.2	Message Queues	197
14.2.3	Building a Message Queue Attribute Set	197
14.2.4	Building a MESSAGE_QUEUE_RECEIVE Option Set	197
14.3	Operations	199
14.3.1	Creating a Message Queue	199
14.3.2	Obtaining Message Queue IDs	199
14.3.3	Receiving a Message	199
14.3.4	Sending a Message	199
14.3.5	Broadcasting a Message	199
14.3.6	Deleting a Message Queue	200
14.4	Directives	201
14.4.1	MESSAGE_QUEUE_CREATE - Create a queue	202
14.4.2	MESSAGE_QUEUE_IDENT - Get ID of a queue	203
14.4.3	MESSAGE_QUEUE_DELETE - Delete a queue	204
14.4.4	MESSAGE_QUEUE_SEND - Put message at rear of a queue	205
14.4.5	MESSAGE_QUEUE_URGENT - Put message at front of a queue	206
14.4.6	MESSAGE_QUEUE_BROADCAST - Broadcast N messages to a queue . . .	207
14.4.7	MESSAGE_QUEUE_RECEIVE - Receive message from a queue	208
14.4.8	MESSAGE_QUEUE_GET_NUMBER_PENDING - Get number of messages pending on a queue	209
14.4.9	MESSAGE_QUEUE_FLUSH - Flush all messages on a queue	210
15	Event Manager	211
15.1	Introduction	212

15.2	Background	213
15.2.1	Event Sets	213
15.2.2	Building an Event Set or Condition	213
15.2.3	Building an EVENT_RECEIVE Option Set	213
15.3	Operations	214
15.3.1	Sending an Event Set	214
15.3.2	Receiving an Event Set	214
15.3.3	Determining the Pending Event Set	214
15.3.4	Receiving all Pending Events	214
15.4	Directives	215
15.4.1	EVENT_SEND - Send event set to a task	216
15.4.2	EVENT_RECEIVE - Receive event condition	217
16	Signal Manager	219
16.1	Introduction	220
16.2	Background	221
16.2.1	Signal Manager Definitions	221
16.2.2	A Comparison of ASRs and ISRs	221
16.2.3	Building a Signal Set	221
16.2.4	Building an ASR Mode	221
16.3	Operations	223
16.3.1	Establishing an ASR	223
16.3.2	Sending a Signal Set	223
16.3.3	Processing an ASR	223
16.4	Directives	224
16.4.1	SIGNAL_CATCH - Establish an ASR	225
16.4.2	SIGNAL_SEND - Send signal set to a task	226
17	Partition Manager	227
17.1	Introduction	228
17.2	Background	229
17.2.1	Partition Manager Definitions	229
17.2.2	Building a Partition Attribute Set	229
17.3	Operations	230
17.3.1	Creating a Partition	230
17.3.2	Obtaining Partition IDs	230
17.3.3	Acquiring a Buffer	230
17.3.4	Releasing a Buffer	230
17.3.5	Deleting a Partition	230
17.4	Directives	231
17.4.1	PARTITION_CREATE - Create a partition	232
17.4.2	PARTITION_IDENT - Get ID of a partition	233
17.4.3	PARTITION_DELETE - Delete a partition	234
17.4.4	PARTITION_GET_BUFFER - Get buffer from a partition	235
17.4.5	PARTITION_RETURN_BUFFER - Return buffer to a partition	236
18	Region Manager	237
18.1	Introduction	238
18.2	Background	239
18.2.1	Region Manager Definitions	239
18.2.2	Building an Attribute Set	239
18.2.3	Building an Option Set	239

18.3	Operations	240
18.3.1	Creating a Region	240
18.3.2	Obtaining Region IDs	240
18.3.3	Adding Memory to a Region	240
18.3.4	Acquiring a Segment	240
18.3.5	Releasing a Segment	240
18.3.6	Obtaining the Size of a Segment	241
18.3.7	Changing the Size of a Segment	241
18.3.8	Deleting a Region	241
18.4	Directives	242
18.4.1	REGION_CREATE - Create a region	243
18.4.2	REGION_IDENT - Get ID of a region	244
18.4.3	REGION_DELETE - Delete a region	245
18.4.4	REGION_EXTEND - Add memory to a region	246
18.4.5	REGION_GET_SEGMENT - Get segment from a region	247
18.4.6	REGION_RETURN_SEGMENT - Return segment to a region	248
18.4.7	REGION_GET_SEGMENT_SIZE - Obtain size of a segment	249
18.4.8	REGION_RESIZE_SEGMENT - Change size of a segment	250
19	Dual-Ported Memory Manager	251
19.1	Introduction	252
19.2	Background	253
19.3	Operations	254
19.3.1	Creating a Port	254
19.3.2	Obtaining Port IDs	254
19.3.3	Converting an Address	254
19.3.4	Deleting a DPMA Port	254
19.4	Directives	255
19.4.1	PORT_CREATE - Create a port	256
19.4.2	PORT_IDENT - Get ID of a port	257
19.4.3	PORT_DELETE - Delete a port	258
19.4.4	PORT_EXTERNAL_TO_INTERNAL - Convert external to internal address	259
19.4.5	PORT_INTERNAL_TO_EXTERNAL - Convert internal to external address	260
20	I/O Manager	261
20.1	Introduction	262
20.2	Background	263
20.2.1	Device Driver Table	263
20.2.2	Major and Minor Device Numbers	263
20.2.3	Device Names	263
20.2.4	Device Driver Environment	263
20.2.5	Runtime Driver Registration	263
20.2.6	Device Driver Interface	264
20.2.7	Device Driver Initialization	264
20.3	Operations	265
20.3.1	Register and Lookup Name	265
20.3.2	Accessing an Device Driver	265
20.4	Directives	266
20.4.1	IO_REGISTER_DRIVER - Register a device driver	267
20.4.2	IO_UNREGISTER_DRIVER - Unregister a device driver	268
20.4.3	IO_INITIALIZE - Initialize a device driver	269
20.4.4	IO_REGISTER_NAME - Register a device	270

20.4.5	IO_LOOKUP_NAME - Lookup a device	271
20.4.6	IO_OPEN - Open a device	272
20.4.7	IO_CLOSE - Close a device	273
20.4.8	IO_READ - Read from a device	274
20.4.9	IO_WRITE - Write to a device	275
20.4.10	IO_CONTROL - Special device services	276
21	Fatal Error Manager	277
21.1	Introduction	278
21.2	Background	279
21.3	Operations	280
21.3.1	Announcing a Fatal Error	280
21.4	Directives	281
21.4.1	FATAL_ERROR_OCCURRED - Invoke the fatal error handler	282
21.4.2	FATAL - Invoke the fatal error handler with error source	283
21.4.3	EXCEPTION_FRAME_PRINT - Prints the exception frame	284
21.4.4	FATAL_SOURCE_TEXT - Returns a text for a fatal source	285
21.4.5	INTERNAL_ERROR_TEXT - Returns a text for an internal error code	286
22	Board Support Packages	287
22.1	Introduction	288
22.2	Reset and Initialization	289
22.2.1	Interrupt Stack Requirements	289
22.2.2	Processors with a Separate Interrupt Stack	290
22.2.3	Processors Without a Separate Interrupt Stack	290
22.3	Device Drivers	291
22.3.1	Clock Tick Device Driver	291
22.4	User Extensions	292
22.5	Multiprocessor Communications Interface (MPCI)	293
22.5.1	Tightly-Coupled Systems	293
22.5.2	Loosely-Coupled Systems	293
22.5.3	Systems with Mixed Coupling	293
22.5.4	Heterogeneous Systems	293
23	User Extensions Manager	295
23.1	Introduction	296
23.2	Background	297
23.2.1	Extension Sets	297
23.2.2	TCB Extension Area	297
23.2.3	Extensions	298
23.2.3.1	TASK_CREATE Extension	298
23.2.3.2	TASK_START Extension	298
23.2.3.3	TASK_RESTART Extension	299
23.2.3.4	TASK_DELETE Extension	299
23.2.3.5	TASK_SWITCH Extension	299
23.2.3.6	TASK_BEGIN Extension	299
23.2.3.7	TASK_EXITTED Extension	300
23.2.3.8	FATAL Error Extension	300
23.2.4	Order of Invocation	300
23.3	Operations	302
23.3.1	Creating an Extension Set	302
23.3.2	Obtaining Extension Set IDs	302

23.3.3	Deleting an Extension Set	302
23.4	Directives	303
23.4.1	EXTENSION_CREATE - Create a extension set	304
23.4.2	EXTENSION_IDENT - Get ID of a extension set	305
23.4.3	EXTENSION_DELETE - Delete a extension set	306
24	Configuring a System	307
24.1	Introduction	308
24.2	Default Value Selection Philosophy	309
24.3	Sizing the RTEMS Workspace	310
24.4	Potential Issues with RTEMS Workspace Size Estimation	311
24.5	Format to be followed for making changes in this file	312
24.6	Configuration Example	313
24.7	Unlimited Objects	315
24.7.1	Per Object Class Unlimited Object Instances	315
24.7.2	Unlimited Object Instances	316
24.7.3	Enable Unlimited Object Instances	316
24.7.4	Specify Unlimited Objects Allocation Size	316
24.8	Classic API Configuration	317
24.8.1	Specify Maximum Classic API Tasks	317
24.8.2	Specify Maximum Classic API Timers	317
24.8.3	Specify Maximum Classic API Timers	317
24.8.4	Specify Maximum Classic API Semaphores	318
24.8.5	Specify Maximum Classic API Semaphores usable with MrsP	318
24.8.6	Specify Maximum Classic API Message Queues	318
24.8.7	Specify Maximum Classic API Barriers	318
24.8.8	Specify Maximum Classic API Periods	318
24.8.9	Specify Maximum Classic API Partitions	319
24.8.10	Specify Maximum Classic API Regions	319
24.8.11	Specify Maximum Classic API Ports	319
24.8.12	Specify Maximum Classic API User Extensions	319
24.9	Classic API Initialization Tasks Table Configuration	320
24.9.1	Instantiate Classic API Initialization Task Table	320
24.9.2	Specifying Classic API Initialization Task Entry Point	320
24.9.3	Specifying Classic API Initialization Task Name	320
24.9.4	Specifying Classic API Initialization Task Stack Size	320
24.9.5	Specifying Classic API Initialization Task Priority	321
24.9.6	Specifying Classic API Initialization Task Attributes	321
24.9.7	Specifying Classic API Initialization Task Modes	321
24.9.8	Specifying Classic API Initialization Task Arguments	321
24.9.9	Not Using Generated Initialization Tasks Table	322
24.10	POSIX API Configuration	323
24.10.1	Specify Maximum POSIX API Threads	323
24.10.2	Specify Maximum POSIX API Mutexes	323
24.10.3	Specify Maximum POSIX API Condition Variables	323
24.10.4	Specify Maximum POSIX API Keys	323
24.10.5	Specify Maximum POSIX API Timers	324
24.10.6	Specify Maximum POSIX API Queued Signals	324
24.10.7	Specify Maximum POSIX API Message Queues	324
24.10.8	Specify Maximum POSIX API Message Queue Descriptors	324
24.10.9	Specify Maximum POSIX API Semaphores	325

24.10.1	Specify Maximum POSIX API Barriers	325
24.10.1	Specify Maximum POSIX API Spinlocks	325
24.10.1	Specify Maximum POSIX API Read/Write Locks	325
24.11	POSIX Initialization Threads Table Configuration	326
24.11.1	Instantiate POSIX API Initialization Thread Table	326
24.11.2	Specifying POSIX API Initialization Thread Entry Point	326
24.11.3	Specifying POSIX API Initialization Thread Stack Size	326
24.11.4	Not Using Generated POSIX Initialization Threads Table	327
24.12	Basic System Information	328
24.12.1	Separate or Unified Work Areas	328
24.12.2	Length of Each Clock Tick	328
24.12.3	Specifying Timeslicing Quantum	328
24.12.4	Specifying the Number of Thread Priority Levels	329
24.12.5	Specifying the Minimum Task Size	329
24.12.6	Configuring the Size of the Interrupt Stack	329
24.12.7	Reserve Task/Thread Stack Memory Above Minimum	330
24.12.8	Automatically Zeroing the RTEMS Workspace and C Program Heap	330
24.12.9	Enable The Task Stack Usage Checker	330
24.12.1	Specify Application Specific User Extensions	331
24.13	Configuring Custom Task Stack Allocation	332
24.13.1	Custom Task Stack Allocator Initialization	332
24.13.2	Custom Task Stack Allocator	332
24.13.3	Custom Task Stack Deallocator	332
24.14	Configuring Memory for Classic API Message Buffers	333
24.14.1	Calculate Memory for a Single Classic Message API Message Queue	333
24.14.2	Reserve Memory for All Classic Message API Message Queues	333
24.15	Seldom Used Configuration Parameters	334
24.15.1	Specify Memory Overhead	334
24.15.2	Do Not Generate Configuration Information	334
24.16	C Library Support Configuration	335
24.16.1	Specify Maximum Number of File Descriptors	335
24.16.2	Disable POSIX Termios Support	335
24.16.3	Specify Maximum Termios Ports	335
24.17	File System Configuration Parameters	336
24.17.1	Providing Application Specific Mount Table	336
24.17.2	Configure devFS as Root File System	336
24.17.3	Specifying Maximum Devices for devFS	336
24.17.4	Disable File System Support	336
24.17.5	Use a Root IMFS with a Minimalistic Feature Set	337
24.17.6	Specify Block Size for IMFS	337
24.17.7	Disable Change Owner Support of Root IMFS	338
24.17.8	Disable Change Mode Support of Root IMFS	338
24.17.9	Disable Change Times Support of Root IMFS	338
24.17.1	Disable Create Hard Link Support of Root IMFS	338
24.17.1	Disable Create Symbolic Link Support of Root IMFS	338
24.17.1	Disable Read Symbolic Link Support of Root IMFS	338
24.17.1	Disable Rename Support of Root IMFS	339
24.17.1	Disable Directory Read Support of Root IMFS	339
24.17.1	Disable Mount Support of Root IMFS	339
24.17.1	Disable Unmount Support of Root IMFS	339
24.17.1	Disable Make Nodes Support of Root IMFS	339

24.17.1	Disable Make Files Support of Root IMFS	340
24.17.1	Disable Remove Nodes Support of Root IMFS	340
24.18	Block Device Cache Configuration	341
24.18.1	Enable Block Device Cache	341
24.18.2	Size of the Cache Memory	341
24.18.3	Minimum Size of a Buffer	341
24.18.4	Maximum Size of a Buffer	341
24.18.5	Swapout Task Swap Period	341
24.18.6	Swapout Task Maximum Block Hold Time	342
24.18.7	Swapout Task Priority	342
24.18.8	Maximum Blocks per Read-Ahead Request	342
24.18.9	Maximum Blocks per Write Request	342
24.18.1	Task Stack Size of the Block Device Cache Tasks	342
24.18.1	Read-Ahead Task Priority	343
24.18.1	Swapout Worker Task Count	343
24.18.1	Swapout Worker Task Priority	343
24.19	BSP Specific Settings	344
24.19.1	Disable BSP Configuration Settings	344
24.19.2	Specify BSP Supports sbrk()	344
24.19.3	Specify BSP Specific Idle Task	344
24.19.4	Specify BSP Suggested Value for IDLE Task Stack Size	344
24.19.5	Specify BSP Specific User Extensions	345
24.19.6	Specifying BSP Specific Interrupt Stack Size	345
24.19.7	Specifying BSP Specific Maximum Devices	345
24.19.8	BSP Recommends RTEMS Workspace be Cleared	345
24.19.9	Specify BSP Prerequisite Drivers	346
24.20	Idle Task Configuration	347
24.20.1	Specify Application Specific Idle Task Body	347
24.20.2	Specify Idle Task Stack Size	347
24.20.3	Specify Idle Task Performs Application Initialization	347
24.21	Scheduler Algorithm Configuration	348
24.21.1	Use Deterministic Priority Scheduler	348
24.21.2	Use Simple Priority Scheduler	348
24.21.3	Use Earliest Deadline First Scheduler	348
24.21.4	Use Constant Bandwidth Server Scheduler	349
24.21.5	Use Deterministic Priority SMP Scheduler	349
24.21.6	Use Simple SMP Priority Scheduler	349
24.21.7	Configuring a Scheduler Name	350
24.21.8	Configuring a User Provided Scheduler	350
24.21.9	Configuring Clustered Schedulers	351
24.22	SMP Specific Configuration Parameters	354
24.22.1	Enable SMP Support for Applications	354
24.22.2	Specify Maximum Processors in SMP System	354
24.23	Device Driver Table	355
24.23.1	Specifying the Maximum Number of Device Drivers	355
24.23.2	Enable Console Device Driver	355
24.23.3	Enable Clock Driver	355
24.23.4	Enable the Benchmark Timer Driver	356
24.23.5	Specify Clock and Benchmark Timer Drivers Are Not Needed	356
24.23.6	Enable Real-Time Clock Driver	356
24.23.7	Enable the Watchdog Device Driver	356

24.23.8	Enable the Graphics Frame Buffer Device Driver	357
24.23.9	Enable Stub Device Driver	357
24.23.10	Specify Application Prerequisite Device Drivers	357
24.23.11	Specify Extra Application Device Drivers	357
24.23.12	Enable /dev/null Device Driver	358
24.23.13	Enable /dev/zero Device Driver	358
24.23.14	Specifying Application Defined Device Driver Table	358
24.24	Multiprocessing Configuration	359
24.24.1	Specify Application Will Use Multiprocessing	359
24.24.2	Configure Node Number in Multiprocessor Configuration	359
24.24.3	Configure Maximum Node in Multiprocessor Configuration	359
24.24.4	Configure Maximum Global Objects in Multiprocessor Configuration	359
24.24.5	Configure Maximum Proxies in Multiprocessor Configuration	360
24.24.6	Configure MPCPI in Multiprocessor Configuration	360
24.24.7	Do Not Generate Multiprocessor Configuration Table	360
24.25	Ada Tasks	361
24.25.1	Specify Application Includes Ada Code	361
24.25.2	Specify the Maximum Number of Ada Tasks.	361
24.25.3	Specify the Maximum Fake Ada Tasks	361
24.26	PCI Library	362
24.27	Go Tasks	363
24.27.1	Specify Application Includes Go Code	363
24.27.2	Specify the maximum number of Go routines	363
24.27.3	Specify the maximum number of Go Channels	363
24.28	Configuration Data Structures	364
25	Multiprocessing Manager	365
25.1	Introduction	366
25.2	Background	367
25.2.1	Nodes	367
25.2.2	Global Objects	367
25.2.3	Global Object Table	367
25.2.4	Remote Operations	368
25.2.5	Proxies	368
25.2.6	Multiprocessor Configuration Table	369
25.3	Multiprocessor Communications Interface Layer	370
25.3.1	INITIALIZATION	370
25.3.2	GET_PACKET	371
25.3.3	RETURN_PACKET	371
25.3.4	RECEIVE_PACKET	371
25.3.5	SEND_PACKET	371
25.3.6	Supporting Heterogeneous Environments	372
25.4	Operations	373
25.4.1	Announcing a Packet	373
25.5	Directives	374
25.5.1	MULTIPROCESSING_ANNOUNCE - Announce the arrival of a packet	375
26	Symmetric Multiprocessing Services	377
26.1	Introduction	378
26.2	Background	379
26.2.1	Uniprocessor versus SMP Parallelism	379
26.2.2	Task Affinity	379

26.2.3	Task Migration	379
26.2.4	Clustered Scheduling	380
26.2.5	Task Priority Queues	381
26.2.6	Scheduler Helping Protocol	381
26.2.7	Critical Section Techniques and SMP	382
26.2.7.1	Disable Interrupts and Interrupt Locks	382
26.2.7.2	Highest Priority Task Assumption	383
26.2.7.3	Disable Preemption	383
26.2.8	Task Unique Data and SMP	384
26.2.8.1	Classic API Per Task Variables	384
26.2.9	OpenMP	384
26.2.10	Thread Dispatch Details	385
26.3	Operations	387
26.3.1	Setting Affinity to a Single Processor	387
26.4	Directives	388
26.4.1	GET_PROCESSOR_COUNT - Get processor count	389
26.4.2	GET_CURRENT_PROCESSOR - Get current processor index	390
26.4.3	SCHEDULER_IDENT - Get ID of a scheduler	391
26.4.4	SCHEDULER_GET_PROCESSOR_SET - Get processor set of a scheduler	392
26.4.5	TASK_GET_SCHEDULER - Get scheduler of a task	393
26.4.6	TASK_SET_SCHEDULER - Set scheduler of a task	394
26.4.7	TASK_GET_AFFINITY - Get task processor affinity	395
26.4.8	TASK_SET_AFFINITY - Set task processor affinity	396
27	PCI Library	397
27.1	Introduction	398
27.2	Background	399
27.2.1	Software Components	399
27.2.2	PCI Configuration	399
27.2.2.1	RTEMS Configuration selection	400
27.2.2.2	Auto Configuration	400
27.2.2.3	Read Configuration	401
27.2.2.4	Static Configuration	401
27.2.2.5	Peripheral Configuration	401
27.2.3	PCI Access	401
27.2.3.1	Configuration space	402
27.2.3.2	I/O space	402
27.2.3.3	Registers over Memory space	402
27.2.3.4	Access functions	402
27.2.3.5	PCI address translation	403
27.2.4	PCI Interrupt	403
27.2.5	PCI Shell command	403
28	Stack Bounds Checker	405
28.1	Introduction	406
28.2	Background	407
28.2.1	Task Stack	407
28.2.2	Execution	407
28.3	Operations	408
28.3.1	Initializing the Stack Bounds Checker	408
28.3.2	Checking for Blown Task Stack	408
28.3.3	Reporting Task Stack Usage	408

28.3.4	When a Task Overflows the Stack	408
28.4	Routines	409
28.4.1	STACK_CHECKER_IS_BLOWN - Has Current Task Blown Its Stack	409
28.4.2	STACK_CHECKER_REPORT_USAGE - Report Task Stack Usage	409
29	CPU Usage Statistics	411
29.1	Introduction	412
29.2	Background	413
29.3	Operations	414
29.3.1	Report CPU Usage Statistics	414
29.3.2	Reset CPU Usage Statistics	414
29.4	Directives	415
29.4.1	cpu_usage_report - Report CPU Usage Statistics	416
29.4.2	cpu_usage_reset - Reset CPU Usage Statistics	417
30	Object Services	419
30.1	Introduction	420
30.2	Background	421
30.2.1	APIs	421
30.2.2	Object Classes	421
30.2.3	Object Names	421
30.3	Operations	422
30.3.1	Decomposing and Recomposing an Object Id	422
30.3.2	Printing an Object Id	422
30.4	Directives	423
30.4.1	BUILD_NAME - Build object name from characters	424
30.4.2	OBJECT_GET_CLASSIC_NAME - Lookup name from id	425
30.4.3	OBJECT_GET_NAME - Obtain object name as string	426
30.4.4	OBJECT_SET_NAME - Set object name	427
30.4.5	OBJECT_ID_GET_API - Obtain API from Id	428
30.4.6	OBJECT_ID_GET_CLASS - Obtain Class from Id	429
30.4.7	OBJECT_ID_GET_NODE - Obtain Node from Id	430
30.4.8	OBJECT_ID_GET_INDEX - Obtain Index from Id	431
30.4.9	BUILD_ID - Build Object Id From Components	432
30.4.10	OBJECT_ID_API_MINIMUM - Obtain Minimum API Value	433
30.4.11	OBJECT_ID_API_MAXIMUM - Obtain Maximum API Value	434
30.4.12	OBJECT_API_MINIMUM_CLASS - Obtain Minimum Class Value	435
30.4.13	OBJECT_API_MAXIMUM_CLASS - Obtain Maximum Class Value	436
30.4.14	OBJECT_ID_API_MINIMUM_CLASS - Obtain Minimum Class Value for an API	437
30.4.15	OBJECT_ID_API_MAXIMUM_CLASS - Obtain Maximum Class Value for an API	438
30.4.16	OBJECT_GET_API_NAME - Obtain API Name	439
30.4.17	OBJECT_GET_API_CLASS_NAME - Obtain Class Name	440
30.4.18	OBJECT_GET_CLASS_INFORMATION - Obtain Class Information	441
31	Chains	443
31.1	Introduction	444
31.2	Background	445
31.2.1	Nodes	445
31.2.2	Controls	445
31.3	Operations	446

31.3.1	Multi-threading	446
31.3.2	Creating a Chain	446
31.3.3	Iterating a Chain	446
31.4	Directives	447
31.4.1	Initialize Chain With Nodes	448
31.4.2	Initialize Empty	449
31.4.3	Is Null Node ?	450
31.4.4	Head	451
31.4.5	Tail	452
31.4.6	Are Two Nodes Equal ?	453
31.4.7	Is the Chain Empty	454
31.4.8	Is this the First Node on the Chain ?	455
31.4.9	Is this the Last Node on the Chain ?	456
31.4.10	Does this Chain have only One Node ?	457
31.4.11	Returns the node count of the chain (unprotected)	458
31.4.12	Is this Node the Chain Head ?	459
31.4.13	Is this Node the Chain Tail ?	460
31.4.14	Extract a Node	461
31.4.15	Extract a Node (unprotected)	462
31.4.16	Get the First Node	463
31.4.17	Get the First Node (unprotected)	464
31.4.18	Insert a Node	465
31.4.19	Insert a Node (unprotected)	466
31.4.20	Append a Node	467
31.4.21	Append a Node (unprotected)	468
31.4.22	Prepend a Node	469
31.4.23	Prepend a Node (unprotected)	470
32	Red-Black Trees	471
32.1	Introduction	472
32.2	Background	473
32.2.1	Nodes	473
32.2.2	Controls	473
32.3	Operations	474
32.4	Directives	475
32.4.1	Documentation for the Red-Black Tree Directives	475
33	Timespec Helpers	477
33.1	Introduction	478
33.2	Background	479
33.2.1	Time Storage Conventions	479
33.3	Operations	480
33.3.1	Set and Obtain Timespec Value	480
33.3.2	Timespec Math	480
33.3.3	Comparing struct timespec Instances	480
33.3.4	Conversions and Validity Check	480
33.4	Directives	481
33.4.1	TIMESPEC_SET - Set struct timespec Instance	482
33.4.2	TIMESPEC_ZERO - Zero struct timespec Instance	483
33.4.3	TIMESPEC_IS_VALID - Check validity of a struct timespec instance	484
33.4.4	TIMESPEC_ADD_TO - Add Two struct timespec Instances	485
33.4.5	TIMESPEC_SUBTRACT - Subtract Two struct timespec Instances	486

33.4.6	TIMESPEC_DIVIDE - Divide Two struct timespec Instances	487
33.4.7	TIMESPEC_DIVIDE_BY_INTEGER - Divide a struct timespec Instance by an Integer	488
33.4.8	TIMESPEC_LESS_THAN - Less than operator	489
33.4.9	TIMESPEC_GREATER_THAN - Greater than operator	490
33.4.10	TIMESPEC_EQUAL_TO - Check equality of timespecs	491
33.4.11	TIMESPEC_GET_SECONDS - Get Seconds Portion of struct timespec In- stance	492
33.4.12	TIMESPEC_GET_NANOSECONDS - Get Nanoseconds Portion of the struct timespec Instance	493
33.4.13	TIMESPEC_TO_TICKS - Convert struct timespec Instance to Ticks	494
33.4.14	TIMESPEC_FROM_TICKS - Convert Ticks to struct timespec Representa- tion	495
34	Constant Bandwidth Server Scheduler API	497
34.1	Introduction	498
34.2	Background	499
34.2.1	Constant Bandwidth Server Definitions	499
34.2.2	Handling Periodic Tasks	499
34.2.3	Registering a Callback Function	499
34.2.4	Limitations	499
34.3	Operations	501
34.3.1	Setting up a server	501
34.3.2	Attaching Task to a Server	501
34.3.3	Detaching Task from a Server	501
34.3.4	Examples	501
34.4	Directives	503
34.4.1	CBS_INITIALIZE - Initialize the CBS library	504
34.4.2	CBS_CLEANUP - Cleanup the CBS library	505
34.4.3	CBS_CREATE_SERVER - Create a new bandwidth server	506
34.4.4	CBS_ATTACH_THREAD - Attach a thread to server	507
34.4.5	CBS_DETACH_THREAD - Detach a thread from server	508
34.4.6	CBS_DESTROY_SERVER - Destroy a bandwidth server	509
34.4.7	CBS_GET_SERVER_ID - Get an ID of a server	510
34.4.8	CBS_GET_PARAMETERS - Get scheduling parameters of a server	511
34.4.9	CBS_SET_PARAMETERS - Set scheduling parameters	512
34.4.10	CBS_GET_EXECUTION_TIME - Get elapsed execution time	513
34.4.11	CBS_GET_REMAINING_BUDGET - Get remaining execution time	514
34.4.12	CBS_GET_APPROVED_BUDGET - Get scheduler approved execution time	515
35	Directive Status Codes	517
35.1	Introduction	518
35.2	Directives	519
35.2.1	STATUS_TEXT - Returns the enumeration name for a status code	520
36	Linker Sets	521
36.1	Introduction	522
36.2	Background	523
36.3	Directives	524
36.3.1	RTEMS_LINKER_SET_BEGIN - Designator of the linker set begin marker	525
36.3.2	RTEMS_LINKER_SET_END - Designator of the linker set end marker . .	526
36.3.3	RTEMS_LINKER_SET_SIZE - The linker set size in characters	527

36.3.4	RTEMS_LINKER_ROSET_DECLARE - Declares a read-only linker set . . .	528
36.3.5	RTEMS_LINKER_ROSET - Defines a read-only linker set	529
36.3.6	RTEMS_LINKER_ROSET_ITEM_DECLARE - Declares a read-only linker set item	530
36.3.7	RTEMS_LINKER_ROSET_ITEM_REFERENCE - References a read-only linker set item	531
36.3.8	RTEMS_LINKER_ROSET_ITEM - Defines a read-only linker set item . . .	532
36.3.9	RTEMS_LINKER_ROSET_ITEM_ORDERED - Defines an ordered read-only linker set item	533
36.3.10	RTEMS_LINKER_RWSET_DECLARE - Declares a read-write linker set . .	534
36.3.11	RTEMS_LINKER_RWSET - Defines a read-write linker set	535
36.3.12	RTEMS_LINKER_RWSET_ITEM_DECLARE - Declares a read-write linker set item	536
36.3.13	RTEMS_LINKER_RWSET_ITEM_REFERENCE - References a read-write linker set item	537
36.3.14	RTEMS_LINKER_RWSET_ITEM - Defines a read-write linker set item . .	538
36.3.15	RTEMS_LINKER_RWSET_ITEM_ORDERED - Defines an ordered read-write linker set item	539
37	Example Application	541
38	Glossary	543
	Index	553

Part I

RTEMS C User's Guide

COPYRIGHT (c) 1988 - 2015.
On-Line Applications Research
Corporation (OAR).

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 <http://www.rtems.org/>. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the Community Project hosted at <http://www.rtems.org/>.

RTEMS Online Resources

Home	https://www.rtems.org/
Developers	https://devel.rtems.org/
Documentation	https://docs.rtems.org/
Bug Reporting	https://devel.rtems.org/query
Mailing Lists	https://lists.rtems.org/
Git Repositories	https://git.rtems.org/

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. 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). These two groups are currently working to-

gether with the IEEE P1003.4 committee to insure that the functionality of their proposed standards is adopted as the real-time extensions to POSIX.

This emerging 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 hardware 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 ex-

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 man-

agement. 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.

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 tic of a real-time system.

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 characteris-

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.

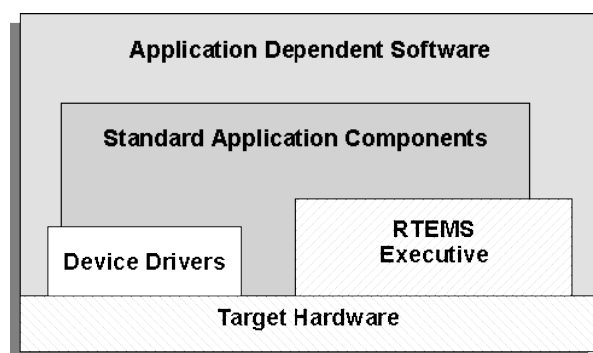
By proper grouping of responses to stimuli into separate tasks, a system can now asynchronously switch between independent streams of execution, directly responding to external stimuli as they occur. This allows the system design to meet critical performance specifications which are typically measured 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 addi-

tion, 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

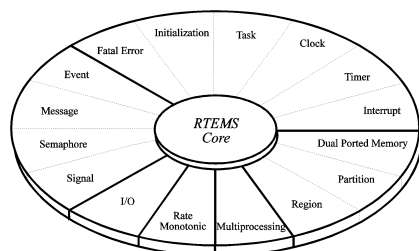
One important design goal of RTEMS was to provide a bridge between two critical layers of typical real-time systems. As shown in the following figure, RTEMS serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real-time applications 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 filesystems, 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 *Chapter 24 - Configuring a System* (page 307).

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 2:

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

Chapter 3:

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

Chapter 4:

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

Chapter 5:

Initialization Manager: describes the functionality and directives provided by the Initialization Manager.

Chapter 6:

Task Manager: describes the functionality and directives provided by the Task Manager.

Chapter 7:

Interrupt Manager: describes the functionality and directives provided by the Interrupt Manager.

Chapter 8:

Clock Manager: describes the functionality

and directives provided by the Clock Manager.

Chapter 9:

Timer Manager: describes the functionality and directives provided by the Timer Manager.

Chapter 10:

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

Chapter 11:

Semaphore Manager: describes the functionality and directives provided by the Semaphore Manager.

Chapter 12:

Barrier Manager: describes the functionality and directives provided by the Barrier Manager.

Chapter 13:

Message Manager: describes the functionality and directives provided by the Message Manager.

Chapter 14:

Event Manager: describes the functionality and directives provided by the Event Manager.

Chapter 15:

Signal Manager: describes the functionality and directives provided by the Signal Manager.

Chapter 16:

Partition Manager: describes the functionality and directives provided by the Partition Manager.

Chapter 17:

Region Manager: describes the functionality and directives provided by the Region Manager.

Chapter 18:

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

Chapter 19:

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

Chapter 20:

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

Chapter 21:

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

Chapter 22:

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

Chapter 23:

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

Chapter 24:

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 25:

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

Chapter 26:

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 27:

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 28:

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 29:

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

Chapter 30:

Constant Bandwidth Server Scheduler API.

Chapter 31:

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

Chapter 32:

Example Application: provides a template for simple RTEMS applications.

Chapter 33:

Glossary: defines terms used throughout this manual.

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 reusable “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... index:: `rtems_build_name`

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:

```
1 rtems_name my_name;
2 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:

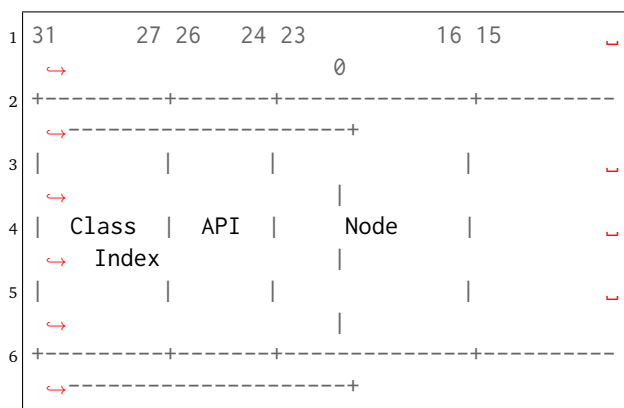
```
1 #include <rtems.h>
2 #include <rtems/bspIo.h>
3 void print_name(rtems_id id)
4 {
5     char buffer[10]; /* name assumed to
   ↪ be 10 characters or less */
6     char *result;
7     result = rtems_object_get_name( id,
   ↪ sizeof(buffer), buffer );
8     printk( "ID=0x%08x name=%s\n", id,
   ↪ ((result) ? result : "no name" ) );
9 }
```

3.2.2 Object IDs

An object ID is a unique 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 RTEMS services. Because of this, there are two object ID formats defined. Each target architecture specifies which format it will use. There is a thirty-two bit format which is used for most of the supported architectures and supports multiprocessor configurations. There is also a simpler sixteen bit format which is appropriate for smaller target architectures and does not support multiprocessor configurations.

3.2.2.1 Thirty-Two Object ID Format

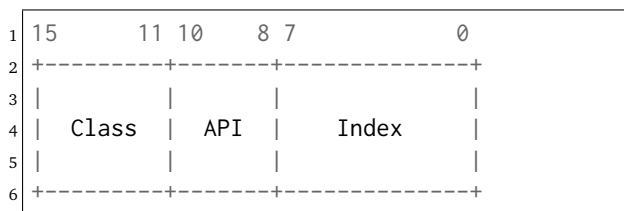
The thirty-two bit format for an object ID is composed of four parts: API, object class, node, and index. The data type `rtems_id` is used to store object IDs.



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.

3.2.2.2 Sixteen Bit Object ID Format

The sixteen bit format for an object ID is composed of three parts: API, object class, and index. The data type `rtems_id` is used to store object IDs.



The sixteen-bit format is designed to be as similar as possible to the thirty-two bit format. The differences are limited to the elimination of the node field and reduction of the index field from sixteen-bits to 8-bits. Thus the sixteen bit format only supports up to 255 object instances per API/Class combination and single processor systems. As this format is typically utilized by sixteen-bit processors with limited address space, this is more than enough object instances.

3.2.3 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:

```
uint32_t rtems_object_id_get_api( rtems_id id );
uint32_t rtems_object_id_get_class( rtems_id id );
uint32_t rtems_object_id_get_node( rtems_id id );
uint32_t rtems_object_id_get_index( rtems_id 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 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 tick. The frequency of clock ticks is completely application dependent and determines the granularity and accuracy of all interval and calendar time operations.

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 clock granularity 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 cal-

endar time in RTEMS services. See *Chapter 9 Section 2.2 - Time and Date Data Structures* (page 111).

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 typically provided by a real time clock or counter/timer device.

3.5 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.

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:

rtems_address

The data type used to manage addresses. It is equivalent to a void * pointer.

rtems_asr

The return type for an RTEMS ASR.

rtems_asr_entry

The address of the entry point to an RTEMS ASR.

rtems_attribute

The data type used to manage the attributes for RTEMS objects. It is primarily used as an argument to object create routines to specify characteristics of the new object.

rtems_boolean

May only take on the values of TRUE and FALSE. This type is deprecated. Use bool instead.

rtems_context

The CPU dependent data structure used to manage the integer and system register portion of each task's context.

rtems_context_fp

The CPU dependent data structure used to manage the floating point portion of each task's context.

rtems_device_driver

The return type for a RTEMS device driver routine.

rtems_device_driver_entry

The entry point to a RTEMS device driver routine.

rtems_device_major_number

The data type used to manage device major numbers.

rtems_device_minor_number

The data type used to manage device minor numbers.

rtems_double

The RTEMS data type that corresponds to double precision floating point on the tar-

get hardware. This type is deprecated. Use double instead.

rtems_event_set

The data type used to manage and manipulate RTEMS event sets with the Event Manager.

rtems_extension

The return type for RTEMS user extension routines.

rtems_fatal_extension

The entry point for a fatal error user extension handler routine.

rtems_id

The data type used to manage and manipulate RTEMS object IDs.

rtems_interrupt_frame

The data structure that defines the format of the interrupt stack frame as it appears to a user ISR. This data structure may not be defined on all ports.

rtems_interrupt_level

The data structure used with the rtems_interrupt_disable, rtems_interrupt_enable, and rtems_interrupt_flash routines. This data type is CPU dependent and usually corresponds to the contents of the processor register containing the interrupt mask level.

rtems_interval

The data type used to manage and manipulate time intervals. Intervals are non-negative integers used to measure the length of time in clock ticks.

rtems_isr

The return type of a function implementing an RTEMS ISR.

rtems_isr_entry

The address of the entry point to an RTEMS ISR. It is equivalent to the entry point of the function implementing the ISR.

rtems_mp_packet_classes

The enumerated type which specifies the categories of multiprocessing messages. For example, one of the classes is for messages that must be processed by the Task Manager.

rtems_mode

The data type used to manage and dynamically manipulate the execution mode of an RTEMS task.

rtems_mpci_entry

The return type of an RTEMS MPCPI routine.

rtems_mpci_get_packet_entry

The address of the entry point to the get packet routine for an MPCPI implementation.

rtems_mpci_initialization_entry

The address of the entry point to the initialization routine for an MPCPI implementation.

rtems_mpci_receive_packet_entry

The address of the entry point to the receive packet routine for an MPCPI implementation.

rtems_mpci_return_packet_entry

The address of the entry point to the return packet routine for an MPCPI implementation.

rtems_mpci_send_packet_entry

The address of the entry point to the send packet routine for an MPCPI implementation.

rtems_mpci_table

The data structure containing the configuration information for an MPCPI.

rtems_name

The data type used to contain the name of a Classic API object. It is an unsigned thirty-two bit integer which can be treated as a numeric value or initialized using `rtems_build_name` to contain four ASCII characters.

rtems_option

The data type used to specify which behavioral options the caller desires. It is commonly used with potentially blocking directives to specify whether the caller is willing to block or return immediately with an error indicating that the resource was not available.

rtems_packet_prefix

The data structure that defines the first bytes in every packet sent between nodes in an RTEMS multiprocessor system. It contains routing information that is expected to be used by the MPCPI layer.

rtems_signal_set

The data type used to manage and manipulate RTEMS signal sets with the Signal Manager.

int8_t

The C99 data type that corresponds to signed eight bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

int16_t

The C99 data type that corresponds to signed sixteen bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

int32_t

The C99 data type that corresponds to signed thirty-two bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

int64_t

The C99 data type that corresponds to signed sixty-four bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

rtems_single

The RTEMS data type that corresponds to single precision floating point on the target hardware. This type is deprecated. Use `float` instead.

rtems_status_codes

The return type for most RTEMS services. This is an enumerated type of approximately twenty-five values. In general, when a service returns a particular status code, it indicates that a very specific error condition has occurred.

rtems_task

The return type for an RTEMS Task.

rtems_task_argument

The data type for the argument passed to each RTEMS task. In RTEMS 4.7 and older, this is an unsigned thirty-two bit integer. In RTEMS 4.8 and newer, this is based upon the

C99 type `uintptr_t` which is guaranteed to be an integer large enough to hold a pointer on the target architecture.

rtcms_task_begin_extension

The entry point for a task beginning execution user extension handler routine.

rtcms_task_create_extension

The entry point for a task creation execution user extension handler routine.

rtcms_task_delete_extension

The entry point for a task deletion user extension handler routine.

rtcms_task_entry

The address of the entry point to an RTEMS ASR. It is equivalent to the entry point of the function implementing the ASR.

rtcms_task_exitted_extension

The entry point for a task exitted user extension handler routine.

rtcms_task_priority

The data type used to manage and manipulate task priorities.

rtcms_task_restart_extension

The entry point for a task restart user extension handler routine.

rtcms_task_start_extension

The entry point for a task start user extension handler routine.

rtcms_task_switch_extension

The entry point for a task context switch user extension handler routine.

rtcms_tcb

The data structure associated with each task in an RTEMS system.

rtcms_time_of_day

The data structure used to manage and manipulate calendar time in RTEMS.

rtcms_timer_service_routine

The return type for an RTEMS Timer Service Routine.

rtcms_timer_service_routine_entry

The address of the entry point to an RTEMS TSR. It is equivalent to the entry point of the function implementing the TSR.

rtcms_vector_number

The data type used to manage and manipulate interrupt vector numbers.

uint8_t

The C99 data type that corresponds to unsigned eight bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

uint16_t

The C99 data type that corresponds to unsigned sixteen bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

uint32_t

The C99 data type that corresponds to unsigned thirty-two bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

uint64_t

The C99 data type that corresponds to unsigned sixty-four bit integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

uintptr_t

The C99 data type that corresponds to the unsigned integer type that is of sufficient size to represent addresses as unsigned integers. This data type is defined by RTEMS in a manner that ensures it is portable across different target processors.

SCHEDULING CONCEPTS

5.1 Introduction

The concept of scheduling in real-time systems dictates the ability to provide 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.

5.2 Scheduling Algorithms

RTEMS provides a plugin framework which allows it to support multiple scheduling algorithms. RTEMS now includes multiple scheduling algorithms in the SuperCore and the user can select which of these they wish to use in their application. 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 priority based scheduling algorithm. This scheduling algorithm is suitable for single core (e.g. non-SMP) systems and is now known as the *Deterministic Priority Scheduler*. Unless the user configures another scheduling algorithm, RTEMS will use this on single core systems.

5.2.1 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 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.

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

5.2.2 Deterministic Priority Scheduler

This is the scheduler implementation which has always been in RTEMS. After the 4.10 release series, it was factored into 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.

5.2.3 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 which are incapable of supporting a large number of tasks.

This scheduler is only aware of a single core.

5.2.4 Simple SMP Priority Scheduler

This scheduler is based upon the Simple Priority Scheduler and is designed to have the same behaviour on a single core system. But this scheduler is capable of scheduling threads across multiple cores in an SMP system. When given a choice of replacing one of two threads at equal priority on different cores, this algorithm favors replacing threads which are preemptible and have executed the longest.

This algorithm is non-deterministic. When scheduling, it must consider which tasks are to be executed on each core while avoiding superfluous task migrations.

5.2.5 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 which goal is to handle periodic behavior. Period is always equal to 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 a 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 oncoming deadline. Moreover, the `rtems_rate_monotonic_cancel` and `rtems_rate_monotonic_delete` calls clear the deadlines assigned to the task.

5.2.6 Constant Bandwidth Server Scheduling (CBS)

5.2.6 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.

5.3 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.

5.3.1 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. RTEMS supports up to 255 priority levels. Level 255 is the lowest priority and level 1 is the highest.

5.3.2 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.

5.3.3 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.

5.3.4 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 `rtcms_task_wake_after` directive with a time interval 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.

5.4 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.

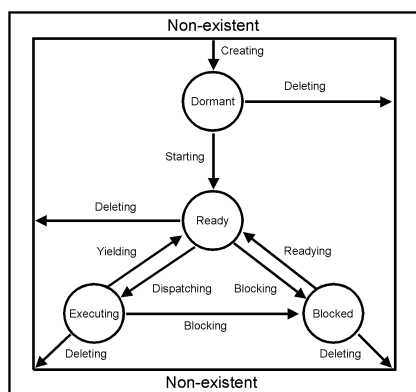
Tasks that utilize a numeric coprocessor and are created with the `RTEMS_FLOATING_POINT` attribute require additional operations during a context switch. These additional operations are necessary to save and restore the floating point context of `RTEMS_FLOATING_POINT` tasks. To avoid unnecessary save 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.

5.5 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 an-

other 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 an `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 time interval. If the time interval 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 which 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.
- A timeout interval expires 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 reenables 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.

INITIALIZATION MANAGER

6.1 Introduction

The Initialization Manager is responsible for initiating and shutting down RTEMS. Initiating RTEMS involves creating and starting all configured initialization tasks, and for invoking the initialization routine for each user-supplied device driver. In a multiprocessor configuration, this manager also initializes the interprocessor communications layer. The directives provided by the Initialization Manager are:

- *rtems_initialize_executive* (page 54) - Initialize RTEMS
- *rtems_shutdown_executive* (page 55) - Shutdown RTEMS

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 System Initialization

System Initialization begins with board reset and continues through RTEMS initialization, initialization of all device drivers, and eventually a context switch to the first user task. Remember, that interrupts are disabled during initialization and the *initialization context* is not a task in any sense and the user should be very careful during initialization.

The BSP must ensure that there is enough stack space reserved for the initialization context to successfully execute the initialization routines for all device drivers and, in multiprocessor configurations, the Multiprocessor Communications Interface Layer initialization routine.

6.2.3 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. This 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.

6.2.4 Initialization Manager Failure

The `rtcms_fatal_error_occurred` directive will be invoked from `rtcms_initialize_executive` for any of the following reasons:

- If either the Configuration Table or the CPU Dependent Information Table is not provided.
- If the starting address of the RTEMS RAM Workspace, supplied by the application in the Configuration Table, is NULL or is not aligned on a four-byte boundary.
- If the size of the RTEMS RAM Workspace is not large enough to initialize and configure the system.
- If the interrupt stack size specified is too small.
- If multiprocessing is configured and the node entry in the Multiprocessor Configuration Table is not between one and the `maximum_nodes` entry.
- If a multiprocessor system is being configured and no Multiprocessor Communications Interface is specified.
- If no user initialization tasks are configured. At least one initialization task must be configured to allow RTEMS to pass control to the application at the end of the executive initialization sequence.

- If any of the user initialization tasks cannot be created or started successfully.

A discussion of RTEMS actions when a fatal error occurs may be found *Chapter 21 Section 3.1 - Announcing a Fatal Error* (page 280).

6.3 Operations

6.3.1 Initializing RTEMS

The `rtms_initialize_executive` Manager directives is called by the `boot_card` routine. The `boot_card` routine 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 `rtms_initialize_executive` directive uses a system initialization linker set to initialize only those parts of the overall RTEMS feature set that is necessary for a particular application. See *Chapter 36 - Linker Sets* (page 521). Each RTEMS feature used the application may optionally register an initialization handler. The system initialization API is available via “`#included <rtms/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_BSP_WORK_AREAS

The work areas consisting of C Program Heap and the RTEMS Workspace are initialized by the Board Support Package. This step is mandatory.

RTEMS_SYSINIT_BSP_START

Basic initialization step provided by the Board Support Package. This step is mandatory.

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_BSP_LIBC

Depending on the application configuration the IO library and root filesystem is initialized. This step is mandatory.

RTEMS_SYSINIT_BEFORE_DRIVERS

This directive performs initialization that must occur between basis RTEMS data structure initialization and device driver initialization. In particular, in a multiprocessor configuration, this directive will create the MPCI Server Task.

RTEMS_SYSINIT_BSP_PRE_DRIVERS

Initialization step performed right before device drivers are initialized provided by the Board Support Package. This step is mandatory.

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_BSP_POST_DRIVERS

Initialization step performed right after device drivers are initialized provided by the Board Support Package. This step is mandatory.

The final action of the `rtms_initialize_executive` directive is to start multitasking. 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 *Chapter 24 - Configuring a System* (page 307).

The final action in the initialization sequence is the initiation of multitasking. When the scheduler and dispatcher are enabled, the highest priority, ready task will be dispatched to run. Control will not be returned to the Board Support Package after multitasking is enabled. The initialization stack may be re-used for interrupt processing.

6.3.2 Shutting Down RTEMS

The `rtems_shutdown_executive` directive is invoked by the application to end multitasking and terminate the system.

6.4 Directives

This section details the Initialization 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.

6.4.1 INITIALIZE_EXECUTIVE - Initialize RTEMS

CALLING SEQUENCE:

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

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

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

NOTES:

This directive should be called by `boot_card` only.

This directive *does not return* to the caller. Errors in the initialization sequence are usually fatal and lead to a system termination.

6.4.2 SHUTDOWN_EXECUTIVE - Shutdown RTEMS

CALLING SEQUENCE:

```
1 void rtems_shutdown_executive(  
2     uint32_t result  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive is called when the application wishes to shutdown RTEMS. The system is terminated with a fatal source of RTEMS_FATAL_SOURCE_EXIT and the specified result code.

NOTES:

This directive *must* be the last RTEMS directive invoked by an application and it *does not return* to the caller.

This directive may be called any time.

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 68) - Create a task
- *rtems_task_ident* (page 70) - Get ID of a task
- *rtems_task_self* (page 71) - Obtain ID of caller
- *rtems_task_start* (page 72) - Start a task
- *rtems_task_restart* (page 73) - Restart a task
- *rtems_task_delete* (page 74) - Delete a task
- *rtems_task_suspend* (page 75) - Suspend a task
- *rtems_task_resume* (page 76) - Resume a task
- *rtems_task_is_suspended* (page 77) - Determine if a task is suspended
- *rtems_task_set_priority* (page 78) - Set task priority
- *rtems_task_mode* (page 79) - Change current task's mode
- *rtems_task_get_note* (page 80) - Get task notepad entry
- *rtems_task_set_note* (page 81) - Set task notepad entry
- *rtems_task_wake_after* (page 82) - Wake up after interval
- *rtems_task_wake_when* (page 83) - Wake up when specified
- *rtems_iterate_over_all_threads* (page 84) - Iterate Over Tasks
- *rtems_task_variable_add* (page 85) - Associate per task variable
- *rtems_task_variable_get* (page 86) - Obtain value of a a per task variable

- *rtems_task_variable_delete* (page 87) - Remove per task variable

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 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.

7.2.4 Task Priority

A task’s priority determines its importance in relation to the other tasks executing on the same processor. RTEMS supports 255 levels of priority ranging from 1 to 255. The data type `rtems_task_priority` is used to store task priorities.

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.

7.2.5 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`.

<code>RTEMS_PREEMPT</code>	enable preemption (default)
<code>RTEMS_NO_PREEMPT</code>	disable preemption
<code>RTEMS_NO_TIMESLICE</code>	disable timeslicing (default)
<code>RTEMS_TIMESLICE</code>	enable timeslicing
<code>RTEMS_ASR</code>	enable ASR processing (default)
<code>RTEMS_NO_ASR</code>	disable ASR processing
<code>RTEMS_INTERRUPT_LEVEL(0)</code>	enable all interrupts (default)
<code>RTEMS_INTERRUPT_LEVEL(n)</code>	execute at interrupt level <code>n</code>

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

7.2.6 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 its argument is:


```

1 rtems_task user_task(
2     rtems_task_argument argument
3 );

```

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

7.2.7 Floating Point Considerations

Creating a task with the `RTEMS_FLOATING_POINT` attribute flag results in additional memory being allocated for the TCB 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. 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 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.

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.

7.2.8 Per Task Variables

Per task variables are deprecated, see the warning below.

Per task variables are used to support global variables whose value may be unique to a task. After indicating that a variable should be treated as private (i.e. per-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.

The value seen by other tasks, including those which have not added the variable to their set and are thus accessing the variable as a common location shared among tasks, cannot be affected by a task once it has added a variable to its local set. Changes made to the variable by other tasks will not affect the value seen by

a task which has added the variable to its private set.

This feature can be used when a routine is to be spawned repeatedly as several independent tasks. Although each task will have its own stack, and thus separate stack variables, they will all share the same static and global variables. To make a variable not shareable (i.e. a “global” variable that is specific to a single task), the tasks can call `rtems_task_variable_add` to make a separate copy of the variable for each task, but all at the same physical address.

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.

A critical point with per-task variables is that each task must separately request that the same global variable is per-task private.

7.2.9 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`.

7.2.10 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 <code>RTEMS_PREEMPT_MASK</code> and enables preemption
RTEMS_NO_PREEMPT	is masked by <code>RTEMS_PREEMPT_MASK</code> and disables preemption
RTEMS_NO_TIMESLICE	is masked by <code>RTEMS_TIMESLICE_MASK</code> and disables timeslicing
RTEMS_TIMESLICE	is masked by <code>RTEMS_TIMESLICE_MASK</code> and enables timeslicing
RTEMS_ASR	is masked by <code>RTEMS_ASR_MASK</code> and enables ASR processing
RTEMS_NO_ASR	is masked by <code>RTEMS_ASR_MASK</code> and disables ASR processing
RTEMS_INTERRUPT_LEVEL(0)	is masked by <code>RTEMS_INTERRUPT_MASK</code> and enables all interrupts
RTEMS_INTERRUPT_LEVEL(n)	is masked by <code>RTEMS_INTERRUPT_MASK</code> 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 interval. The task is blocked until the delay interval has elapsed, at which time the task is unblocked. A task calling the `rtems_task_wake_after` directive with a delay interval 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 Notepad Locations

RTEMS provides sixteen notepad locations for each task. Each notepad location may contain a note consisting of four bytes of information. RTEMS provides two directives, `rtems_task_set_note` and `rtems_task_get_note`, that enable a user to access and change the notepad locations. The `rtems_task_set_note` directive enables the user to set a task's notepad entry to a specified note. The `rtems_task_get_note` directive allows the user to obtain the note contained in

any one of the sixteen notepads of a specified task.

7.3.9 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.

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.

7.3.10 Transition Advice for Obsolete Directives

7.3.10.1 Notepads

Task notepads and the associated directives `rtems_task_get_note` and `rtems_task_set_note` were removed after the 4.11 Release Series. 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 is an option for some use cases.

7.4 Directives

This section details the task 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.

7.4.1 TASK_CREATE - Create a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_create(
2     rtems_name          name,
3     rtems_task_priority initial_priority,
4     size_t              stack_size,
5     rtems_mode          initial_modes,
6     rtems_attribute     attribute_set,
7     rtems_id            *id
8 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task created successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	invalid task name
RTEMS_INVALID_PRIORITY	invalid task priority
RTEMS_MP_NOT_CONFIGURED	multiprocessing not configured
RTEMS_TOO_MANY	too many tasks created
RTEMS_UNSATISFIED	not enough memory for stack/FP context
RTEMS_TOO_MANY	too many global objects

DESCRIPTION:

This directive creates a task which resides on the local node. It allocates and initializes a TCB, a stack, and an optional floating point context area. The mode parameter contains values which sets the task's initial execution mode. The RTEMS_FLOATING_POINT attribute should be specified if the created task is to use a numeric coprocessor. For performance reasons, it is recommended that tasks not using the numeric coprocessor should specify the RTEMS_NO_FLOATING_POINT attribute. If the RTEMS_GLOBAL attribute is specified, the task can be accessed from remote nodes. The task id, returned in id, is used in other task related directives to access the task. When created, a task is placed in the dormant state and can only be made ready to execute using the directive rtems_task_start.

NOTES:

This directive will not cause the calling task to be preempted.

Valid task priorities range from a high of 1 to a low of 255.

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. In addition to being able to specify the task stack size as an integer, there are two constants which may be specified:

RTEMS_MINIMUM_STACK_SIZE

The minimum stack size *RECOMMENDED* for use on this processor. This value is selected by the RTEMS developers 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.

RTEMS_CONFIGURED_MINIMUM_STACK_SIZE

Indicates this task is to be created with a stack size of 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 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.

Application developers should consider the stack usage of the device drivers when calculating the stack size required for tasks which utilize the driver.

The following task attribute constants are defined by RTEMS:

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

The following task mode constants are defined by RTEMS:

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 interrupt level portion of the task execution 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.

Tasks should not be made global unless remote tasks must interact with them. This avoids the system overhead incurred by the creation of a global task. When a global task is created, the task's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including tasks, is limited by the maximum_global_objects field in the Configuration Table.

7.4.2 TASK_IDENT - Get ID of a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_ident(
2     rtems_name  name,
3     uint32_t    node,
4     rtems_id   *id
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	invalid task name
RTEMS_INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the task id associated with the task name specified in name. A task may obtain its own id by specifying RTEMS_SELF or its own task name in name. If the task name is not unique, then the task id returned will match one of the tasks with that name. However, this task id is not guaranteed to correspond to the desired task. The task id, returned in id, is used in other task related directives to access the task.

NOTES:

This directive will not cause the running task to be preempted.

If node is RTEMS_SEARCH_ALL_NODES, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

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.

7.4.3 TASK_SELF - Obtain ID of caller

CALLING SEQUENCE:

```
1 rtems_id rtems_task_self(void);
```

DIRECTIVE STATUS CODES:

Returns the object Id of the calling task.

DESCRIPTION:

This directive returns the Id of the calling task.

NOTES:

If called from an interrupt service routine, this directive will return the Id of the interrupted task.

7.4.4 TASK_START - Start a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_start(
2   rtems_id          id,
3   rtems_task_entry  entry_point,
4   rtems_task_argument argument
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	ask started successfully
RTEMS_INVALID_ADDRESS	invalid task entry point
RTEMS_INVALID_ID	invalid task id
RTEMS_INCORRECT_STATE	task not in the dormant state
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot start remote task

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 starting address of the task is given in `entry_point`. The task's starting argument is contained in `argument`. This argument can be a single value or used as an index into an array of parameter blocks. The type of this numeric argument is an unsigned integer type with 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.

NOTES:

The calling task will be preempted if its preemption mode is enabled and the task being started has a higher priority.

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` directive.

7.4.5 TASK_RESTART - Restart a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_restart(
2   rtems_id      id,
3   rtems_task_argument argument
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task restarted successfully
RTEMS_INVALID_ID	task id invalid
RTEMS_INCORRECT_STATE	task never started
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot restart remote task

DESCRIPTION:

This directive resets the task specified by `id` to begin execution at its original starting address. 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 starting argument is contained in `argument`. This argument can be a single value or an index into an array of parameter blocks. The type of this numeric argument is an unsigned integer type with 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. This new argument may be used to distinguish between the initial `rtems_task_start` of the task and any ensuing calls to `rtems_task_restart` of the task. This can be beneficial in deleting a task. Instead of deleting a task using the `rtems_task_delete` 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.

NOTES:

If `id` is `RTEMS_SELF`, the calling task will be restarted and will not return from this directive.

The calling task will be preempted if its pre-

emption mode is enabled and the task being restarted has a higher priority.

The task must reside on the local node, even if the task was created with the `RTEMS_GLOBAL` option.

7.4.6 TASK_DELETE - Delete a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_delete(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task deleted successfully
RTEMS_INVALID_ID	task id invalid
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot restart remote task

DESCRIPTION:

This directive deletes a task, either the calling task or another task, as specified by id. RTEMS stops the execution of the task and reclaims 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 does not reclaim the following resources: region segments, partition buffers, semaphores, timers, or rate monotonic periods.

NOTES:

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 current task (RTEMS_SELF) will force RTEMS to select another task to execute.

When a global task is deleted, the task id 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 option.

7.4.7 TASK_SUSPEND - Suspend a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_suspend(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task suspended successfully
RTEMS_INVALID_ID	task id invalid
RTEMS_ALREADY_SUSPENDED	task already suspended

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` directive for this task and any blocked state has been removed.

NOTES:

The requesting task can suspend itself 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.

Suspending a global task which does not reside on the local node will generate a request to the remote node to suspend the specified task.

If the task specified by `id` is already suspended, then the `RTEMS_ALREADY_SUSPENDED` status code is returned.

7.4.8 TASK_RESUME - Resume a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_resume(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task resumed successfully
RTEMS_INVALID_ID	task id invalid
RTEMS_INCORRECT_STATE	task not suspended

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.

NOTES:

The running task may be preempted if its preemption mode is enabled and the local task being resumed has a higher priority.

Resuming a global task which does not reside on the local node will generate a request to the remote node to resume the specified task.

If the task specified by `id` is not suspended, then the `RTEMS_INCORRECT_STATE` status code is returned.

7.4.9 TASK_IS_SUSPENDED - Determine if a task is Suspended

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_is_suspended(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task is NOT suspended
RTEMS_ALREADY_SUSPENDED	task is currently suspended
RTEMS_INVALID_ID	task id invalid
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	not supported on remote tasks

DESCRIPTION:

This directive returns a status code indicating whether or not the specified task is currently suspended.

NOTES:

This operation is not currently supported on remote tasks.

7.4.10 TASK_SET_PRIORITY - Set task priority

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_set_priority(
2   rtems_id          id,
3   rtems_task_priority new_priority,
4   rtems_task_priority *old_priority
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task priority set successfully
RTEMS_INVALID_ID	invalid task id
RTEMS_INVALID_ADDRESS	invalid return argument pointer
RTEMS_INVALID_PRIORITY	invalid task priority

DESCRIPTION:

This directive manipulates the priority of the task specified by `id`. An `id` of `RTEMS_SELF` is used to indicate the calling task. 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`. Valid priorities range from a high of 1 to a low of 255.

NOTES:

The calling task may be preempted if its preemption mode is enabled and it lowers its own priority or raises another task's priority.

In case the new priority equals the current priority of the task, then nothing happens.

Setting the priority of a global task which does not reside on the local node will generate a request to the remote node to change the priority of the specified task.

If the task specified by `id` is currently holding any binary semaphores which use the priority inheritance algorithm, then the task's priority cannot be lowered immediately. If the task's priority were lowered immediately, then priority inversion results. The requested lowering of the task's priority will

occur when the task has released all priority inheritance binary semaphores. The task's priority can be increased regardless of the task's use of priority inheritance binary semaphores.

7.4.11 TASK_MODE - Change the current task mode

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_mode(
2     rtems_mode mode_set,
3     rtems_mode mask,
4     rtems_mode *previous_mode_set
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	task mode set successfully
RTEMS_INVALID_ADDRESS	previous_mode_set is NULL

DESCRIPTION:

This directive 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 current 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 parameter.

NOTES:

The calling task will be preempted if it enables preemption and a higher priority task is ready to run.

Enabling timeslicing has no effect if preemption is disabled. For a task to be timesliced, that task must have both preemption and timeslicing enabled.

A task can obtain its current execution mode, without modifying it, by calling this directive with a mask value of RTEMS_CURRENT_MODE.

To temporarily disable the processing of a valid ASR, a task should call this directive with the RTEMS_NO_ASR indicator specified in mode.

The set of task mode constants and each mode's corresponding mask constant 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

7.4.12 TASK_GET_NOTE - Get task notepad entry

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_get_note(
2   rtems_id id,
3   uint32_t notepad,
4   uint32_t *note
5 );

```

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.

7.4.13 TASK_SET_NOTE - Set task notepad entry

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_set_note(
2   rtems_id id,
3   uint32_t notepad,
4   uint32_t note
5 );

```

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.

7.4.14 TASK_WAKE_AFTER - Wake up after interval

CALLING SEQUENCE:

```
1 rtems_status_code rtems_task_wake_after(  
2   rtems_interval ticks  
3 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	always successful
------------------	-------------------

DESCRIPTION:

This directive blocks the calling task for the specified number of system clock ticks. When the requested interval has elapsed, the task is made ready. The clock tick directives automatically updates the delay period.

NOTES:

Setting the system date and time with the `rtems_clock_set` directive has no effect on a `rtems_task_wake_after` blocked task.

A task may give up the processor and remain in the ready state by specifying a value of `RTEMS_YIELD_PROCESSOR` in ticks.

The maximum timer interval that can be specified is the maximum value which can be represented by the `uint32_t` type.

A clock tick is required to support the functionality of this directive.

7.4.15 TASK_WAKE_WHEN - Wake up when specified

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_wake_when(
2   rtems_time_of_day *time_buffer
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	awakened at date/time successfully
RTEMS_INVALID_ADDRESS	time_buffer is NULL
RTEMS_INVALID_TIME_OF_DAY	invalid time buffer
RTEMS_NOT_DEFINED	system date and time is not set

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.

NOTES:

The ticks portion of `time_buffer` structure is ignored. The timing granularity of this directive is a second.

A clock tick is required to support the functionality of this directive.

7.4.16 ITERATE_OVER_ALL_THREADS - Iterate Over Tasks

CALLING SEQUENCE:

```
1 typedef void (*rtems_per_thread_  
  ↪routine)(Thread_Control *the_thread);  
2 void rtems_iterate_over_all_threads(  
3   rtems_per_thread_routine routine  
4 );
```

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 intended for routine use in an operational system.

NOTES:

There is NO protection while this routine is called. Thus it is possible that the_thread could be deleted while this is operating. By not having protection, the user is free to invoke support routines from the C Library which require semaphores for data structures.

7.4.17 TASK_VARIABLE_ADD - Associate per task variable

destructor function could be ‘free’.

Per-task variables are disabled in SMP configurations and this service is not available.

Warning: This directive is deprecated and task variables will be removed.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_variable_add(
2   rtems_id tid,
3   void **task_variable,
4   void (*dtor)(void *)
5 );

```

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

7.4.18 TASK_VARIABLE_GET - Obtain value of a per task variable

Warning: This directive is deprecated and task variables will be removed.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_variable_get(
2   rtems_id tid,
3   void **task_variable,
4   void **task_variable_value
5 );

```

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.

7.4.19 TASK_VARIABLE_DELETE - Remove per task variable

Warning: This directive is deprecated and task variables will be removed.

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_task_variable_
  ↪delete(
2   rtems_id id,
3   void    **task_variable
4 );

```

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.

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 pre-empted upon exit from an ISR. The interrupt manager includes the following directive:

- *rtems_interrupt_catch* (page 96) - Establish an ISR
- *rtems_interrupt_disable* (page 97) - Disable Interrupts
- *rtems_interrupt_enable* (page 98) - Enable Interrupts
- *rtems_interrupt_flash* (page 99) - Flash Interrupt
- *rtems_interrupt_local_disable* (page 100) - Disable Interrupts on Current Processor
- *rtems_interrupt_local_enable* (page 101) - Enable Interrupts on Current Processor
- *rtems_interrupt_lock_initialize* (page 102) - Initialize an ISR Lock
- *rtems_interrupt_lock_acquire* (page 103) - Acquire an ISR Lock
- *rtems_interrupt_lock_release* (page 104) - Release an ISR Lock
- *rtems_interrupt_lock_acquire_isr* (page 105) - Acquire an ISR Lock from ISR
- *rtems_interrupt_lock_release_isr* (page 106) - Release an ISR Lock from ISR
- *rtems_interrupt_is_in_progress* (page 107) - Is an ISR in Progress

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:

```
1 rtems_isr user_isr(
2     rtems_vector_number vector
3 );
```

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`
 - `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`
 - `rtems_timecounter_tick`
 - `rtems_timecounter_simple_downcounter_tick`
 - `rtems_timecounter_simple_upcounter_tick`
- 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_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 interrupt 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.

8.4.1 INTERRUPT_CATCH - Establish an ISR

CALLING SEQUENCE:

```

1 rtems_status_code rtems_interrupt_catch(
2   rtems_isr_entry   new_isr_handler,
3   rtems_vector_number vector,
4   rtems_isr_entry   *old_isr_handler
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	ISR established successfully
RTEMS_ INVALID_ NUMBER	illegal vector number
RTEMS_ INVALID_ ADDRESS	illegal ISR entry point or invalid old_isr_handler

DESCRIPTION:

This directive establishes an interrupt service routine (ISR) for the specified interrupt 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.

NOTES:

This directive will not cause the calling task to be preempted.

8.4.2 INTERRUPT_DISABLE - Disable Interrupts

CALLING SEQUENCE:

```
1 void rtems_interrupt_disable(  
2     rtems_interrupt_level level  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive disables all maskable interrupts and returns the previous level. A later invocation of the `rtems_interrupt_enable` directive should be used to restore the interrupt level.

Macro

This directive is implemented as a macro which modifies the level parameter.

NOTES:

This directive will not cause the calling task to be preempted.

This directive is only available on uniprocessor configurations. The directive `rtems_interrupt_local_disable` is available on all configurations.

8.4.3 INTERRUPT_ENABLE - Enable Interrupts

CALLING SEQUENCE:

```
1 void rtems_interrupt_enable(  
2   rtems_interrupt_level level  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive enables maskable interrupts to the level which was returned by a previous call to `rtems_interrupt_disable`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to `rtems_interrupt_disable` and will be enabled when this directive returns to the caller.

NOTES:

This directive will not cause the calling task to be preempted.

This directive is only available on uniprocessor configurations. The directive `rtems_interrupt_local_enable` is available on all configurations.

8.4.4 INTERRUPT_FLASH - Flash Interrupts

CALLING SEQUENCE:

```
1 void rtems_interrupt_flash(  
2     rtems_interrupt_level level  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive temporarily enables maskable interrupts to the level which was returned by a previous call to `rtems_interrupt_disable`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to `rtems_interrupt_disable` and will be redissabled when this directive returns to the caller.

NOTES:

This directive will not cause the calling task to be preempted.

This directive is only available on uni-processor configurations. The directives `rtems_interrupt_local_disable` and `rtems_interrupt_local_enable` is available on all configurations.

8.4.5 INTERRUPT_LOCAL_DISABLE - Disable Interrupts on Current Processor

CALLING SEQUENCE:

```
1 void rtems_interrupt_local_disable(  
2     rtems_interrupt_level level  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive disables all maskable interrupts and returns the previous level. A later invocation of the `rtems_interrupt_local_enable` directive should be used to restore the interrupt level.

Macro

This directive is implemented as a macro which modifies the `level` parameter.

NOTES:

This directive will not cause the calling task to be preempted.

On SMP configurations this will not ensure system wide mutual exclusion. Use interrupt locks instead.

8.4.6 INTERRUPT_LOCAL_ENABLE - Enable Interrupts on Current Processor

CALLING SEQUENCE:

```
1 void rtems_interrupt_local_enable(  
2     rtems_interrupt_level level  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive enables maskable interrupts to the level which was returned by a previous call to `rtems_interrupt_local_disable`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to `rtems_interrupt_local_disable` and will be enabled when this directive returns to the caller.

NOTES:

This directive will not cause the calling task to be preempted.

8.4.7 INTERRUPT_LOCK_INITIALIZE - Initialize an ISR Lock

CALLING SEQUENCE:

```
1 void rtems_interrupt_lock_initialize(  
2     rtems_interrupt_lock *lock,  
3     const char           *name  
4 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

Initializes an interrupt lock. The name must be persistent throughout the lifetime of the lock.

NOTES:

Concurrent initialization leads to unpredictable results.

8.4.8 INTERRUPT_LOCK_ACQUIRE - Acquire an ISR Lock

CALLING SEQUENCE:

```
1 void rtems_interrupt_lock_acquire(  
2     rtems_interrupt_lock      *lock,  
3     rtems_interrupt_lock_context *lock_  
4     ↪ context  
5 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

Interrupts will be disabled. On SMP configurations this directive acquires a SMP lock.

NOTES:

A separate lock context must be provided for each acquire/release pair, for example an automatic variable.

An attempt to recursively acquire the lock may result in an infinite loop with interrupts disabled.

This directive will not cause the calling thread to be preempted. This directive can be used in thread and interrupt context.

8.4.9 INTERRUPT_LOCK_RELEASE - Release an ISR Lock

CALLING SEQUENCE:

```
1 void rtems_interrupt_lock_release(  
2     rtems_interrupt_lock      *lock,  
3     rtems_interrupt_lock_context *lock_  
4     ↪ context  
5 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

The interrupt status will be restored. On SMP configurations this directive releases a SMP lock.

NOTES:

The lock context must be the one used to acquire the lock, otherwise the result is unpredictable.

This directive will not cause the calling thread to be preempted. This directive can be used in thread and interrupt context.

8.4.10 INTERRUPT_LOCK_ACQUIRE_ISR - Acquire an ISR Lock from ISR

CALLING SEQUENCE:

```
1 void rtems_interrupt_lock_acquire_isr(  
2     rtems_interrupt_lock      *lock,  
3     rtems_interrupt_lock_context *lock_  
4     ↪ context  
5 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

The interrupt status will remain unchanged. On SMP configurations this directive acquires a SMP lock.

NOTES:

A separate lock context must be provided for each acquire/release pair, for example an automatic variable.

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.

8.4.11 INTERRUPT_LOCK_RELEASE_ISR

- Release an ISR Lock from ISR

CALLING SEQUENCE:

```
1 void rtems_interrupt_lock_release_isr(  
2     rtems_interrupt_lock      *lock,  
3     rtems_interrupt_lock_context *lock_  
4     ↪ context  
5 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

The interrupt status will remain unchanged. In SMP configurations, this directive releases an SMP lock.

NOTES:

The lock context must be the one used to acquire the lock, otherwise the result is unpredictable.

This directive is intended for device drivers and should be called from the corresponding interrupt service routine.

8.4.12 INTERRUPT_IS_IN_PROGRESS - Is an ISR in Progress

CALLING SEQUENCE:

```
1 bool rtems_interrupt_is_in_progress(void);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive returns TRUE if the 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.

NOTES:

This directive will not cause the calling task to be preempted.

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 115) - Set date and time
- *rtems_clock_get* (page 116) - Get date and time information
- *rtems_clock_get_tod* (page 117) - Get date and time in TOD format
- *rtems_clock_get_tod_timeval* (page 118) - Get date and time in timeval format
- *rtems_clock_get_seconds_since_epoch* (page 119) - Get seconds since epoch
- *rtems_clock_get_ticks_per_second* (page 120) - Get ticks per second
- *rtems_clock_get_ticks_since_boot* (page 121) - Get current ticks counter value
- *rtems_clock_tick_later* (page 122) - Get tick value in the future
- *rtems_clock_tick_later_usec* (page 123) - Get tick value in the future in microseconds
- *rtems_clock_tick_before* (page 124) - Is tick value is before a point in time
- *rtems_clock_get_uptime* (page 125) - Get time since boot
- *rtems_clock_get_uptime_timeval* (page 126) - Get time since boot in timeval format
- *rtems_clock_get_uptime_seconds* (page 127) - Get seconds since boot
- *rtems_clock_get_uptime_nanoseconds* (page 128) - Get nanoseconds since boot

9.2 Background

9.2.1 Required Support

For the features provided by the clock manager to be utilized, periodic timer interrupts are required. Therefore, a real-time clock or hardware timer is necessary to create the timer interrupts. The clock tick directive is normally called by the timer ISR to announce to RTEMS that a system clock tick has occurred. Elapsed time is measured in ticks. A tick is defined to be an integral number of microseconds which is specified by the user in the Configuration Table.

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:

```

1 struct rtems_tod_control {
2     uint32_t year; /* greater than 1987 */
3     uint32_t month; /* 1 - 12 */
4     uint32_t day; /* 1 - 31 */
5     uint32_t hour; /* 0 - 23 */
6     uint32_t minute; /* 0 - 59 */
7     uint32_t second; /* 0 - 59 */
8     uint32_t ticks; /* elapsed between ↵
↵seconds */
9 };
10 typedef struct rtems_tod_control 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` directive. Some applications expect to operate on a *UNIX-style* date and time data structure. The `rtems_clock_get_tod_timeval` always returns the date and time in `struct timeval` format. The `rtems_clock_get` directive can optionally return the current date and time in this format.

The `struct timeval` data structure has two fields: `tv_sec` and `tv_usec` which are seconds and microseconds, respectively. The `tv_sec` field in this data structure is the number of seconds since the POSIX epoch of *January 1, 1970*

but will never be prior to the RTEMS epoch of *January 1, 1988*.

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 in the Configuration Table. 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` and `rtems_task_wake_when` directives and, as a result, does not have an RTEMS ID. 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_message_queue_receive`, `rtems_event_receive`, `rtems_semaphore_obtain` and `rtems_region_get_segment` 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

The `rtems_clock_get` directive allows 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 information returned by the `rtems_clock_get` directive is dependent on the option selected by the

caller. This is specified using one of the following constants associated with the enumerated type `rtems_clock_get_options`:

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.

9.4 Directives

This section details the clock 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.

9.4.1 CLOCK_SET - Set date and time

CALLING SEQUENCE:

```

1 rtems_status_code rtems_clock_set(
2     rtems_time_of_day *time_buffer
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	date and time set successfully
RTEMS_INVALID_ADDRESS	time_buffer is NULL
RTEMS_INVALID_CLOCK	invalid time of day

DESCRIPTION:

This directive sets the system date and time. The date, time, and ticks in the `time_buffer` structure are all range-checked, and an error is returned if any one is out of its valid range.

NOTES:

Years before 1988 are invalid.

The system date and time are based on the configured tick rate (number of microseconds in a tick).

Setting the time forward may cause a higher priority task, blocked waiting on a specific time, to be made ready. In this case, the calling task will be preempted after the next clock tick.

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.

9.4.2 CLOCK_GET - Get date and time information

Warning: This directive is deprecated and will be removed.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_clock_get(
2   rtems_clock_get_options option,
3   void *time_buffer
4 );

```

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	(<code>rtems_time_of_day *</code>)
RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH	(<code>rtems_interval *</code>)
RTEMS_CLOCK_GET_TICKS_SINCE_BOOT	(<code>rtems_interval *</code>)
RTEMS_CLOCK_GET_TICKS_PER_SECOND	(<code>rtems_interval *</code>)
RTEMS_CLOCK_GET_TIME_VALUE	(<code>struct timeval *</code>)

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.

9.4.3 CLOCK_GET_TOD - Get date and time in TOD format

CALLING SEQUENCE:

```

1 rtems_status_code rtems_clock_get_tod(
2   rtems_time_of_day *time_buffer
3 );

```

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 date and time has not been set with a previous call to `rtems_clock_set`, then the `RTEMS_NOT_DEFINED` status code is returned.

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.

9.4.4 CLOCK_GET_TOD_TIMEVAL - Get date and time in timeval format

CALLING SEQUENCE:

```
1 rtems_status_code    rtems_clock_get_tod_  
  ↪ interval(  
2     struct timeval  *time  
3 );
```

DIRECTIVE STATUS CODES:

DESCRIPTION:

This directive obtains the system date and time in POSIX struct timeval format. If 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.

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.

9.4.5 CLOCK_GET_SECONDS_SINCE_EPOCH

- Get seconds since epoch

CALLING SEQUENCE:

```
1 rtems_status_code rtems_clock_get_seconds_  
  ↪since_epoch(  
2     rtems_interval *the_interval  
3 );
```

DIRECTIVE STATUS CODES:

DESCRIPTION:

This directive returns the number of seconds since the RTEMS epoch and the current system date and time. If 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.

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.

9.4.6 CLOCK_GET_TICKS_PER_SECOND

- Get ticks per second

CALLING SEQUENCE:

```
1 rtems_interval rtems_clock_get_ticks_per_  
   ↪second(void);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive returns the number of clock ticks per second. This is strictly based upon the microseconds per clock tick that the application has configured.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted.

9.4.7 CLOCK_GET_TICKS_SINCE_BOOT - Get current ticks counter value

CALLING SEQUENCE:

```
1 rtems_interval    rtems_clock_get_ticks_  
   ↪since_boot(void);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive returns the current tick counter value. 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` is another and potentially more accurate way of obtaining similar information.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted.

9.4.8 CLOCK_TICK_LATER - Get tick value in the future

CALLING SEQUENCE:

```
1 rtems_interval rtems_clock_tick_later(  
2   rtems_interval delta  
3 );
```

DESCRIPTION:

Returns the ticks counter value delta ticks in the future.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted.

9.4.9 CLOCK_TICK_LATER_USEC - Get tick value in the future in microseconds

CALLING SEQUENCE:

```
1 rtems_interval    rtems_clock_tick_later_  
   ↪usec(  
2   rtems_interval delta_in_usec  
3 );
```

DESCRIPTION:

Returns the ticks counter value at least delta microseconds in the future.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted.

9.4.10 CLOCK_TICK_BEFORE - Is tick value is before a point in time

CALLING SEQUENCE:

```
1 rtems_interval rtems_clock_tick_before(  
2   rtems_interval tick  
3 );
```

DESCRIPTION:

Returns true if the current ticks counter value indicates a time before the time specified by the tick value and false otherwise.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted.

EXAMPLE:

```
1 status busy( void )  
2 {  
3     rtems_interval timeout = rtems_clock_  
4     ↪tick_later_usec( 10000 );  
5     do {  
6         if ( ok() ) {  
7             return success;  
8         }  
9     } while ( rtems_clock_tick_before( ↪  
10    ↪timeout ) );  
11     return timeout;  
12 }
```


9.4.11 CLOCK_GET_UPTIME - Get the time since boot

CALLING SEQUENCE:

```
1 rtems_status_code rtems_clock_get_uptime(  
2     struct timespec *uptime  
3 );
```

DIRECTIVE STATUS CODES:

DESCRIPTION:

This directive returns the seconds and nanoseconds since the system was booted. If the BSP supports nanosecond clock accuracy, the time reported will probably be different on every call.

NOTES:

This directive may be called from an ISR.

9.4.12 CLOCK_GET_UPTIME_TIMEVAL - Get the time since boot in timeval format

CALLING SEQUENCE:

```
1 void rtems_clock_get_uptime_timeval(  
2     struct timeval *uptime  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive returns the seconds and microseconds since the system was booted. If the BSP supports nanosecond clock accuracy, the time reported will probably be different on every call.

NOTES:

This directive may be called from an ISR.

9.4.13 CLOCK_GET_UPTIME_SECONDS - Get the seconds since boot

CALLING SEQUENCE:

```
1 time_t rtems_clock_get_uptime_  
↔seconds(void);
```

DIRECTIVE STATUS CODES:

The system uptime in seconds.

DESCRIPTION:

This directive returns the seconds since the system was booted.

NOTES:

This directive may be called from an ISR.

9.4.14 CLOCK_GET_UPTIME_NANOSECONDS

- Get the nanoseconds since boot

CALLING SEQUENCE:

```
1 uint64_t      rtems_clock_get_uptime_  
  ↪nanoseconds(void);
```

DIRECTIVE STATUS CODES:

The system uptime in nanoseconds.

DESCRIPTION:

This directive returns the nanoseconds since the system was booted.

NOTES:

This directive may be called from an ISR.

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 134) - Create a timer
- *rtems_timer_ident* (page 135) - Get ID of a timer
- *rtems_timer_cancel* (page 136) - Cancel a timer
- *rtems_timer_delete* (page 137) - Delete a timer
- *rtems_timer_fire_after* (page 138) - Fire timer after interval
- *rtems_timer_fire_when* (page 139) - Fire timer when specified
- *rtems_timer_initiate_server* (page 140) - Initiate server for task-based timers
- *rtems_timer_server_fire_after* (page 141) - Fire task-based timer after interval
- *rtems_timer_server_fire_when* (page 142) - Fire task-based timer when specified
- *rtems_timer_reset* (page 143) - Reset an interval 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 higher than any RTEMS application task, and is created non-preemptible, and thus can be viewed logically as the lowest priority interrupt.

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. However, since the Timer Server task is non-preemptible, only directives allowed from an ISR can be called in the timer service routine.

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:

```

1 rtems_timer_service_routine user_routine(
2     rtems_id timer_id,
3     void *user_data
4 );

```

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 timer 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.

10.4.1 TIMER_CREATE - Create a timer

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_create(
2     rtems_name name,
3     rtems_id *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer created successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	invalid timer name
RTEMS_TOO_MANY	too many timers created

DESCRIPTION:

This directive creates a timer. The assigned timer id is returned in id. This id is used to access the timer with other timer manager directives. For control and maintenance of the timer, RTEMS allocates a TMCB from the local TMCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

10.4.2 TIMER_IDENT - Get ID of a timer

CALLING SEQUENCE:

```
1 rtems_status_code rtems_timer_ident(  
2     rtems_name name,  
3     rtems_id *id  
4 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	timer name not found

DESCRIPTION:

This directive obtains the timer id associated with the timer name to be acquired. If the timer name is not unique, then the timer id will match one of the timers with that name. However, this timer id is not guaranteed to correspond to the desired timer. The timer id is used to access this timer in other timer related directives.

NOTES:

This directive will not cause the running task to be preempted.

10.4.3 TIMER_CANCEL - Cancel a timer

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_cancel(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer canceled successfully
RTEMS_INVALID_ID	invalid timer id

DESCRIPTION:

This directive cancels the timer id. This timer will be reinitiated by the next invocation of `rtems_timer_reset`, `rtems_timer_fire_after`, or `rtems_timer_fire_when` with this id.

NOTES:

This directive will not cause the running task to be preempted.

10.4.4 TIMER_DELETE - Delete a timer

CALLING SEQUENCE:

```
1 rtems_status_code rtems_timer_delete(  
2     rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

RTEMS_	timer deleted
SUCCESSFUL	successfully
RTEMS_INVALID_	invalid timer id
ID	

DESCRIPTION:

This directive deletes the timer specified by `id`. If the timer is running, it is automatically canceled. The TMCB for the deleted timer is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A timer can be deleted by a task other than the task which created the timer.

10.4.5 TIMER_FIRE_AFTER - Fire timer after interval

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_fire_after(
2     rtems_id          id,
3     rtems_interval    ↵
4     ↵ticks,
5     rtems_timer_service_routine_entry ↵
6     ↵routine,
7     void              ↵
8     ↵*user_data
9 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer initiated successfully
RTEMS_INVALID_ADDRESS	routine is NULL
RTEMS_INVALID_ID	invalid timer id
RTEMS_INVALID_NUMBER	invalid interval

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 ticks clock ticks has passed. When the timer fires, the timer service routine routine will be invoked with the argument `user_data`.

NOTES:

This directive will not cause the running task to be preempted.

10.4.6 TIMER_FIRE_WHEN - Fire timer when specified

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_fire_when(
2     rtems_id id,
3     rtems_time_of_day wall_time,
4     rtems_timer_service_routine_entry routine,
5     void user_data
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer initiated successfully
RTEMS_INVALID_ADDRESS	routine is NULL
RTEMS_INVALID_ADDRESS	wall_time is NULL
RTEMS_INVALID_ID	invalid timer id
RTEMS_NOT_DEFINED	system date and time is not set
RTEMS_INVALID_CLOCK	invalid time of day

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`.

NOTES:

This directive will not cause the running task to be preempted.

10.4.7 TIMER_INITIATE_SERVER - Initiate server for task-based timers

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_timer_initiate_
  ↪server(
2   uint32_t          priority,
3   uint32_t          stack_size,
4   rtems_attribute  attribute_set
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_	Timer Server initiated
SUCCESSFUL	successfully
RTEMS_TOO_	too many tasks created
MANY	

DESCRIPTION:

This directive initiates the Timer Server task. This task is responsible for executing all timers initiated via the `rtems_timer_server_fire_after` or `rtems_timer_server_fire_when` directives.

NOTES:

This directive could cause the calling task to be preempted.

The Timer Server task is created using the `rtems_task_create` service and must be accounted for when configuring the system.

Even through this directive invokes the `rtems_task_create` and `rtems_task_start` directives, it should only fail due to resource allocation problems.

10.4.8 TIMER_SERVER_FIRE_AFTER - Fire task-based timer after interval

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_server_fire_
  ↳after(
2   rtems_id          id,
3   rtems_interval   ↳
  ↳ticks,
4   rtems_timer_service_routine_entry ↳
  ↳routine,
5   void             ↳
  ↳*user_data
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer initiated successfully
RTEMS_INVALID_ADDRESS	routine is NULL
RTEMS_INVALID_ID	invalid timer id
RTEMS_INVALID_NUMBER	invalid interval
RTEMS_INCORRECT_STATE	Timer Server not initiated

DESCRIPTION:

This directive initiates the timer specified by `id` and specifies that when it fires it will be executed by the Timer Server.

If the timer is running, it is automatically canceled before being initiated. The timer is scheduled to fire after an interval ticks clock ticks has passed. When the timer fires, the timer service routine routine will be invoked with the argument `user_data`.

NOTES:

This directive will not cause the running task to be preempted.

10.4.9 TIMER_SERVER_FIRE_WHEN

- Fire task-based timer when specified

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_server_fire_
  ↳when(
2   rtems_id          id,
3   rtems_time_of_day
  ↳*wall_time,
4   rtems_timer_service_routine_entry
  ↳routine,
5   void
  ↳*user_data
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer initiated successfully
RTEMS_INVALID_ADDRESS	routine is NULL
RTEMS_INVALID_ADDRESS	wall_time is NULL
RTEMS_INVALID_ID	invalid timer id
RTEMS_NOT_DEFINED	system date and time is not set
RTEMS_INVALID_CLOCK	invalid time of day
RTEMS_INCORRECT_STATE	Timer Server not initiated

DESCRIPTION:

This directive initiates the timer specified by `id` and specifies that when it fires it will be executed by the Timer Server.

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`.

NOTES:

This directive will not cause the running task to be preempted.

10.4.10 TIMER_RESET - Reset an interval timer

CALLING SEQUENCE:

```

1 rtems_status_code rtems_timer_reset(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	timer reset successfully
RTEMS_INVALID_ID	invalid timer id
RTEMS_NOT_DEFINED	attempted to reset a when or newly created timer

DESCRIPTION:

This directive resets the timer associated with `id`. This timer must have been previously initiated with either the `rtems_timer_fire_after` or `rtems_timer_server_fire_after` 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` or `rtems_timer_server_fire_after` directive used.

NOTES:

If the timer has not been used or the last usage of this timer was by a `rtems_timer_fire_when` or `rtems_timer_server_fire_when` 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.

This directive will not cause the running task to be preempted.

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 156)
- Create a rate monotonic period
- *rtems_rate_monotonic_ident* (page 157) -
Get ID of a period
- *rtems_rate_monotonic_cancel* (page 158)
- Cancel a period
- *rtems_rate_monotonic_delete* (page 159)
- Delete a rate monotonic period
- *rtems_rate_monotonic_period* (page 160)
- Conclude current/Start next period
- *rtems_rate_monotonic_get_status*
(page 161) - Obtain status from a period
- *rtems_rate_monotonic_get_statistics*
(page 162) - Obtain statistics from a period
- *rtems_rate_monotonic_reset_statistics*
(page 163) - Reset statistics for a period
- *rtems_rate_monotonic_reset_all_statistics*
(page 164) - Reset statistics for all periods
- *rtems_rate_monotonic_report_statistics*
(page 165) - Print period statistics report

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 implicitly 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 Rate Monotonic Manager 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... index:: aperiodic task, definition

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 guarantee that a set of tasks is schedulable. 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 static 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 static priority algorithm, then RMS will be able to schedule that task set. RMS bases its 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, 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 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 $\text{Time}(\text{index}) / \text{Period}(\text{index})$. The processor utilization can be calculated as follows:

```

1 Utilization = 0
2 for index = 1 to maximum_tasks
3   Utilization = Utilization + (Time(index)/
  ↪ Period(index))

```

To ensure schedulability even under transient overload, the processor utilization must adhere to the following rule:

```

Utilization = maximum_tasks * (2**(1/maximum_
  ↪ tasks) - 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.

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, exe-

cution time, and processor utilization for each task:

Task	RMS Priority	Pe-riod	Execu-tion 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.

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	Pe-riod	Execu-tion 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:

Dead-line Time	Task 1	Task 2	Task 3	Total Execution Time	All Dead-lines 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.

11.2.5.7 Further Reading

For more information on Rate Monotonic Scheduling and its schedulability analysis, the reader is referred to the following:

- C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment." *Journal of the Association of Computing Machinery*. January 1973. pp. 46-61.
- John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." *IEEE Real-Time Systems Symposium*. 1989. pp. 166-171.
- Lui Sha and John Goodenough. "Real-Time Scheduling theory and Ada." *IEEE Computer*. April 1990. pp. 53-62.
- Alan Burns. "Scheduling hard real-time systems: a review." *Software Engineering Journal*. May 1991. pp. 116-128.

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 period will be initiated with a length of period ticks and the calling task returns immediately with a timeout error status.

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.

```

1 rtems_task      Periodic_task(rtems_task_
  ↪argument arg)
2 {
3     rtems_name    name;
4     rtems_id      period;
5     rtems_status_code status;
6     name = rtems_build_name( 'P', 'E', 'R',
  ↪'D' );
7     status = rtems_rate_monotonic_create(
  ↪name, &period );
8     if ( status != RTEMS_STATUS_SUCCESSFUL
  ↪) {
9         printf( "rtems_monotonic_create
  ↪failed with status of %d.\n", rc );
10        exit( 1 );
11    }
12    while ( 1 ) {
13        if ( rtems_rate_monotonic_period(
  ↪period, 100 ) == RTEMS_TIMEOUT )
14            break;
15        /* Perform some periodic actions */
16    }
17    /* missed period so delete period and
  ↪SELF */
18    status = rtems_rate_monotonic_delete(
  ↪period );
19    if ( status != RTEMS_STATUS_SUCCESSFUL
  ↪) {
20        printf( "rtems_rate_monotonic_delete
  ↪failed with status of %d.\n", status );
21        exit( 1 );
22    }
23    status = rtems_task_delete( SELF );
  ↪/* should not return */
24    printf( "rtems_task_delete returned with
  ↪status of %d.\n", status );
25    exit( 1 );
26 }

```

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.

```

1 rtems_task      Periodic_task(rtems_task_
  ↪argument arg)
2 {
3     rtems_name    name_1, name_2;
4     rtems_id      period_1, period_2;
5     rtems_status_code status;
6     name_1 = rtems_build_name( 'P', 'E', 'R',
  ↪', '1' );
7     name_2 = rtems_build_name( 'P', 'E', 'R',
  ↪', '2' );
8     (void) rtems_rate_monotonic_create(
  ↪name_1, &period_1 );
9     (void) rtems_rate_monotonic_create(
  ↪name_2, &period_2 );
10    while ( 1 ) {
11        if ( rtems_rate_monotonic_period(
  ↪period_1, 100 ) == TIMEOUT )
12            break;
13        if ( rtems_rate_monotonic_period(
  ↪period_2, 40 ) == TIMEOUT )
14            break;
15        /*
16         * Perform first set of actions
  ↪between clock
17         * ticks 0 and 39 of every 100
  ↪ticks.
18         */
19        if ( rtems_rate_monotonic_period(
  ↪period_2, 30 ) == TIMEOUT )
20            break;
21        /*
22         * Perform second set of actions
  ↪between clock 40 and 69
23         * of every 100 ticks. THEN ...
24         *
25         * Check to make sure we didn't
  ↪miss the period_2 period.
26         */
27        if ( rtems_rate_monotonic_period(
  ↪period_2, STATUS ) == TIMEOUT )
28            break;
29        (void) rtems_rate_monotonic_cancel(
  ↪period_2 );
30    }
31    /* missed period so delete period and
  ↪SELF */

```

```
32     (void ) rtems_rate_monotonic_delete( ↵  
↵period_1 );  
33     (void ) rtems_rate_monotonic_delete( ↵  
↵period_2 );  
34     (void ) task_delete( SELF );  
35 }
```

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 executed, 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 rate monotonic 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.

11.4.1 RATE_MONOTONIC_CREATE - Create a rate monotonic period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↪create(
2   rtems_name  name,
3   rtems_id   *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	rate monotonic period created successfully
RTEMS_ INVALID_ NAME	invalid period name
RTEMS_TOO_ MANY	too many periods created

DESCRIPTION:

This directive creates a rate monotonic period. The assigned rate monotonic id is returned in id. This id is used to access the period with other rate monotonic manager directives. For control and maintenance of the rate monotonic period, RTEMS allocates a PCB from the local PCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

11.4.2 RATE_MONOTONIC_IDENT - Get ID of a period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↪ident(
2   rtems_name name,
3   rtems_id  *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_	period identified
SUCCESSFUL	successfully
RTEMS_INVALID_	period name not
NAME	found

DESCRIPTION:

This directive obtains the period id associated with the period name to be acquired. If the period name is not unique, then the period id will match one of the periods with that name. However, this period id is not guaranteed to correspond to the desired period. The period id is used to access this period in other rate monotonic manager directives.

NOTES:

This directive will not cause the running task to be preempted.

11.4.3 RATE_MONOTONIC_CANCEL - Cancel a period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↪cancel(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	period canceled successfully
RTEMS_ INVALID_ID	invalid rate monotonic period id
RTEMS_NOT_ OWNER_OF_ RESOURCE	rate monotonic period not created by calling task

DESCRIPTION:

This directive cancels the rate monotonic period id. This period will be reinitiated by the next invocation of `rtems_rate_monotonic_period` with id.

NOTES:

This directive will not cause the running task to be preempted.

The rate monotonic period specified by id must have been created by the calling task.

11.4.4 RATE_MONOTONIC_DELETE - Delete a rate monotonic period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↪delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_	period deleted
SUCCESSFUL	successfully
RTEMS_	invalid rate monotonic
INVALID_ID	period id

DESCRIPTION:

This directive deletes the rate monotonic period specified by id. If the period is running, it is automatically canceled. The PCB for the deleted period is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A rate monotonic period can be deleted by a task other than the task which created the period.

11.4.5 RATE_MONOTONIC_PERIOD

- Conclude current/Start next period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↳period(
2   rtems_id        id,
3   rtems_interval length
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	period initiated successfully
RTEMS_ INVALID_ ID	invalid rate monotonic period id
RTEMS_ NOT_ OWNER_OF_ RESOURCE	period not created by calling task
RTEMS_ NOT_ DEFINED	period has never been initiated (only possible when period is set to PERIOD_STATUS)
RTEMS_ TIMEOUT	period has expired

DESCRIPTION:

This directive initiates the rate monotonic period id with a length of period ticks. If id is running, then the calling task will block for the remainder of the period before reinitiating the period with the specified period. If id was not running (either expired or never initiated), the period is immediately initiated and the directive returns immediately.

If invoked with a period of RTEMS_PERIOD_STATUS ticks, the current state of id will be returned. The directive status indicates the current state of the period. This does not alter the state or period of the period.

NOTES:

This directive will not cause the running task to be preempted.

11.4.6 RATE_MONOTONIC_GET_STATUS - Obtain status from a period

of the `rtems_rate_monotonic_period` directive.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_rate_monotonic_
  ↪get_status(
2   rtems_id id,
3   rtems_rate_monotonic_period_status ↪
  ↪*status
4 );
```

NOTES:

This directive will not cause the running task to be preempted.

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	period initiated successfully
RTEMS_INVALID_ID	invalid rate monotonic period id
RTEMS_INVALID_ADDRESS	invalid address of status

*DESCRIPTION:

This directive returns status information associated with the rate monotonic period id in the following data structure:

```

1 typedef struct {
2   rtems_id ↪
  ↪owner;
3   rtems_rate_monotonic_period_states ↪
  ↪state;
4   rtems_rate_monotonic_period_time_t ↪
  ↪since_last_period;
5   rtems_thread_cpu_usage_t ↪
  ↪executed_since_last_period;
6 } rtems_rate_monotonic_period_status;
```

A configure time option can be used to select whether the time information is given in ticks or seconds and nanoseconds. The default is seconds and nanoseconds. If the period's state is `RATE_MONOTONIC_INACTIVE`, both time values will be set to 0. Otherwise, both time values will contain time information since the last invocation of the `rtems_rate_monotonic_period` directive. More specifically, the `ticks_since_last_period` value contains the elapsed time which has occurred since the last invocation of the `rtems_rate_monotonic_period` directive and the `ticks_executed_since_last_period` contains how much processor time the owning task has consumed since the invocation

11.4.7 RATE_MONOTONIC_GET_STATISTICS NOTES:

- Obtain statistics from a period

This directive will not cause the running task to be preempted.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_rate_monotonic_
  ↳get_statistics(
2   rtems_id
  ↳ id,
3   rtems_rate_monotonic_period_statistics_
  ↳*statistics
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_	period initiated
SUCCESSFUL	successfully
RTEMS_INVALID_	invalid rate
ID	monotonic period id
RTEMS_INVALID_	invalid address of
ADDRESS	statistics

DESCRIPTION:

This directive returns statistics information associated with the rate monotonic period id in the following data structure:

```

1 typedef struct {
2   uint32_t count;
3   uint32_t missed_count;
4   #ifdef RTEMS_ENABLE_NANOSECOND_CPU_
  ↳USAGE_STATISTICS
5   struct timespec min_cpu_time;
6   struct timespec max_cpu_time;
7   struct timespec total_cpu_time;
8   #else
9   uint32_t min_cpu_time;
10  uint32_t max_cpu_time;
11  uint32_t total_cpu_time;
12  #endif
13  #ifdef RTEMS_ENABLE_NANOSECOND_RATE_
  ↳MONOTONIC_STATISTICS
14  struct timespec min_wall_time;
15  struct timespec max_wall_time;
16  struct timespec total_wall_time;
17  #else
18  uint32_t min_wall_time;
19  uint32_t max_wall_time;
20  uint32_t total_wall_time;
21  #endif
22 } rtems_rate_monotonic_period_statistics;

```

This directive returns the current statistics information for the period instance associated with id. The information returned is indicated by the structure above.

11.4.8 RATE_MONOTONIC_RESET_STATISTICS

- Reset statistics for a period

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_rate_monotonic_
  ↪reset_statistics(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	period initiated successfully
RTEMS_ INVALID_ID	invalid rate monotonic period id

DESCRIPTION:

This directive resets the statistics information associated with this rate monotonic period instance.

NOTES:

This directive will not cause the running task to be preempted.

11.4.9 RATE_MONOTONIC_RESET_ALL_STATISTICS

- Reset statistics for all periods

CALLING SEQUENCE:

```
1 void rtems_rate_monotonic_reset_all_
   ↪statistics(void);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive resets the statistics information associated with all rate monotonic period instances.

NOTES:

This directive will not cause the running task to be preempted.

11.4.10 RATE_MONOTONIC_REPORT_STATISTICS

- Print period statistics report

CALLING SEQUENCE:

```
1 void          rtems_rate_monotonic_report_
   ↪statistics(void);
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive prints a report on all active periods which have executed at least one period. The following is an example of the output generated by this directive.

```
1 ID          OWNER    PERIODS  MISSED  CPU_
   ↪TIME      WALL TIME
2 MIN/MAX/AVG MIN/MAX/AVG
3 0x42010001  TA1         502     0     0/1/
   ↪0.99     0/0/0.00
4 0x42010002  TA2         502     0     0/1/
   ↪0.99     0/0/0.00
5 0x42010003  TA3         501     0     0/1/
   ↪0.99     0/0/0.00
6 0x42010004  TA4         501     0     0/1/
   ↪0.99     0/0/0.00
7 0x42010005  TA5          10     0     0/1/
   ↪0.90     0/0/0.00
```

NOTES:

This directive will not cause the running task to be preempted.

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 176) - Create a semaphore
- *rtems_semaphore_ident* (page 178) - Get ID of a semaphore
- *rtems_semaphore_delete* (page 179) - Delete a semaphore
- *rtems_semaphore_obtain* (page 180) - Acquire a semaphore
- *rtems_semaphore_release* (page 181) - Release a semaphore
- *rtems_semaphore_flush* (page 182) - Unblock all tasks waiting on a semaphore
- *rtems_semaphore_set_priority* (page 183) - Set priority by scheduler for a 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 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. 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 must issue the `rtems_semaphore_release` directive to allow another task to execute the critical section.

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 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 task requests access to shared resource which is currently allocated to 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.

12.2.3 Priority Inheritance

Priority inheritance is 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. Each time a task blocks attempting to obtain the resource, the task holding the resource may have its priority increased.

On SMP configurations, in case the task holding the resource and the task that blocks attempting to obtain the resource are in different scheduler instances, the priority of the holder is raised to the pseudo-interrupt priority (priority boosting). The pseudo-interrupt priority is the highest priority.

RTEMS supports priority inheritance for local, binary semaphores that use the priority task wait queue blocking discipline. When a task of higher priority than the task holding the semaphore blocks, the priority of the task holding the semaphore is increased to that of the blocking task. When the task holding the task completely releases the binary semaphore (i.e. not for a nested release), the holder's priority is restored to the value it had before any higher priority was inherited.

The RTEMS implementation of the priority inheritance algorithm takes into account the scenario in which a task holds more than one binary semaphore. The holding task will execute at the priority of the higher of the highest ceiling priority or at the priority of the highest priority task blocked waiting for any of the semaphores the task holds. Only when the task releases ALL of the binary semaphores it holds will its priority be restored to the normal value.

12.2.4 Priority Ceiling

Priority ceiling is an algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task which will EVER block waiting for that resource. This algorithm addresses the problem of priority inversion although it avoids the possibility of changing the priority of the task holding the resource multiple times. The priority ceiling algorithm will only change the priority of the task holding the resource a maximum of one time. The ceiling priority is set at creation time and must be the priority of the highest priority task which will ever attempt to acquire that semaphore.

RTEMS supports priority ceiling for local, binary semaphores that use the priority task wait queue blocking discipline. When a task of lower priority than the ceiling priority successfully obtains the semaphore, its priority is raised to the ceiling priority. When the task holding the task completely releases the binary semaphore (i.e. not for a nested release), the holder's priority is restored to the value it had before any higher priority was put into effect.

The need to identify the highest priority task which will attempt to obtain a particular semaphore can be a difficult task in a large, complicated system. Although the priority ceiling algorithm is more efficient than the priority inheritance algorithm with respect to the maximum number of task priority changes which may occur while a task holds a particular semaphore, the priority inheritance algorithm is more forgiving in that it does not require this apriori information.

The RTEMS implementation of the priority ceiling algorithm takes into account the scenario in which a task holds more than one binary semaphore. The holding task will execute at the priority of the higher of the highest ceiling priority or at the priority of the highest priority task blocked waiting for any of the semaphores the task holds. Only when the task releases ALL of the binary semaphores it holds will its priority be restored to the normal value.

12.2.5 Multiprocessor Resource Sharing Protocol

The Multiprocessor Resource Sharing Protocol (MrsP) is defined in *A. Burns and A.J. Wellings, A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP, Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013), July 2013*. It is a generalization of the Priority Ceiling Protocol to SMP systems. Each MrsP semaphore uses a ceiling priority per scheduler instance. These ceiling priorities can be specified with `rtems_semaphore_set_priority()`. A task obtaining or owning a MrsP semaphore will execute with the ceiling priority for its scheduler instance as specified by the MrsP semaphore object. Tasks waiting to get ownership of a

MrsP semaphore will not relinquish the processor voluntarily. In case the owner of a MrsP semaphore gets preempted it can ask all tasks waiting for this semaphore to help out and temporarily borrow the right to execute on one of their assigned processors.

12.2.6 Building a Semaphore Attribute Set

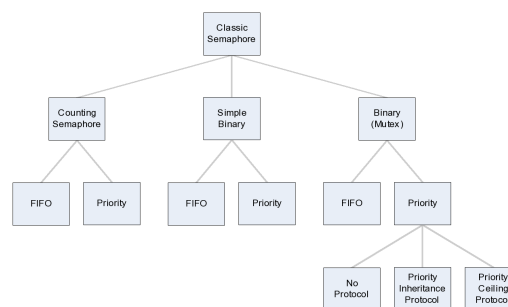
In general, an attribute set is built by a bitwise OR of the desired attribute components. The following table lists the set of valid semaphore attributes:

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 combinations 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.



12.2.7 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 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 semaphore 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.

12.4.1 SEMAPHORE_CREATE - Create a semaphore

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_create(
2     rtems_name      name,
3     uint32_t        count,
4     rtems_attribute attribute_set,
5     rtems_task_priority priority_ceiling,
6     rtems_id        *id
7 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore created successfully
RTEMS_INVALID_NAME	invalid semaphore name
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_TOO_MANY	too many semaphores created
RTEMS_NOT_DEFINED	invalid attribute set
RTEMS_INVALID_NUMBER	invalid starting count for binary semaphore
RTEMS_MP_NOT_CONFIGURED	multiprocessing not configured
RTEMS_TOO_MANY	too many global objects

DESCRIPTION:

This directive creates a semaphore which resides on the local node. The created semaphore has the user-defined name specified in name and the initial count specified in count. For control and maintenance of the semaphore, RTEMS allocates and initializes a SMCB. The RTEMS-assigned semaphore id is returned in id. This semaphore id is used with other semaphore related directives to access the semaphore.

Specifying PRIORITY in attribute_set causes tasks waiting for a semaphore to be serviced according to task priority. When FIFO is selected, tasks are serviced in First In-First Out order.

NOTES:

This directive will not cause the calling task to be preempted.

The priority inheritance and priority ceiling algorithms are only supported for local, binary semaphores that use the priority task wait queue blocking discipline.

The following semaphore attribute constants are defined by RTEMS:

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, block on 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

Semaphores should not be made global unless remote tasks must interact with the cre-

ated 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 id must be transmitted to every node in the system for insertion in the local copy of the global object table.

Note, some combinations of attributes are not valid. See the earlier discussion on this.

The total number of global objects, including semaphores, is limited by the `maximum_global_objects` field in the Configuration Table.

It is not allowed to create an initially locked MrsP semaphore and the `RTEMS_INVALID_NUMBER` status code will be returned on SMP configurations in this case. This prevents lock order reversal problems with the allocator mutex.

12.4.2 SEMAPHORE_IDENT - Get ID of a semaphore

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_ident(
2   rtems_name  name,
3   uint32_t    node,
4   rtems_id    *id
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore identified successfully
RTEMS_INVALID_NAME	semaphore name not found
RTEMS_INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the semaphore id associated with the semaphore name. If the semaphore name is not unique, then the semaphore id will match one of the semaphores with that name. However, this semaphore id is not guaranteed to correspond to the desired semaphore. The semaphore id is used by other semaphore related directives to access the semaphore.

NOTES:

This directive will not cause the running task to be preempted.

If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

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.

12.4.3 SEMAPHORE_DELETE - Delete a semaphore

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore deleted successfully
RTEMS_INVALID_ID	invalid semaphore id
RTEMS_RESOURCE_IN_USE	binary semaphore is in use
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot delete remote semaphore

DESCRIPTION:

This directive deletes the semaphore specified by `id`. All tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. The SMCB for this semaphore is reclaimed by RTEMS.

NOTES:

The calling task will be preempted if it is enabled by the task's execution mode and a higher priority local task is waiting on the deleted semaphore. The calling task will NOT be preempted if all of the tasks that are waiting on the semaphore are remote tasks.

The calling task does not have to be the task that created the semaphore. Any local task that knows the semaphore id can delete the semaphore.

When a global semaphore is deleted, the semaphore id 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` option.

Proxies, used to represent remote tasks, are reclaimed when the semaphore is deleted.

12.4.4 SEMAPHORE_OBTAIN - Acquire a semaphore

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_obtain(
2     rtems_id      id,
3     rtems_option  option_set,
4     rtems_interval timeout
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore obtained successfully
RTEMS_UNSATISFIED	semaphore not available
RTEMS_TIMEOUT	timed out waiting for semaphore
RTEMS_OBJECT_WAS_DELETED	semaphore deleted while waiting
RTEMS_INVALID_ID	invalid semaphore id

DESCRIPTION:

This directive acquires the semaphore specified by `id`. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` components of the options parameter indicate whether the calling task wants to wait for the semaphore to become available or return immediately if the semaphore is not currently available. 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 acquired by returning immediately with a successful return code.

If the calling task chooses to return immediately and the current semaphore count is zero or negative, then a 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 or negative, then it is decremented by one and the calling task is placed on the semaphore's wait queue and blocked. If the semaphore was created with the `RTEMS_PRIORITY` attribute, then the calling task is inserted into the 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. If the binary semaphore was created with the `RTEMS_INHERIT_PRIORITY` attribute, then the priority of the task currently holding the binary semaphore is guaranteed to be greater than or equal to that of the blocking task. If the binary semaphore was created with the `RTEMS_PRIORITY_CEILING` 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 obtaining the semaphore is elevated to that of the ceiling.

The timeout parameter specifies the maximum interval the calling task is willing to be blocked waiting for the semaphore. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever. If the semaphore is available or the `RTEMS_NO_WAIT` option component is set, then timeout is ignored.

Deadlock situations are detected for MrsP semaphores and the `RTEMS_UNSATISFIED` status code will be returned on SMP configurations in this case.

NOTES:

The following semaphore acquisition option constants are defined by RTEMS:

RTEMS_WAIT	task will wait for semaphore (default)
RTEMS_NO_WAIT	task should not wait

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.

A clock tick is required to support the timeout functionality of this directive.

It is not allowed to obtain a MrsP semaphore more than once by one task at a time (nested access) and the `RTEMS_UNSATISFIED` status code will be returned on SMP configurations in this case.

12.4.5 SEMAPHORE_RELEASE - Release a semaphore

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_release(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore released successfully
RTEMS_INVALID_ID	invalid semaphore id
RTEMS_NOT_OWNER_OF_RESOURCE	calling task does not own semaphore
RTEMS_INCORRECT_STATE	invalid unlock order

DESCRIPTION:

This directive releases the semaphore specified by id. The semaphore count is incremented by one. If the count is zero or negative, then the first task on this semaphore's wait queue is removed and unblocked. The unblocked task may preempt the running task if the running task's preemption mode is enabled and the unblocked task has a higher priority than the running task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

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.

The outermost release of a local, binary, priority inheritance or priority ceiling semaphore may result in the calling task having its priority lowered. This will occur if the calling task holds no other binary

semaphores and it has inherited a higher priority.

The MrsP semaphores must be released in the reversed obtain order, otherwise the RTEMS_INCORRECT_STATE status code will be returned on SMP configurations in this case.

12.4.6 SEMAPHORE_FLUSH - Unblock all tasks waiting on a semaphore

be returned on SMP configurations in this case.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_flush(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	semaphore released successfully
RTEMS_INVALID_ID	invalid semaphore id
RTEMS_NOT_DEFINED	operation not defined for the protocol of the semaphore
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	not supported for remote semaphores

DESCRIPTION:

This directive unblocks all tasks waiting on the semaphore specified by `id`. Since there are tasks blocked on the semaphore, the semaphore's count is not changed by this directive and thus is zero before and after this directive is executed. Tasks which are unblocked as the result of this directive will return from the `rtems_semaphore_obtain` directive with a status code of `RTEMS_UNSATISFIED` to indicate that the semaphore was not obtained.

This directive may unblock any number of tasks. Any of the unblocked tasks may preempt the running task if the running task's preemption mode is enabled and an unblocked task has a higher priority than the running task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

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 MrsP semaphore and the `RTEMS_NOT_DEFINED` status code will

12.4.7 SEMAPHORE_SET_PRIORITY - Set priority by scheduler for a semaphore

- For other protocols this operation is not defined.

EXAMPLE:

CALLING SEQUENCE:

```

1 rtems_status_code rtems_semaphore_set_
  ↪ priority(
2   rtems_id          semaphore_id,
3   rtems_id          scheduler_id,
4   rtems_task_priority new_priority,
5   rtems_task_priority *old_priority
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successful operation
RTEMS_INVALID_ID	invalid semaphore or scheduler id
RTEMS_INVALID_ADDRESS	old_priority is NULL
RTEMS_INVALID_PRIORITY	invalid new priority value
RTEMS_NOT_DEFINED	operation not defined for the protocol of the semaphore
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	not supported for remote semaphores

DESCRIPTION:

This directive sets the priority value with respect to the specified scheduler of a semaphore.

The special priority value `RTEMS_CURRENT_PRIORITY` can be used to get the current priority value without changing it.

The interpretation of the priority value depends on the protocol of the semaphore object.

- The Multiprocessor Resource Sharing Protocol needs a ceiling priority per scheduler instance. This operation can be used to specify these priority values.
- For the Priority Ceiling Protocol the ceiling priority is used with this operation.

```

1 #include <assert.h>
2 #include <stdlib.h>
3 #include <rtems.h>
4
5 #define SCHED_A rtems_build_name(' ', ' ',
  ↪ ' ', 'A')
6 #define SCHED_B rtems_build_name(' ', ' ',
  ↪ ' ', 'B')
7
8 static void Init(rtems_task_argument arg)
9 {
10     rtems_status_code sc;
11     rtems_id          semaphore_id;
12     rtems_id          scheduler_a_id;
13     rtems_id          scheduler_b_id;
14     rtems_task_priority prio;
15
16     /* Get the scheduler identifiers */
17     sc = rtems_scheduler_ident(SCHED_A, &
  ↪ scheduler_a_id);
18     assert(sc == RTEMS_SUCCESSFUL);
19     sc = rtems_scheduler_ident(SCHED_B, &
  ↪ scheduler_b_id);
20     assert(sc == RTEMS_SUCCESSFUL);
21
22     /* Create a MrsP semaphore object */
23     sc = rtems_semaphore_create(
24     ↪ rtems_build_name('M', 'R', 'S', 'P'
  ↪ '),
25         1,
26         RTEMS_MULTIPROCESSOR_RESOURCE_
  ↪ SHARING | RTEMS_BINARY_SEMAPHORE,
27         1,
28         &semaphore_id
29     );
30     assert(sc == RTEMS_SUCCESSFUL);
31
32     /*
33      * The ceiling priority values
  ↪ per scheduler are equal to the value
  ↪ specified
34      * for object creation.
35      */
36     prio = RTEMS_CURRENT_PRIORITY;
37     sc = rtems_semaphore_set_
  ↪ priority(semaphore_id, scheduler_a_id,
  ↪ prio, &prio);
38     assert(sc == RTEMS_SUCCESSFUL);
39     assert(prio == 1);
40
41     /* Check the old value and set a new
  ↪ ceiling priority for scheduler B */

```

```

42     prio = 2;
43         sc = rtems_semaphore_set_
↳priority(semaphore_id, scheduler_b_id,
↳prio, &prio);
44     assert(sc == RTEMS_SUCCESSFUL);
45     assert(prio == 1);
46
47     /* Check the ceiling priority values */
48     prio = RTEMS_CURRENT_PRIORITY;
49         sc = rtems_semaphore_set_
↳priority(semaphore_id, scheduler_a_id,
↳prio, &prio);
50     assert(sc == RTEMS_SUCCESSFUL);
51     assert(prio == 1);
52     prio = RTEMS_CURRENT_PRIORITY;
53         sc = rtems_semaphore_set_
↳priority(semaphore_id, scheduler_b_id,
↳prio, &prio);
54     assert(sc == RTEMS_SUCCESSFUL);
55     assert(prio == 2);
56     sc = rtems_semaphore_delete(semaphore_
↳id);
57     assert(sc == RTEMS_SUCCESSFUL);
58     exit(0);
59 }
60
61 #define CONFIGURE_SMP_APPLICATION
62 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_
↳DRIVER
63 #define CONFIGURE_APPLICATION_NEEDS_
↳CONSOLE_DRIVER
64 #define CONFIGURE_MAXIMUM_TASKS 1
65 #define CONFIGURE_MAXIMUM_SEMAPHORES 1
66 #define CONFIGURE_MAXIMUM_MRSP_SEMAPHORES_
↳1
67 #define CONFIGURE_SMP_MAXIMUM_PROCESSORS 2
68 #define CONFIGURE_SCHEDULER_SIMPLE_SMP
69
70 #include <rtems/scheduler.h>
71
72 RTEMS_SCHEDULER_CONTEXT_SIMPLE_SMP(a);
73 RTEMS_SCHEDULER_CONTEXT_SIMPLE_SMP(b);
74
75 #define CONFIGURE_SCHEDULER_CONTROLS \
76     RTEMS_SCHEDULER_CONTROL_SIMPLE_
↳SMP(a, SCHED_A), \
77     RTEMS_SCHEDULER_CONTROL_SIMPLE_
↳SMP(b, SCHED_B)
78 #define CONFIGURE_SMP_SCHEDULER_
↳ASSIGNMENTS \
79     RTEMS_SCHEDULER_ASSIGN(0, RTEMS_
↳SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
↳\
80     RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳SCHEDULER_ASSIGN_PROCESSOR_MANDATORY)
81 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
82 #define CONFIGURE_INIT

```

```
83 #include <rtems/confdefs.h>
```

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 190) - Create a barrier
- *rtems_barrier_ident* (page 191) - Get ID of a barrier
- *rtems_barrier_delete* (page 192) - Delete a barrier
- *rtems_barrier_wait* (page 193) - Wait at a barrier
- *rtems_barrier_release* (page 194) - Release a 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 Operations

13.3.1 Creating a Barrier

The `rtems_barrier_create` directive creates a barrier with a user-specified name and the desired attributes. RTEMS allocates a Barrier Control Block (BCB) from the BCB free list. This data structure is used by RTEMS to manage the newly created barrier. Also, a unique barrier ID is generated and returned to the calling task.

13.3.2 Obtaining Barrier IDs

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

13.3.3 Waiting at a Barrier

The `rtems_barrier_wait` directive is used to wait at the specified barrier. Since a barrier is, by definition, never immediately, the task may wait forever for the barrier to be released or it may specify a timeout. Specifying a timeout limits the interval the task will wait before returning with an error status code.

If the barrier is configured as automatic and there are already one less than the maximum number of waiters, then the call will unblock all tasks waiting at the barrier and the caller will return immediately.

When the task does wait to acquire the barrier, then it is placed in the barrier's task wait queue in FIFO order. All tasks waiting on a barrier are returned an error code when the barrier is deleted.

13.3.4 Releasing a Barrier

The `rtems_barrier_release` directive is used to release the specified barrier. When the `rtems_barrier_release` is invoked, all tasks waiting at the barrier are immediately made ready to execute and begin to compete for the processor to execute.

13.3.5 Deleting a Barrier

The `rtems_barrier_delete` directive removes a barrier from the system and frees its control block. A barrier can be deleted by any local task that knows the barrier's ID. As a result of this directive, 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. Any subsequent references to the barrier's name and ID are invalid.

13.4 Directives

This section details the barrier 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.

13.4.1 BARRIER_CREATE - Create a barrier

CALLING SEQUENCE:

```

1 rtems_status_code rtems_barrier_create(
2     rtems_name      name,
3     rtems_attribute attribute_set,
4     uint32_t        maximum_waiters,
5     rtems_id        *id
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	barrier created successfully
RTEMS_INVALID_NAME	invalid barrier name
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_TOO_MANY	too many barriers created

DESCRIPTION:

This directive creates a barrier which resides on the local node. The created barrier has the user-defined name specified in `name` and the initial count specified in `count`. For control and maintenance of the barrier, RTEMS allocates and initializes a BCB. The RTEMS-assigned barrier id is returned in `id`. This barrier id is used with other barrier related directives to access the barrier.

RTEMS_BARRIER_MANUAL_RELEASE	only release
------------------------------	--------------

Specifying `RTEMS_BARRIER_AUTOMATIC_RELEASE` in `attribute_set` causes tasks calling the `rtems_barrier_wait` directive to block until there are `maximum_waiters - 1` tasks waiting at the barrier. When the `maximum_waiters` task invokes the `rtems_barrier_wait` directive, the previous `maximum_waiters - 1` tasks are automatically released and the caller returns.

In contrast, when the `RTEMS_BARRIER_MANUAL_RELEASE` attribute is specified, there is no limit on the number of tasks that will block at the barrier. Only when the `rtems_barrier_release` directive is invoked, are the tasks waiting at the barrier unblocked.

NOTES:

This directive will not cause the calling task to be preempted.

The following barrier attribute constants are defined by RTEMS:

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 <code>rtems_barrier_release</code> directive. (default)

13.4.2 BARRIER_IDENT - Get ID of a barrier

CALLING SEQUENCE:

```

1 rtems_status_code rtems_barrier_ident(
2   rtems_name      name,
3   rtems_id        *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	barrier identified successfully
RTEMS_INVALID_NAME	barrier name not found
RTEMS_INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the barrier id associated with the barrier name. If the barrier name is not unique, then the barrier id will match one of the barriers with that name. However, this barrier id is not guaranteed to correspond to the desired barrier. The barrier id is used by other barrier related directives to access the barrier.

NOTES:

This directive will not cause the running task to be preempted.

13.4.3 BARRIER_DELETE - Delete a barrier

CALLING SEQUENCE:

```

1 rtems_status_code rtems_barrier_delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	barrier deleted successfully
RTEMS_INVALID_ID	invalid barrier id

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. The BCB for this barrier is reclaimed by RTEMS.

NOTES:

The calling task will be preempted if it is enabled by the task's execution mode and a higher priority local task is waiting on the deleted barrier. The calling task will NOT be preempted if all of the tasks that are waiting on the barrier are remote tasks.

The calling task does not have to be the task that created the barrier. Any local task that knows the barrier id can delete the barrier.

13.4.4 BARRIER_OBTAIN - Acquire a barrier

CALLING SEQUENCE:

```

1 rtems_status_code rtems_barrier_wait(
2   rtems_id      id,
3   rtems_interval timeout
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	barrier released and task unblocked
RTEMS_UNSATISFIED	barrier not available
RTEMS_TIMEOUT	timed out waiting for barrier
RTEMS_OBJECT_WAS_DELETED	barrier deleted while waiting
RTEMS_INVALID_ID	invalid barrier id

DESCRIPTION:

This directive acquires the barrier specified by `id`. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` components of the options parameter indicate whether the calling task wants to wait for the barrier to become available or return immediately if the barrier is not currently available. With either `RTEMS_WAIT` or `RTEMS_NO_WAIT`, if the current barrier count is positive, then it is decremented by one and the barrier is successfully acquired by returning immediately with a successful return code.

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 is the Nth task which trips the automatic release condition.

The timeout parameter specifies the maximum interval the calling task is willing to be blocked waiting for the barrier. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever. If the barrier is available or the `RTEMS_NO_WAIT` option component is set, then timeout is ignored.

NOTES:

The following barrier acquisition option constants are defined by RTEMS:

RTEMS_WAIT	task will wait for barrier (default)
RTEMS_NO_WAIT	task should not wait

A clock tick is required to support the timeout functionality of this directive.

13.4.5 BARRIER_RELEASE - Release a barrier

CALLING SEQUENCE:

```
1 rtems_status_code rtems_barrier_release(  
2   rtems_id id,  
3   uint32_t *released  
4 );
```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	barrier released successfully
RTEMS_ INVALID_ID	invalid barrier id

DESCRIPTION:

This directive releases the barrier specified by id. All tasks waiting at the barrier will be unblocked. If the running task's preemption mode is enabled and one of the unblocked tasks has a higher priority than the running task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

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 202) - Create a queue
- *rtems_message_queue_ident* (page 203) - Get ID of a queue
- *rtems_message_queue_delete* (page 204) - Delete a queue
- *rtems_message_queue_send* (page 205) - Put message at rear of a queue
- *rtems_message_queue_urgent* (page 206) - Put message at front of a queue
- *rtems_message_queue_broadcast* (page 207) - Broadcast N messages to a queue
- *rtems_message_queue_receive* (page 208) - Receive message from a queue
- *rtems_message_queue_get_number_pending* (page 209) - Get number of messages pending on a queue
- *rtems_message_queue_flush* (page 210) - Flush all messages on a queue

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 message 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.

14.4.1 MESSAGE_QUEUE_CREATE - Create a queue

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_message_queue_
   ↪ create(
2     rtems_name         name,
3     uint32_t           count,
4     size_t             max_message_size,
5     rtems_attribute   attribute_set,
6     rtems_id          *id
7 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	queue created successfully
RTEMS_INVALID_NAME	invalid queue name
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NUMBER	invalid message count
RTEMS_INVALID_SIZE	invalid message size
RTEMS_TOO_MANY	too many queues created
RTEMS_UNSATISFIED	unable to allocate message buffers
RTEMS_MP_NOT_CONFIGURED	multiprocessing not configured
RTEMS_TOO_MANY	too many global objects

DESCRIPTION:

This directive creates a message queue which resides on the local node with the user-defined name specified in `name`. For control and maintenance of the queue, RTEMS allocates and initializes a QCB. Memory is allocated from the RTEMS Workspace for the specified count of messages, each of `max_message_size` bytes in length. The RTEMS-assigned queue id, returned in `id`, is used to access the message queue.

Specifying `RTEMS_PRIORITY` in `attribute_set` causes tasks waiting for a message to be serviced according to task priority. When `RTEMS_FIFO` is specified, waiting tasks are serviced in First In-First Out order.

NOTES:

This directive will not cause the calling task to be preempted.

The following message queue attribute constants are defined by RTEMS:

RTEMS_FIFO	tasks wait by FIFO (default)
RTEMS_PRIORITY	tasks wait by priority
RTEMS_LOCAL	local message queue (default)
RTEMS_GLOBAL	global message queue

Message queues should not be made global unless remote tasks must interact with the created 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 id must be transmitted to every node in the system for insertion in the local copy of the global object table.

For GLOBAL message queues, the maximum message size is effectively limited to the longest message which the MPCIE is capable of transmitting.

The total number of global objects, including message queues, is limited by the `maximum_global_objects` field in the configuration table.

14.4.2 MESSAGE_QUEUE_IDENT - Get ID of a queue

CALLING SEQUENCE:

```

1 rtems_status_code  rtems_message_queue_
  ↪ident(
2   rtems_name  name,
3   uint32_t    node,
4   rtems_id    *id
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	queue identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	queue name not found
RTEMS_INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the queue id associated with the queue name specified in name. If the queue name is not unique, then the queue id will match one of the queues with that name. However, this queue id is not guaranteed to correspond to the desired queue. The queue id is used with other message related directives to access the message queue.

NOTES:

This directive will not cause the running task to be preempted.

If node is RTEMS_SEARCH_ALL_NODES, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

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.

14.4.3 MESSAGE_QUEUE_DELETE - are reclaimed when the message queue is deleted.
Delete a queue

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_message_queue_
  ↪delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	queue deleted successfully
RTEMS_INVALID_ID	invalid queue id
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot delete remote queue

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. If no tasks are waiting, but the queue contains messages, then RTEMS returns these message buffers back to the system message buffer pool. The QCB for this queue as well as the memory for the message buffers is reclaimed by RTEMS.

NOTES:

The calling task will be preempted if its preemption mode is enabled and one or more local tasks with a higher priority than the calling task are waiting on the deleted queue. The calling task will NOT be preempted if the tasks that are waiting are remote tasks.

The calling task does not have to be the task that created the queue, although the task and queue must reside on the same node.

When the queue is deleted, any messages in the queue are returned to the free message buffer pool. Any information stored in those messages is lost.

When a global message queue is deleted, the message queue id must be transmitted to every node in the system for deletion from the local copy of the global object table.

Proxies, used to represent remote tasks,

14.4.4 MESSAGE_QUEUE_SEND - Put message at rear of a queue

ate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

CALLING SEQUENCE:

```

1 rtems_status_code rtems_message_queue_
  ↪ send(
2   rtems_id id,
3   cons void *buffer,
4   size_t size
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	message sent successfully
RTEMS_INVALID_ID	invalid queue id
RTEMS_INVALID_SIZE	invalid message size
RTEMS_INVALID_ADDRESS	buffer is NULL
RTEMS_UNSATISFIED	out of message buffers
RTEMS_TOO_MANY	queue's limit has been reached

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.

NOTES:

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 a message to a global message queue which does not reside on the local node will generate a request to the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropri-

14.4.5 MESSAGE_QUEUE_URGENT - Put message at front of a queue

ate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_message_queue_
  ↪urgent(
2   rtems_id      id,
3   const void *buffer,
4   size_t       size
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_	message sent
SUCCESSFUL	successfully
RTEMS_INVALID_	invalid queue id
ID	
RTEMS_INVALID_	invalid message size
SIZE	
RTEMS_INVALID_	buffer is NULL
ADDRESS	
RTEMS_	out of message
UNSATISFIED	buffers
RTEMS_TOO_MANY	queue's limit has
	been reached

DESCRIPTION:

This directive sends the message buffer of size bytes in length to the queue specified by id. If a task is waiting on the queue, then the message is copied to the task's buffer and the task is unblocked. If no tasks are waiting on 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.

NOTES:

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 a message to a global message queue which does not reside on the local node will generate a request telling the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropri-

14.4.6 MESSAGE_QUEUE_BROADCAST - Broadcast N messages to a queue

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_message_queue_
  ↳broadcast(
2   rtems_id      id,
3   const void    *buffer,
4   size_t        size,
5   uint32_t      *count
6 );

```

a different node from the message queue, a copy of the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	message broadcasted successfully
RTEMS_INVALID_ID	invalid queue id
RTEMS_INVALID_ADDRESS	buffer is NULL
RTEMS_INVALID_ADDRESS	count is NULL
RTEMS_INVALID_SIZE	invalid message size

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.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

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`.

Broadcasting a message to a global message queue which does not reside on the local node will generate a request telling the remote node to broadcast the message to the specified message queue.

When a task is unblocked which resides on

14.4.7 MESSAGE_QUEUE_RECEIVE - Receive message from a queue

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_message_queue_
  ↳receive(
2   rtems_id           id,
3   void               *buffer,
4   size_t             *size,
5   rtems_option       option_set,
6   rtems_interval     timeout
7 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	message received successfully
RTEMS_INVALID_ID	invalid queue id
RTEMS_INVALID_ADDRESS	buffer is NULL
RTEMS_INVALID_ADDRESS	size is NULL
RTEMS_UNSATISFIED	queue is empty
RTEMS_TIMEOUT	timed out waiting for message
RTEMS_OBJECT_WAS_DELETED	queue deleted while waiting

DESCRIPTION:

This directive receives a message from the message queue specified in `id`. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` options of the options parameter allow the calling task to specify whether to wait for a message to become available or return immediately. For either option, if there is at least one message in the queue, then it is copied to `buffer`, `size` is set to return the length of the message in bytes, and this directive returns immediately with a successful return 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 a status code indicating this condition is returned. 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.

A task choosing to wait at the queue can optionally specify a timeout value in the timeout parameter. The timeout parameter specifies the maximum interval to wait before the calling task desires to be unblocked. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

NOTES:

The following message receive option constants are defined by RTEMS:

RTEMS_WAIT	task will wait for a message (default)
RTEMS_NO_WAIT	task should not wait

Receiving a message from a global message queue which does not reside on the local node will generate a request to the remote node to obtain a message from the specified message queue. If no message is available and `RTEMS_WAIT` was specified, then the task must be blocked until a message is posted. A proxy is allocated on the remote node to represent the task until the message is posted.

A clock tick is required to support the timeout functionality of this directive.

14.4.8 MESSAGE_QUEUE_GET_NUMBER_PENDING

- Get number of messages pending on a queue

CALLING SEQUENCE:

```

1 rtems_status_code rtems_message_queue_get_
  ↳number_pending(
2   rtems_id id,
3   uint32_t *count
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	number of messages pending returned successfully
RTEMS_ INVALID_ ADDRESS	count is NULL
RTEMS_ INVALID_ID	invalid queue id

DESCRIPTION:

This directive returns the number of messages pending on this message queue in count. If no messages are present on the queue, count is set to zero.

NOTES:

Getting the number of pending messages on a global message queue which does not reside on the local node will generate a request to the remote node to actually obtain the pending message count for the specified message queue.

14.4.9 MESSAGE_QUEUE_FLUSH - Flush all messages on a queue

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_message_queue_
  ↪flush(
2   rtems_id id,
3   uint32_t *count
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	message queue flushed successfully
RTEMS_INVALID_ADDRESS	count is NULL
RTEMS_INVALID_ID	invalid queue id

DESCRIPTION:

This directive removes all pending messages from the specified queue id. The number of messages removed is returned in count. If no messages are present on the queue, count is set to zero.

NOTES:

Flushing all messages on a global message queue which does not reside on the local node will generate a request to the remote node to actually flush the specified message queue.

EVENT MANAGER

15.1 Introduction

The event manager provides a high performance method of intertask communication and synchronization. The directives provided by the event manager are:

- *rtems_event_send* (page 216) - Send event set to a task
- *rtems_event_receive* (page 217) - Receive event condition

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:

<code>RTEMS_WAIT</code>	task will wait for event (default)
<code>RTEMS_NO_WAIT</code>	task should not wait
<code>RTEMS_EVENT_ALL</code>	return after all events (default)
<code>RTEMS_EVENT_ANY</code>	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 event 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.

15.4.1 EVENT_SEND - Send event set to a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_event_send (
2   rtems_id      id,
3   rtems_event_set event_in
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	event set sent successfully
RTEMS_INVALID_ID	invalid task id

DESCRIPTION:

This directive sends an event set, `event_in`, to the task specified by `id`. If a blocked task's input event condition is satisfied by this directive, then it will be made ready. If its input event condition is not satisfied, then the events satisfied are updated and the events not satisfied are left pending. If the task specified by `id` is not blocked waiting for events, then the events sent are left pending.

NOTES:

Specifying `RTEMS_SELF` for `id` results in the event set being sent to the calling task.

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` 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.

15.4.2 EVENT_RECEIVE - Receive event condition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_event_receive (
2   rtems_event_set event_in,
3   rtems_option   option_set,
4   rtems_interval ticks,
5   rtems_event_set *event_out
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	event received successfully
RTEMS_UNSATISFIED	input event not satisfied (RTEMS_NO_WAIT)
RTEMS_INVALID_ADDRESS	event_out is NULL
RTEMS_TIMEOUT	timed out waiting for event

DESCRIPTION:

This directive attempts to receive the event condition specified in `event_in`. If `event_in` is set to `RTEMS_PENDING_EVENTS`, then the current pending events are returned in `event_out` and left pending. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` options in the `option_set` parameter are used to specify whether or not the task is willing to wait for the event condition to be satisfied. `RTEMS_EVENT_ANY` and `RTEMS_EVENT_ALL` are used in the `option_set` parameter are used to specify whether a single event or the complete event set is necessary to satisfy the event condition. The `event_out` parameter is returned to the calling task with the value that corresponds to the events in `event_in` that were satisfied.

If pending events satisfy the event condition, then `event_out` is set to the satisfied events and the pending events in the event condition are cleared. If the event condition

is not satisfied and `RTEMS_NO_WAIT` is specified, then `event_out` is set to the currently satisfied events. If the calling task chooses to wait, then it will block waiting for the event condition.

If the calling task must wait for the event condition to be satisfied, then the timeout parameter is used to specify the maximum interval to wait. If it is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

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.

The following event receive option constants are defined by RTEMS:

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

A clock tick is required to support the functionality of this directive.

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 225) - Establish an ASR
- *rtems_signal_send* (page 226) - Send signal set to a 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:

```

1 rtems_asr user_routine(
2     rtems_signal_set signals
3 );
```

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 signal 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.

16.4.1 SIGNAL_CATCH - Establish an ASR

CALLING SEQUENCE:

```

1 rtems_status_code rtems_signal_catch(
2     rtems_asr_entry asr_handler,
3     rtems_mode      mode
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	always successful
------------------	-------------------

DESCRIPTION:

This directive establishes an asynchronous signal routine (ASR) for the calling task. The `asr_handler` parameter specifies the entry point of the ASR. If `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` parameter specifies the execution mode for the ASR. This execution mode supersedes the task's execution mode while the ASR is executing.

NOTES:

This directive will not cause the calling task to be preempted.

The following task mode constants are defined by RTEMS:

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

16.4.2 SIGNAL_SEND - Send signal set to a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_signal_send(
2     rtems_id      id,
3     rtems_signal_set signal_set
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	signal sent successfully
RTEMS_INVALID_ID	task id invalid
RTEMS_INVALID_NUMBER	empty signal set
RTEMS_NOT_DEFINED	ASR invalid

DESCRIPTION:

This directive sends a signal set to the task specified in `id`. The `signal_set` parameter contains the signal set to be sent to the task.

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.

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.

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 232) - Create a partition
- *rtems_partition_ident* (page 233) - Get ID of a partition
- *rtems_partition_delete* (page 234) - Delete a partition
- *rtems_partition_get_buffer* (page 235) - Get buffer from a partition
- *rtems_partition_return_buffer* (page 236) - Return buffer to a 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 (PCB) from the PCB 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 PCB for that partition is returned to the PCB 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 partition 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.

17.4.1 PARTITION_CREATE - Create a partition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_partition_create(
2   rtems_name      name,
3   void            *starting_address,
4   uint32_t        length,
5   uint32_t        buffer_size,
6   rtems_attribute attribute_set,
7   rtems_id        *id
8 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	partition created successfully
RTEMS_INVALID_NAME	invalid partition name
RTEMS_TOO_MANY	too many partitions created
RTEMS_INVALID_ADDRESS	address not on four byte boundary
RTEMS_INVALID_ADDRESS	starting_address is NULL
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_SIZE	length or buffer size is 0
RTEMS_INVALID_SIZE	length is less than the buffer size
RTEMS_INVALID_SIZE	buffer size not a multiple of 4
RTEMS_MP_NOT_CONFIGURED	multiprocessing not configured
RTEMS_TOO_MANY	too many global objects

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 assigned partition id is returned in `id`. This partition id is used to access the partition with other partition related directives. For control and maintenance of the partition, RTEMS allocates a PTCB from the local PTCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

The `starting_address` must be properly aligned for the target architecture.

The `buffer_size` parameter must be a multiple of the CPU alignment factor. 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.

Memory from the partition is not used by RTEMS to store the Partition Control Block.

The following partition attribute constants are defined by RTEMS:

RTEMS_LOCAL	local partition (default)
RTEMS_GLOBAL	global partition

The PTCB for a global partition is allocated on the local node. The memory space used for the partition must reside in shared memory. 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 id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including partitions, is limited by the `maximum_global_objects` field in the Configuration Table.

17.4.2 PARTITION_IDENT - Get ID of a partition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_partition_ident(
2   rtems_name name,
3   uint32_t node,
4   rtems_id *id
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	partition identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	partition name not found
RTEMS_INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the partition id associated with the partition name. If the partition name is not unique, then the partition id will match one of the partitions with that name. However, this partition id is not guaranteed to correspond to the desired partition. The partition id is used with other partition related directives to access the partition.

NOTES:

This directive will not cause the running task to be preempted.

If `node` is `RTEMS_SEARCH_ALL_NODES`, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

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.

17.4.3 PARTITION_DELETE - Delete a partition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_partition_delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	partition deleted successfully
RTEMS_INVALID_ID	invalid partition id
RTEMS_RESOURCE_IN_USE	buffers still in use
RTEMS_ILLEGAL_ON_REMOTE_OBJECT	cannot delete remote partition

DESCRIPTION:

This directive deletes the partition specified by id. The partition cannot be deleted if any of its buffers are still allocated. The PTCB for the deleted partition is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the partition. Any local task that knows the partition id can delete the partition.

When a global partition is deleted, the partition id 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 option.

17.4.4 PARTITION_GET_BUFFER - Get buffer from a partition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_partition_get_
  ↪buffer(
2   rtems_id id,
3   void **buffer
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	buffer obtained successfully
RTEMS_INVALID_ADDRESS	buffer is NULL
RTEMS_INVALID_ID	invalid partition id
RTEMS_UNSATISFIED	all buffers are allocated

DESCRIPTION:

This directive allows a buffer to be obtained from the partition specified in `id`. The address of the allocated buffer is returned in `buffer`.

NOTES:

This directive will not cause the running task to be preempted.

All buffers begin on a four byte boundary.

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 specified partition.

17.4.5 PARTITION_RETURN_BUFFER - Return buffer to a partition

CALLING SEQUENCE:

```

1 rtems_status_code rtems_partition_return_
  ↪buffer(
2   rtems_id id,
3   void *buffer
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	buffer returned successfully
RTEMS_INVALID_ADDRESS	buffer is NULL
RTEMS_INVALID_ID	invalid partition id
RTEMS_INVALID_ADDRESS	buffer address not in partition

DESCRIPTION:

This directive returns the buffer specified by buffer to the partition specified by id.

NOTES:

This directive will not cause the running task to be preempted.

Returning a buffer to a global partition which does not reside on the local node will generate a request telling the remote node to return the buffer to the specified partition.

Returning a buffer multiple times is an error. It will corrupt the internal state of the partition.

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 243) - Create a region
- *rtems_region_ident* (page 244) - Get ID of a region
- *rtems_region_delete* (page 245) - Delete a region
- *rtems_region_extend* (page 246) - Add memory to a region
- *rtems_region_get_segment* (page 247) - Get segment from a region
- *rtems_region_return_segment* (page 248) - Return segment to a region
- *rtems_region_get_segment_size* (page 249) - Obtain size of a segment
- *rtems_region_resize_segment* (page 250) - Change size of a 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.

Note: Please see the release notes or RTEMS source code for information regarding restrictions on the location of the memory being added in relation to memory already in the region.

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 region 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.

18.4.1 REGION_CREATE - Create a region

CALLING SEQUENCE:

```

1 rtems_status_code rtems_region_create(
2     rtems_name      name,
3     void            *starting_address,
4     intptr_t        length,
5     uint32_t         page_size,
6     rtems_attribute attribute_set,
7     rtems_id        *id
8 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	region created successfully
RTEMS_INVALID_NAME	invalid region name
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_ADDRESS	starting_address is NULL
RTEMS_INVALID_ADDRESS	address not on four byte boundary
RTEMS_TOO_MANY	too many regions created
RTEMS_INVALID_SIZE	invalid page size

DESCRIPTION:

This directive creates a region from a physically contiguous memory space which starts at `starting_address` and is `length` bytes long. Segments allocated from the region will be a multiple of `page_size` bytes in length. The assigned region id is returned in `id`. This region id is used as an argument to other region related directives to access the region.

For control and maintenance of the region, RTEMS allocates and initializes an RNCB from the RNCB free pool. Thus memory from the region is not used to store the RNCB. However, some overhead within the region is required by RTEMS each time a segment is constructed in the region.

Specifying `RTEMS_PRIORITY` in `attribute_set` causes tasks waiting for a segment to be serviced according to task priority. Specify-

ing `RTEMS_FIFO` in `attribute_set` or selecting `RTEMS_DEFAULT_ATTRIBUTES` will cause waiting tasks to be serviced in First In-First Out order.

The `starting_address` parameter must be aligned on a four byte boundary. The `page_size` parameter must be a multiple of four greater than or equal to eight.

NOTES:

This directive will not cause the calling task to be preempted.

The following region attribute constants are defined by RTEMS:

RTEMS_FIFO	tasks wait by FIFO (default)
RTEMS_PRIORITY	tasks wait by priority

18.4.2 REGION_IDENT - Get ID of a region

CALLING SEQUENCE:

```

1 rtems_status_code rtems_region_ident(
2   rtems_name name,
3   rtems_id *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	region identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	region name not found

DESCRIPTION:

This directive obtains the region id associated with the region name to be acquired. If the region name is not unique, then the region id will match one of the regions with that name. However, this region id is not guaranteed to correspond to the desired region. The region id is used to access this region in other region manager directives.

NOTES:

This directive will not cause the running task to be preempted.

18.4.3 REGION_DELETE - Delete a region

CALLING SEQUENCE:

```
1 rtems_status_code rtems_region_delete(  
2   rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	region deleted successfully
RTEMS_INVALID_ID	invalid region id
RTEMS_RESOURCE_IN_USE	segments still in use

DESCRIPTION:

This directive deletes the region specified by `id`. The region cannot be deleted if any of its segments are still allocated. The RNCB for the deleted region is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the region. Any local task that knows the region id can delete the region.

18.4.4 REGION_EXTEND - Add memory to a region

CALLING SEQUENCE:

```

1 rtems_status_code rtems_region_extend(
2   rtems_id id,
3   void *starting_address,
4   intptr_t length
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	region extended successfully
RTEMS_INVALID_ADDRESS	starting_address is NULL
RTEMS_INVALID_ID	invalid region id
RTEMS_INVALID_ADDRESS	invalid address of area to add

DESCRIPTION:

This directive adds the memory which starts at `starting_address` for `length` bytes to the region specified by `id`.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the region. Any local task that knows the region id can extend the region.

18.4.5 REGION_GET_SEGMENT - Get segment from a region

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_region_get_
  ↪segment(
2   rtems_id             id,
3   intptr_t             size,
4   rtems_option         option_set,
5   rtems_interval       timeout,
6   void                 **segment
7 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	segment obtained successfully
RTEMS_INVALID_ADDRESS	segment is NULL
RTEMS_INVALID_ID	invalid region id
RTEMS_INVALID_SIZE	request is for zero bytes or exceeds the size of maximum segment which is possible for this region
RTEMS_UNSATISFIED	segment of requested size not available
RTEMS_TIMEOUT	timed out waiting for segment
RTEMS_OBJECT_WAS_DELETED	region deleted while waiting

DESCRIPTION:

This directive obtains a variable size segment from the region specified by `id`. The address of the allocated segment is returned in `segment`. The `RTEMS_WAIT` and `RTEMS_NO_WAIT` components of the options parameter are used to specify whether the calling tasks wish to wait for a segment to become available or return immediately if no segment is available. For either option, if a sufficiently sized segment is available, then the segment is successfully acquired by returning immediately with the `RTEMS_SUCCESSFUL` status code.

If the calling task chooses to return imme-

diately and a segment large enough is not available, then an error code indicating this fact is returned. If the calling task chooses to wait for the segment and a segment large enough is not available, then the calling task is placed on the region's segment wait queue and blocked. If the region was created with the `RTEMS_PRIORITY` option, then the calling task is inserted into the wait queue according to its priority. However, if the region was created with the `RTEMS_FIFO` option, then the calling task is placed at the rear of the wait queue.

The timeout parameter specifies the maximum interval that a task is willing to wait to obtain a segment. If timeout is set to `RTEMS_NO_TIMEOUT`, then the calling task will wait forever.

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.

The following segment acquisition option constants are defined by RTEMS:

RTEMS_WAIT	task will wait for segment (default)
RTEMS_NO_WAIT	task should not wait

A clock tick is required to support the timeout functionality of this directive.

18.4.6 REGION_RETURN_SEGMENT - Return segment to a region

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_region_return_
  ↪segment(
2   rtems_id id,
3   void    *segment
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	segment returned successfully
RTEMS_INVALID_ADDRESS	segment is NULL
RTEMS_INVALID_ID	invalid region id
RTEMS_INVALID_ADDRESS	segment address not in region

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.

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.

18.4.7 REGION_GET_SEGMENT_SIZE - Obtain size of a segment

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_region_get_
  ↪segment_size(
2   rtems_id id,
3   void *segment,
4   ssize_t *size
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	segment obtained successfully
RTEMS_INVALID_ADDRESS	segment is NULL
RTEMS_INVALID_ADDRESS	size is NULL
RTEMS_INVALID_ID	invalid region id
RTEMS_INVALID_ADDRESS	segment address not in region

DESCRIPTION:

This directive obtains the size in bytes of the specified segment.

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.

18.4.8 REGION_RESIZE_SEGMENT - Change size of a segment

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_region_resize_
  ↪segment(
2   rtems_id          id,
3   void              *segment,
4   ssize_t           size,
5   ssize_t           *old_size
6 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	segment obtained successfully
RTEMS_INVALID_ ADDRESS	segment is NULL
RTEMS_INVALID_ ADDRESS	old_size is NULL
RTEMS_INVALID_ ID	invalid region id
RTEMS_INVALID_ ADDRESS	segment address not in region
RTEMS_ UNSATISFIED	unable to make segment larger

DESCRIPTION:

This directive is used to increase or decrease the size of a segment. When increasing the size of a segment, it is possible that there is not memory available contiguous to the segment. In this case, the request is unsatisfied.

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.

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 256) - Create a port
- *rtems_port_ident* (page 257) - Get ID of a port
- *rtems_port_delete* (page 258) - Delete a port
- *rtems_port_external_to_internal* (page 259) - Convert external to internal address
- *rtems_port_internal_to_external* (page 260) - Convert internal to external address

19.2 Background

A dual-ported memory area (DPMA) is a 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

directive. When a port is deleted, its control block is returned to the DPCB free list.

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`

19.4 Directives

This section details the dual-ported memory 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.

19.4.1 PORT_CREATE - Create a port

CALLING SEQUENCE:

```

1 rtems_status_code rtems_port_create(
2   rtems_name  name,
3   void        *internal_start,
4   void        *external_start,
5   uint32_t    length,
6   rtems_id    *id
7 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	port created successfully
RTEMS_ INVALID_NAME	invalid port name
RTEMS_ INVALID_ ADDRESS	address not on four byte boundary
RTEMS_ INVALID_ ADDRESS	id is NULL
RTEMS_TOO_ MANY	too many DP memory areas created

DESCRIPTION:

This directive creates a port which resides on the local node for the specified DPMA. The assigned port id is returned in id. This port id is used as an argument to other dual-ported memory manager directives to convert addresses within this DPMA.

For control and maintenance of the port, RTEMS allocates and initializes an DPCB from the DPCB free pool. Thus memory from the dual-ported memory area is not used to store the DPCB.

NOTES:

The internal_address and external_address parameters must be on a four byte boundary.

This directive will not cause the calling task to be preempted.

19.4.2 PORT_IDENT - Get ID of a port

CALLING SEQUENCE:

```

1 rtems_status_code rtems_port_ident(
2     rtems_name name,
3     rtems_id *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	port identified successfully
RTEMS_INVALID_ADDRESS	id is NULL
RTEMS_INVALID_NAME	port name not found

DESCRIPTION:

This directive obtains the port id associated with the specified name to be acquired. If the port name is not unique, then the port id will match one of the DPMAs with that name. However, this port id is not guaranteed to correspond to the desired DPMA. The port id is used to access this DPMA in other dual-ported memory area related directives.

NOTES:

This directive will not cause the running task to be preempted.

19.4.3 PORT_DELETE - Delete a port

CALLING SEQUENCE:

```

1 rtems_status_code rtems_port_delete(
2     rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_	port deleted
SUCCESSFUL	successfully
RTEMS_INVALID_	invalid port id
ID	

DESCRIPTION:

This directive deletes the dual-ported memory area specified by `id`. The DPCB for the deleted dual-ported memory area is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the port. Any local task that knows the port id can delete the port.

19.4.4 PORT_EXTERNAL_TO_INTERNAL

- Convert external to internal address

CALLING SEQUENCE:

```

1 rtems_status_code rtems_port_external_to_
  ↪internal(
2   rtems_id id,
3   void *external,
4   void **internal
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_INVALID_ADDRESS	internal is NULL
RTEMS_SUCCESSFUL	successful conversion

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.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

19.4.5 PORT_INTERNAL_TO_EXTERNAL

- Convert internal to external address

CALLING SEQUENCE:

```

1 rtems_status_code rtems_port_internal_to_
  ↪external(
2   rtems_id id,
3   void *internal,
4   void **external
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_INVALID_ADDRESS	external is NULL
RTEMS_SUCCESSFUL	successful conversion

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.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

I/O MANAGER

20.1 Introduction

The input/output interface 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_initialize* (page 269) - Initialize a device driver
- *rtems_io_register_driver* (page 267) - Register a device driver
- *rtems_io_unregister_driver* (page 268) - Unregister a device driver
- *rtems_io_register_name* (page 270) - Register a device name
- *rtems_io_lookup_name* (page 271) - Look up a device name
- *rtems_io_open* (page 272) - Open a device
- *rtems_io_close* (page 273) - Close a device
- *rtems_io_read* (page 274) - Read from a device
- *rtems_io_write* (page 275) - Write to a device
- *rtems_io_control* (page 276) - Special device services

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 whether 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.

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.

20.2.6 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:

```

1 rtems_device_driver io_entry(
2     rtems_device_major_number major,
3     rtems_device_minor_number minor,
4     void *argument_block
5 );

```

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.

20.3 Operations

20.3.1 Register and Lookup Name

The `rtcms_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 `rtcms_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 `rtcms_io_initialize`, `rtcms_io_open`, `rtcms_io_close`, `rtcms_io_read`, `rtcms_io_write`, and `rtcms_io_control` and the underlying device driver entry points.

20.4 Directives

This section details the I/O 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.

20.4.1 IO_REGISTER_DRIVER - Register a device driver

in use.

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_io_register_
   ↪ driver(
2     rtems_device_major_number  major,
3     rtems_driver_address_table *driver_
   ↪ table,
4     rtems_device_major_number *registered_
   ↪ major
5 );

```

NOTES:

The Device Driver Table size is specified in the Configuration Table configuration. This needs to be set to maximum size the application requires.

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully registered
RTEMS_INVALID_ADDRESS	invalid registered major pointer
RTEMS_INVALID_ADDRESS	invalid driver table
RTEMS_INVALID_NUMBER	invalid major device number
RTEMS_TOO_MANY	no available major device table slot
RTEMS_RESOURCE_IN_USE	major device number entry in use

DESCRIPTION:

This directive attempts to add a new device driver to the Device Driver Table. The user can specify a specific major device number via the directive's `major` parameter, or let the registration routine find the next available major device number by specifying a major number of 0. The selected major device number is returned via the `registered_major` directive parameter. The directive automatically allocation major device numbers from the highest value down.

This directive automatically invokes the `IO_INITIALIZE` directive if the driver address table has an initialization and open entry.

The directive returns `RTEMS_TOO_MANY` if Device Driver Table is full, and `RTEMS_RESOURCE_IN_USE` if a specific major device number is requested and it is already

20.4.2 IO_UNREGISTER_DRIVER - Unregister a device driver

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_io_unregister_
   ↪ driver(
2   rtems_device_major_number    major
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully registered
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive removes a device driver from the Device Driver Table.

NOTES:

Currently no specific checks are made and the driver is not closed.

20.4.3 IO_INITIALIZE - Initialize a device driver

CALLING SEQUENCE:

```

1 rtems_status_code rtems_io_initialize(
2     rtems_device_major_number major,
3     rtems_device_minor_number minor,
4     void *argument
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver initialization routine specified in the Device Driver Table for this major number. This directive is automatically invoked for each device driver when multitasking is initiated via the `initialize_executive` directive.

A device driver initialization module 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.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being initialized.

20.4.4 IO_REGISTER_NAME - Register a device

CALLING SEQUENCE:

```
1 rtems_status_code rtems_io_register_name(  
2     const char          *name,  
3     rtems_device_major_number major,  
4     rtems_device_minor_number minor  
5 );
```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	successfully initialized
RTEMS_TOO_ MANY	too many devices registered

DESCRIPTION:

This directive associates name with the specified major/minor number pair.

NOTES:

This directive will not cause the calling task to be preempted.

20.4.5 IO_LOOKUP_NAME - Lookup a device

CALLING SEQUENCE:

```
1 rtems_status_code rtems_io_lookup_name(  
2     const char      *name,  
3     rtems_driver_name_t *device_info  
4 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_UNSATISFIED	name not registered

DESCRIPTION:

This directive returns the major/minor number pair associated with the given device name in `device_info`.

NOTES:

This directive will not cause the calling task to be preempted.

20.4.6 IO_OPEN - Open a device

CALLING SEQUENCE:

```
1 rtems_status_code rtems_io_open(  
2     rtems_device_major_number major,  
3     rtems_device_minor_number minor,  
4     void *argument  
5 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver open routine specified in the Device Driver Table for this major number. The open entry point is commonly used by device drivers to provide exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

20.4.7 IO_CLOSE - Close a device

CALLING SEQUENCE:

```
1 rtems_status_code rtems_io_close(  
2     rtems_device_major_number major,  
3     rtems_device_minor_number minor,  
4     void *argument  
5 );
```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	successfully initialized
RTEMS_INVALID_ NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver close routine specified in the Device Driver Table for this major number. The close entry point is commonly used by device drivers to relinquish exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

20.4.8 IO_READ - Read from a device

CALLING SEQUENCE:

```

1 rtems_status_code rtems_io_read(
2     rtems_device_major_number major,
3     rtems_device_minor_number minor,
4     void *argument
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver read routine specified in the Device Driver Table for this major number. 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.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

20.4.9 IO_WRITE - Write to a device

CALLING SEQUENCE:

```
1 rtems_status_code rtems_io_write(  
2     rtems_device_major_number major,  
3     rtems_device_minor_number minor,  
4     void *argument  
5 );
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver write routine specified in the Device Driver Table for this major number. 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.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

20.4.10 IO_CONTROL - Special device services

CALLING SEQUENCE:

```

1 rtems_status_code rtems_io_control(
2   rtems_device_major_number major,
3   rtems_device_minor_number minor,
4   void *argument
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successfully initialized
RTEMS_INVALID_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver I/O control routine specified in the Device Driver Table for this major number. 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 rate of a serial line, while an I/O control operation for a floppy disk driver may cause a seek operation.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

FATAL ERROR MANAGER

21.1 Introduction

The fatal error manager processes all fatal or irrecoverable errors and other sources of system termination (for example after `exit()`). The directives provided by the fatal error manager are:

- *rtems_fatal_error_occurred* (page 282) - Invoke the fatal error handler
- *rtems_fatal* (page 283) - Invoke the fatal error handler with error source
- *rtems_exception_frame_print* (page 284) - Print the CPU exception frame
- *rtems_fatal_source_text* (page 285) - Return the fatal source text
- *rtems_internal_error_text* (page 286) - Return the error code text

21.2 Background

The fatal error manager is called upon detection of an irrecoverable error condition by either RTEMS or the application software. Fatal errors can be detected from three sources:

- the executive (RTEMS)
- user system code
- user application code

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 static or dynamic user extension set may include a fatal error handler. The fatal error handler in the static extension set 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. If no user handlers are configured or if all the user handler return control to the fatal error manager, then the RTEMS default fatal error handler is invoked. If the default fatal error handler is invoked, then the system state is marked as failed.

Although the precise behavior of the default fatal error handler is processor specific, in general, it will disable all maskable interrupts, place the error code in a known processor dependent place (generally either on the stack or in a register), and halt the processor. The precise actions of the RTEMS fatal error are discussed in the Default Fatal Error Processing chapter of the Applications Supplement document for a specific target processor.

21.3 Operations

21.3.1 Announcing a Fatal Error

The `rtems_fatal_error_occurred` directive is invoked when a fatal error is detected. Before invoking any user-supplied fatal error handlers or the RTEMS fatal error handler, the `rtems_fatal_error_occurred` directive stores useful information in the variable `_Internal_errors_What_happened`. This structure contains three pieces of information:

- the source of the error (API or executive core),
- whether the error was generated internally by the executive, and a
- a numeric code to indicate the error type.

The error type indicator is dependent on the source of the error and whether or not the error was internally generated by the executive. If the error was generated from an API, then the error code will be of that API's error or status codes. The status codes for the RTEMS API are in `cpukit/rtems/include/rtems/rtems/status.h`. Those for the POSIX API can be found in `<errno.h>`.

The `rtems_fatal_error_occurred` directive is responsible for invoking an optional user-supplied fatal error handler and/or the RTEMS fatal error handler. All fatal error handlers are passed an error code to describe the error detected.

Occasionally, an application requires more sophisticated fatal error processing such as passing control to a debugger. For these cases, a user-supplied fatal error handler can be specified in the RTEMS configuration table. The User Extension Table field `fatal` contains the address of the fatal error handler to be executed when the `rtems_fatal_error_occurred` directive is called. If the field is set to `NULL` or if the configured fatal error handler returns to the executive, then the default handler provided by RTEMS is executed. This default handler will halt execution on the processor where the error occurred.

21.4 Directives

This section details the fatal error 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.

21.4.1 FATAL_ERROR_OCCURRED - Invoke the fatal error handler

CALLING SEQUENCE:

```
1 void rtems_fatal_error_occurred(  
2     uint32_t the_error  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive processes fatal errors. If the FATAL error extension is defined in the configuration table, then the user-defined error extension is called. If configured and the provided FATAL error extension returns, then the RTEMS default error handler is invoked. This directive can be invoked by RTEMS or by the user's application code including initialization tasks, other tasks, and ISRs.

NOTES:

This directive supports local operations only.

Unless the user-defined error extension takes special actions such as restarting the calling task, this directive WILL NOT RETURN to the caller.

The user-defined extension for this directive may wish to initiate a global shutdown.

21.4.2 FATAL - Invoke the fatal error handler with error source

CALLING SEQUENCE:

```
1 void rtems_fatal(  
2     rtems_fatal_source source,  
3     rtems_fatal_code  error  
4 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive invokes the internal error handler with is internal set to false. See also `rtems_fatal_error_occurred`.

21.4.3 EXCEPTION_FRAME_PRINT -

Prints the exception frame

CALLING SEQUENCE:

```
1 void rtems_exception_frame_print(  
2     const rtems_exception_frame *frame  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

Prints the exception frame via `printk()`.

21.4.4 FATAL_SOURCE_TEXT - Returns a text for a fatal source

CALLING SEQUENCE:

```
1 const char *rtems_fatal_source_text(  
2     rtems_fatal_source source  
3 );
```

DIRECTIVE STATUS CODES:

The fatal source text or "?" in case the passed fatal source is invalid.

DESCRIPTION:

Returns a text for a fatal source. The text for fatal source is the enumerator constant.

21.4.5 INTERNAL_ERROR_TEXT - Returns a text for an internal error code

CALLING SEQUENCE:

```
1 const char *rtcms_internal_error_text(  
2     rtcms_fatal_code error  
3 );
```

DIRECTIVE STATUS CODES:

The error code text or "?" in case the passed error code is invalid.

DESCRIPTION:

Returns a text for an internal error code. The text for each internal error code is the enumerator constant.

BOARD SUPPORT PACKAGES

22.1 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.

22.2 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.

22.2.1 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 configured minimum stack size.

22.2.2 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.

22.2.3 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.

22.3 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.

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.

22.3.1 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 time-slicing, 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.

22.4 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 *Chapter 23 - User Extensions Manager* (page 295).

22.5 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.

22.5.1 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 inter-processor 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.

22.5.2 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.

22.5.3 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.

22.5.4 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.

USER EXTENSIONS MANAGER

23.1 Introduction

The RTEMS 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 304) - Create an extension set
- *rtems_extension_ident* (page 305) - Get ID of an extension set
- *rtems_extension_delete* (page 306) - Delete an extension set

23.2 Background

User extension routines are invoked 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

These extensions are invoked as a function with arguments that are appropriate to the system event.

23.2.1 Extension Sets

An extension set is defined as a set of routines which are invoked at each of the critical system events at which user extension routines are invoked. Together a set of these routines typically perform a specific functionality such as performance monitoring or debugger support. RTEMS is informed of the entry points which constitute an extension set via the following structure:.. index:: rtems_extensions_table

```

1 typedef struct {
2     rtems_task_create_extension    thread_
3     ↪create;
4     rtems_task_start_extension    thread_
5     ↪start;
6     rtems_task_restart_extension  thread_
7     ↪restart;
8     rtems_task_delete_extension   thread_
9     ↪delete;
10    rtems_task_switch_extension    thread_
    ↪switch;
    rtems_task_begin_extension     thread_
    ↪begin;
    rtems_task_exitted_extension   thread_
    ↪exitted;
    rtems_fatal_extension          fatal;
} rtems_extensions_table;

```

RTEMS allows the user to have multiple extension sets active at the same time. First, a single static extension set may be defined as the application's User Extension Table which is included as part of the Configuration Table. This extension set is active for the entire life of the system and may not be deleted. This extension set is especially important because it is the only way the application can provided a FATAL error extension which is invoked if RTEMS fails during the initialize_executive directive. The static extension set is optional and may be configured as NULL if no static extension set is required.

Second, the user can install dynamic extensions using the rtems_extension_create directive. These extensions are RTEMS objects in that they have a name, an ID, and can be dynamically created and deleted. In contrast to the static extension set, these extensions can only be created and installed after the initialize_executive directive successfully completes execution. Dynamic extensions are useful for encapsulating the functionality of an extension set. For example, the application could use extensions to manage a special coprocessor, do performance monitoring, and to do stack bounds checking. Each of these extension sets could be written and installed independently of the others.

All user extensions are optional and RTEMS places no naming restrictions on the user. The user extension entry points are copied into an internal RTEMS structure. This means the user does not need to keep the table after creating it, and changing the handler entry points dynamically in a table once created has no effect. Creating a table local to a function can save space in space limited applications.

Extension switches do not effect the context switch overhead if no switch handler is installed.

23.2.2 TCB Extension Area

RTEMS provides for a pointer to a user-defined data area for each extension set to be linked to each task's control block. This set of pointers is an extension of the TCB and can be used

to store additional data required by the user's extension functions.

The TCB extension is an array of pointers in the TCB. The index into the table can be obtained from the extension id returned when the extension is created:

```
1 index = rtems_object_id_get_index(extension_
  ↪ id);
```

The number of pointers in the area is the same as the number of 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 TASK_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 TASK_CREATE or TASK_START extension. The application is responsible for managing this extended memory area for the TCBs. The memory may be reinitialized by the TASK_RESTART extension and should be deallocated by the TASK_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.

23.2.3 Extensions

The sections that follow will contain a description of each extension. Each section will contain a prototype of a function with the appropriate calling sequence for the corresponding extension. The names given for the C function and its arguments are all defined by the user. The names used in the examples were arbitrary

chosen and impose no naming conventions on the user.

23.2.3.1 TASK_CREATE Extension

The TASK_CREATE extension directly corresponds to the `rtems_task_create` directive. If this extension is defined in any static or dynamic extension set and a task is being created, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
1 bool user_task_create(
  rtems_tcb *current_task,
  rtems_tcb *new_task
);
```

where `current_task` can be used to access the TCB for the currently executing task, and `new_task` can be used to access the TCB for the new task being created. This extension is invoked from the `rtems_task_create` directive after `new_task` has been completely initialized, but before it is placed on a ready TCB chain.

The user extension is expected to return the boolean value `true` if it successfully executed and `false` otherwise. A task create user extension will frequently attempt to allocate resources. If this allocation fails, then the extension should return `false` and the entire task create operation will fail.

23.2.3.2 TASK_START Extension

The TASK_START extension directly corresponds to the `task_start` directive. If this extension is defined in any static or dynamic extension set and a task is being started, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
1 void user_task_start(
  rtems_tcb *current_task,
  rtems_tcb *started_task
);
```

where `current_task` can be used to access the TCB for the currently executing task, and `started_task` can be used to access the TCB for

the dormant task being started. This extension is invoked from the `task_start` directive after `started_task` has been made ready to start execution, but before it is placed on a ready TCB chain.

23.2.3.3 TASK_RESTART Extension

The `TASK_RESTART` extension directly corresponds to the `task_restart` directive. If this extension is defined in any static or dynamic extension set and a task is being restarted, then the extension should have a prototype similar to the following:

```
1 void user_task_restart(  
2     rtems_tcb *current_task,  
3     rtems_tcb *restarted_task  
4 );
```

where `current_task` can be used to access the TCB for the currently executing task, and `restarted_task` can be used to access the TCB for the task being restarted. This extension is invoked from the `task_restart` directive after `restarted_task` has been made ready to start execution, but before it is placed on a ready TCB chain.

23.2.3.4 TASK_DELETE Extension

The `TASK_DELETE` extension is associated with the `task_delete` directive. If this extension is defined in any static or dynamic extension set and a task is being deleted, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
1 void user_task_delete(  
2     rtems_tcb *current_task,  
3     rtems_tcb *deleted_task  
4 );
```

where `current_task` can be used to access the TCB for the currently executing task, and `deleted_task` can be used to access the TCB for the task being deleted. This extension is invoked from the `task_delete` directive after the TCB has been removed from a ready TCB chain, but before all its resources including

the TCB have been returned to their respective free pools. This extension should not call any RTEMS directives if a task is deleting itself (`current_task` is equal to `deleted_task`).

23.2.3.5 TASK_SWITCH Extension

The `TASK_SWITCH` extension corresponds to a task context switch. If this extension is defined in any static or dynamic extension set and a task context switch is in progress, then the extension routine will automatically be invoked by RTEMS. The extension should have a prototype similar to the following:

```
1 void user_task_switch(  
2     rtems_tcb *current_task,  
3     rtems_tcb *heir_task  
4 );
```

where `current_task` can be used to access the TCB for the task that is being swapped out, and `heir_task` can be used to access the TCB for the task being swapped in. This extension is invoked from RTEMS' dispatcher routine after the `current_task` context has been saved, but before the `heir_task` context has been restored. This extension should not call any RTEMS directives.

23.2.3.6 TASK_BEGIN Extension

The `TASK_BEGIN` extension is invoked when a task begins execution. It is invoked immediately before the body of the starting procedure and executes in the context in the task. This user extension have a prototype similar to the following:

```
1 void user_task_begin(  
2     rtems_tcb *current_task  
3 );
```

where `current_task` can be used to access the TCB for the currently executing task which has begun. The distinction between the `TASK_BEGIN` and `TASK_START` extension is that the `TASK_BEGIN` extension is executed in the context of the actual task while the `TASK_START` extension is executed in the context of the task performing the `task_start` direc-

tive. For most extensions, this is not a critical distinction.

23.2.3.7 TASK_EXITTED Extension

The TASK_EXITTED extension is invoked when a task exits the body of the starting procedure by either an implicit or explicit return statement. This user extension have a prototype similar to the following:

```
1 void user_task_exitted(  
2     rtems_tcb *current_task  
3 );
```

where `current_task` can be used to access the TCB for the currently executing task which has just exited.

Although exiting of task is often considered to be a fatal error, this extension allows recovery by either restarting or deleting the exiting task. If the user does not wish to recover, then a fatal error may be reported. If the user does not provide a TASK_EXITTED extension or the provided handler returns control to RTEMS, then the RTEMS default handler will be used. This default handler invokes the directive `fatal_error_occurred` with the RTEMS_TASK_EXITTED directive status.

23.2.3.8 FATAL Error Extension

The FATAL error extension is associated with the `fatal_error_occurred` directive. If this extension is defined in any static or dynamic extension set and the `fatal_error_occurred` directive has been invoked, then this extension will be called. This extension should have a prototype similar to the following:

```
1 void user_fatal_error(  
2     Internal_errors_Source the_source,  
3     bool                  is_internal,  
4     uint32_t              the_error  
5 );
```

where `the_error` is the error code passed to the `fatal_error_occurred` directive. This extension is invoked from the `fatal_error_occurred` directive.

If defined, the user's FATAL error extension is invoked before RTEMS' default fatal error routine is invoked and the processor is stopped. For example, this extension could be used to pass control to a debugger when a fatal error occurs. This extension should not call any RTEMS directives.

23.2.4 Order of Invocation

When one of the critical system events occur, the user extensions are invoked in either "forward" or "reverse" order. Forward order indicates that the static extension set is invoked followed by the dynamic extension sets in the order in which they were created. Reverse order means that the dynamic extension sets are invoked in the opposite of the order in which they were created followed by the static extension set. By invoking the extension sets in this order, extensions can be built upon one another. At the following system events, the extensions are invoked in forward order:

1. Task creation
2. Task initiation
3. Task reinitiation
4. Task deletion
5. Task context switch
6. Post task context switch
7. Task begins to execute

At the following system events, the extensions are invoked in reverse order:

1. Task deletion
2. Fatal error detection

At these system events, the extensions are invoked in reverse order to insure that if an extension set is built upon another, the more complicated extension is invoked before the extension set it is built upon. For example, by invoking the static extension set last it is known that the "system" fatal error extension will be the last fatal error extension executed. Another example is use of the task 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 TASK_DELETE extension is invoked after that of the other extensions.

23.3 Operations

23.3.1 Creating an Extension Set

The `rtems_extension_create` directive creates and installs an extension set by allocating a Extension Set Control Block (ESCB), assigning the extension set a user-specified name, and assigning it an extension set ID. Newly created extension sets are immediately installed and are invoked upon the next system even supporting an extension.

23.3.2 Obtaining Extension Set IDs

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

23.3.3 Deleting an Extension Set

The `rtems_extension_delete` directive is used to delete an extension set. The extension set's control block is returned to the ESCB free list when it is deleted. An extension set can be deleted by a task other than the task which created the extension set. Any subsequent references to the extension's name and ID are invalid.

23.4 Directives

This section details the user extension 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.

23.4.1 EXTENSION_CREATE - Create a extension set

CALLING SEQUENCE:

```

1 rtems_status_code rtems_extension_create(
2     rtems_name      name,
3     rtems_extensions_table *table,
4     rtems_id        *id
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	extension set created successfully
RTEMS_INVALID_NAME	invalid extension set name
RTEMS_TOO_MANY	too many extension sets created

DESCRIPTION:

This directive creates a extension set. The assigned extension set id is returned in id. This id is used to access the extension set with other user extension manager directives. For control and maintenance of the extension set, RTEMS allocates an ESCB from the local ESCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

23.4.2 EXTENSION_IDENT - Get ID of a extension set

CALLING SEQUENCE:

```
1 rtems_status_code rtems_extension_ident(  
2     rtems_name name,  
3     rtems_id *id  
4 );
```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	extension set identified successfully
RTEMS_ INVALID_NAME	extension set name not found

DESCRIPTION:

This directive obtains the extension set id associated with the extension set name to be acquired. If the extension set name is not unique, then the extension set id will match one of the extension sets with that name. However, this extension set id is not guaranteed to correspond to the desired extension set. The extension set id is used to access this extension set in other extension set related directives.

NOTES:

This directive will not cause the running task to be preempted.

23.4.3 EXTENSION_DELETE - Delete a extension set

CALLING SEQUENCE:

```

1 rtems_status_code rtems_extension_delete(
2   rtems_id id
3 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	extension set deleted successfully
RTEMS_INVALID_ID	invalid extension set id

DESCRIPTION:

This directive deletes the extension set specified by `id`. If the extension set is running, it is automatically canceled. The ESCB for the deleted extension set is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A extension set can be deleted by a task other than the task which created the extension set.

NOTES:

This directive will not cause the running task to be preempted.

CONFIGURING A SYSTEM

24.1 Introduction

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 should only rarely to 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.

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 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.

24.2 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.

24.3 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` macro to add a value to the estimate. See *Chapter 24 Section 15.1 - Specify Memory Overhead* (page 334) 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` and `CONFIGURE_MAXIMUM_TASKS` 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 *Chapter 24 Section 7 - Unlimited Objects* (page 315) 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.

24.4 Potential Issues with RTEMS Workspace Size Estimation

The `<rtcms/confdefs.h>` file estimates the amount of memory required for the RTEMS Workspace. This estimate is only as accurate as the information given to `<rtcms/confdefs.h>` and may be either too high or too low for a variety of reasons. Some of the reasons that `<rtcms/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, `<rtcms/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.

24.5 Format to be followed for making changes in this file

MACRO NAME:

Should be alphanumeric. Can have ‘_’ (underscore).

DATA TYPE:

Please refer to all existing formats.

RANGE:

The range depends on the Data Type of the macro.

- If the data type is of type task priority, then its value should be an integer in the range of 1 to 255.
- If the data type is an integer, then it can have numbers, characters (in case the value is defined using another macro) and arithmetic operations (+, -, *, /).
- If the data type is a function pointer the first character should be an alphabet or an underscore. The rest of the string can be alphanumeric.
- If the data type is RTEMS Attributes or RTEMS Mode then the string should be alphanumeric.
- If the data type is RTEMS NAME then the value should be an integer ≥ 0 or `RTEMS_BUILD_NAME('U', 'I', '1', ' ')`

DEFAULT VALUE:

The default value should be in the following formats- Please note that the ‘.’ (full stop) is necessary.

- In case the value is not defined then: This is not defined by default.
- If we know the default value then: The default value is XXX.
- If the default value is BSP Specific then: This option is BSP specific.

DESCRIPTION:

The description of the macro. (No specific format)

NOTES:

Any further notes. (No specific format)

24.6 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:

```

1 #include <bsp.h>
2 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_
   ↪DRIVER
3 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_
   ↪DRIVER
4 #define CONFIGURE_MICROSECONDS_PER_TICK    ↪
   ↪1000 /* 1 millisecond */
5 #define CONFIGURE_TICKS_PER_TIMESLICE      ↪
   ↪50 /* 50 milliseconds */
6 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
7 #define CONFIGURE_MAXIMUM_TASKS 4
8 #define CONFIGURE_MAXIMUM_MESSAGE_QUEUES 1
9 #define CONFIGURE_MESSAGE_BUFFER_MEMORY \
10     CONFIGURE_MESSAGE_BUFFERS_FOR_
   ↪QUEUE(20, sizeof(struct USER_MESSAGE))
11 #define CONFIGURE_INIT
12 #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` 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`, 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` and `CONFIGURE_TICKS_PER_TIMESLICE`, 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`, 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` 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` that the application requires a maximum of four (4) simultaneously existing Classic API tasks. Similarly, by specifying `CONFIGURE_MAXIMUM_MESSAGE_QUEUES`, 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`. 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` 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.

24.7 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 *Chapter 24 Section 3 - Sizing the RTEMS Workspace* (page 310) 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 API-independent object classes can be configured in unlimited mode:

- POSIX Keys
- POSIX Key Value Pairs

The following object classes in the Classic API can be configured in unlimited mode:

- Tasks
- Timers
- Semaphores
- Message Queues
- Periods
- Barriers
- Partitions
- Regions
- Ports

Additionally, the following object classes from the POSIX API can be configured in unlimited mode:

- Threads
- Mutexes
- Condition Variables
- Timers
- Message Queues
- Message Queue Descriptors

- Semaphores
- Barriers
- Read/Write Locks
- Spinlocks

The following object classes can *not* be configured in unlimited mode:

- Drivers
- File Descriptors
- User Extensions
- POSIX Queued Signals

Due to the memory requirements of unlimited objects it is strongly recommended to use them only in combination with the unified work areas. See *Chapter 24 Section 12.1 - Separate or Unified Work Areas* (page 328) 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 separate configuration defines for supported object classes:

```

1 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_
   ↳DRIVER
2 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_
   ↳DRIVER
3 #define CONFIGURE_UNIFIED_WORK_AREAS
4 #define CONFIGURE_UNLIMITED_OBJECTS
5 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
6 #define CONFIGURE_INIT
7 #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.

24.7.1 Per Object Class Unlimited Object Instances

When the number of objects is not known ahead of time, RTEMS provides an auto-extending mode that can be enabled individ-

ually for each object type by using the macro `rtems_resource_unlimited`. 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`:

```
1 #define CONFIGURE_MAXIMUM_TASKS rtems_
   ↪resource_unlimited(5)
```

Object maximum specifications can be evaluated with the `rtems_resource_is_unlimited` and “`rtems_resource_maximum_per_allocation`” macros.

24.7.2 Unlimited Object Instances

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 and specifying the allocation size.

24.7.3 Enable Unlimited Object Instances

CONSTANT:

`CONFIGURE_UNLIMITED_OBJECTS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_UNLIMITED_OBJECTS` enables `rtems_resource_unlimited` mode for Classic API and POSIX API objects that do not

already have a specific maximum limit defined.

NOTES:

When using unlimited objects, it is common practice to also specify `CONFIGURE_UNIFIED_WORK_AREAS` so the system operates with a single pool of memory for both RTEMS and application memory allocations.

24.7.4 Specify Unlimited Objects Allocation Size

CONSTANT:

`CONFIGURE_UNLIMITED_ALLOCATION_SIZE`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

If not defined and `CONFIGURE_UNLIMITED_OBJECTS` is defined, the default value is eight (8).

DESCRIPTION:

`CONFIGURE_UNLIMITED_ALLOCATION_SIZE` provides an allocation size to use for `rtems_resource_unlimited` when using `CONFIGURE_UNLIMITED_OBJECTS`.

NOTES:

By allowing users to declare all resources as being unlimited the user can avoid identifying and limiting the resources used. `CONFIGURE_UNLIMITED_OBJECTS` does not support varying the allocation sizes for different objects; users who want that much control can define the `rtems_resource_unlimited` macros themselves.

```
1 #define CONFIGURE_UNLIMITED_OBJECTS
2 #define CONFIGURE_UNLIMITED_ALLOCATION_SIZE_
   ↪5
```


24.8 Classic API Configuration

This section defines the Classic API related system configuration parameters supported by `<rtems/confdefs.h>`.

24.8.1 Specify Maximum Classic API Tasks

CONSTANT:

`CONFIGURE_MAXIMUM_TASKS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_TASKS` is the maximum number of Classic API Tasks that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

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 parameter `CONFIGURE_EXTRA_TASK_STACKS` is used to specify task stack requirements ABOVE the minimum size required. See *Chapter 24 Section 12.7 - Reserve Task/Thread Stack Memory Above Minimum* (page 330) for more information about `CONFIGURE_EXTRA_TASK_STACKS`.

The maximum number of POSIX threads is specified by `CONFIGURE_MAXIMUM_POSIX_THREADS`.

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.

24.8.2 Specify Maximum Classic API Timers

CONSTANT:

`CONFIGURE_ENABLE_CLASSIC_API_NOTEPADS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, and Classic API Notepads are not supported.

DESCRIPTION:

`CONFIGURE_ENABLE_CLASSIC_API_NOTEPADS` should be defined if the user wants to have support for Classic API Notepads in their application.

NOTES:

Disabling Classic API Notepads saves the allocation of sixteen (16) thirty-two bit integers. This saves sixty-four bytes per task/thread plus the allocation overhead. Notepads are rarely used in applications and this can save significant memory in a low RAM system. Classic API Notepads are deprecated, and this option has been removed from post 4.11 versions of RTEMS.

24.8.3 Specify Maximum Classic API Timers

CONSTANT:

`CONFIGURE_MAXIMUM_TIMERS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_TIMERS` is the maximum number of Classic API Timers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.4 Specify Maximum Classic API Semaphores

CONSTANT:

CONFIGURE_MAXIMUM_SEMAPHORES

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_SEMAPHORES is the maximum number of Classic API Semaphores that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.5 Specify Maximum Classic API Semaphores usable with MrsP

CONSTANT:

CONFIGURE_MAXIMUM_MRSP_SEMAPHORES

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_MRSP_SEMAPHORES is the maximum number of Classic API Semaphores using the Multiprocessor Resource Sharing Protocol (MrsP) that can be concurrently active.

NOTES:

This configuration option is only used on SMP configurations. On uni-processor configurations the Priority Ceiling Protocol is used for MrsP semaphores and thus no extra memory is necessary.

24.8.6 Specify Maximum Classic API Message Queues

CONSTANT:

CONFIGURE_MAXIMUM_MESSAGE_QUEUES

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_MESSAGE_QUEUES is the maximum number of Classic API Message Queues that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.7 Specify Maximum Classic API Barriers

CONSTANT:

CONFIGURE_MAXIMUM_BARRIERS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_BARRIERS is the maximum number of Classic API Barriers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.8 Specify Maximum Classic API Periods

CONSTANT:

CONFIGURE_MAXIMUM_PERIODS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_PERIODS is the maximum number of Classic API Periods that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.9 Specify Maximum Classic API Partitions

CONSTANT:

CONFIGURE_MAXIMUM_PARTITIONS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_PARTITIONS is the maximum number of Classic API Partitions that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.10 Specify Maximum Classic API Regions

CONSTANT:

CONFIGURE_MAXIMUM_REGIONS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_REGIONS is the maximum number of Classic API Regions that can be concurrently active.

NOTES:

None.

24.8.11 Specify Maximum Classic API Ports

CONSTANT:

CONFIGURE_MAXIMUM_PORTS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_PORTS is the maximum number of Classic API Ports that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.8.12 Specify Maximum Classic API User Extensions

CONSTANT:

CONFIGURE_MAXIMUM_USER_EXTENSIONS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_USER_EXTENSIONS is the maximum number of Classic API User Extensions that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.9 Classic API Initialization Tasks Table Configuration

The `<rtems/confdefs.h>` configuration system can automatically generate an Initialization Tasks Table named `Initialization_tasks` with a single entry. The following parameters control the generation of that table.

24.9.1 Instantiate Classic API Initialization Task Table

CONSTANT:

`CONFIGURE_RTEMS_INIT_TASKS_TABLE`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_RTEMS_INIT_TASKS_TABLE` is defined if the user wishes to use a Classic RTEMS API Initialization Task Table. The table built by `<rtems/confdefs.h>` specifies the parameters for a single task. This is sufficient for applications which initialize the system from a single task.

By default, this field is not defined as the user **MUST** select their own API for initialization tasks.

NOTES:

The application may choose to use the initialization tasks or threads table from another API.

A compile time error will be generated if the user does not configure any initialization tasks or threads.

24.9.2 Specifying Classic API Initialization Task Entry Point

CONSTANT:

`CONFIGURE_INIT_TASK_ENTRY_POINT`

DATA TYPE:

Task entry function pointer (`rtems_task_entry`).

RANGE:

Valid task entry function pointer.

DEFAULT VALUE:

The default value is `Init`.

DESCRIPTION:

`CONFIGURE_INIT_TASK_ENTRY_POINT` is the entry point (a.k.a. function name) of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

The user must implement the function `Init` or the function name provided in this configuration parameter.

24.9.3 Specifying Classic API Initialization Task Name

CONSTANT:

`CONFIGURE_INIT_TASK_NAME`

DATA TYPE:

RTEMS Name (`rtems_name`).

RANGE:

Any value.

DEFAULT VALUE:

The default value is `rtems_build_name('U','I','1',' ')`.

DESCRIPTION:

`CONFIGURE_INIT_TASK_NAME` is the name of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

None.

24.9.4 Specifying Classic API Initialization Task Stack Size

CONSTANT:

`CONFIGURE_INIT_TASK_STACK_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is RTEMS_MINIMUM_STACK_SIZE.

DESCRIPTION:

CONFIGURE_INIT_TASK_STACK_SIZE is the stack size of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

If the stack size specified is greater than the configured minimum, it must be accounted for in CONFIGURE_EXTRA_TASK_STACKS. See *Chapter 24 Section 12.7 - Reserve Task/Thread Stack Memory Above Minimum* (page 330) for more information about CONFIGURE_EXTRA_TASK_STACKS.

24.9.5 Specifying Classic API Initialization Task Priority

CONSTANT:

CONFIGURE_INIT_TASK_PRIORITY

DATA TYPE:

RTEMS Task Priority (rtems_task_priority).

RANGE:

One (1) to CONFIGURE_MAXIMUM_PRIORITY.

DEFAULT VALUE:

The default value is 1, which is the highest priority in the Classic API.

DESCRIPTION:

CONFIGURE_INIT_TASK_PRIORITY is the initial priority of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

None.

24.9.6 Specifying Classic API Initialization Task Attributes

CONSTANT:

CONFIGURE_INIT_TASK_ATTRIBUTES

DATA TYPE:

RTEMS Attributes (rtems_attribute).

RANGE:

Valid task attribute sets.

DEFAULT VALUE:

The default value is RTEMS_DEFAULT_ATTRIBUTES.

DESCRIPTION:

CONFIGURE_INIT_TASK_ATTRIBUTES is the task attributes of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

None.

24.9.7 Specifying Classic API Initialization Task Modes

CONSTANT:

CONFIGURE_INIT_TASK_INITIAL_MODES

DATA TYPE:

RTEMS Mode (rtems_mode).

RANGE:

Valid task mode sets.

DEFAULT VALUE:

The default value is RTEMS_NO_PREEMPT.

DESCRIPTION:

CONFIGURE_INIT_TASK_INITIAL_MODES is the initial execution mode of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

None.

24.9.8 Specifying Classic API Initialization Task Arguments

CONSTANT:

CONFIGURE_INIT_TASK_ARGUMENTS

DATA TYPE:

RTEMS Task Argument (rtems_task_argument).

RANGE:

Complete range of the type.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_INIT_TASK_ARGUMENTS is the task argument of the single initialization task defined by the Classic API Initialization Tasks Table.

NOTES:

None.

24.9.9 Not Using Generated Initialization Tasks Table

CONSTANT:

CONFIGURE_HAS_OWN_INIT_TASK_TABLE

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_HAS_OWN_INIT_TASK_TABLE is defined if the user wishes to define their own Classic API Initialization Tasks Table. This table should be named `Initialization_tasks`.

NOTES:

This is a seldom used configuration parameter. The most likely use case is when an application desires to have more than one initialization task.

24.10 POSIX API Configuration

The parameters in this section are used to configure resources for the RTEMS POSIX API. They are only relevant if the POSIX API is enabled at configure time using the `--enable-posix` option.

24.10.1 Specify Maximum POSIX API Threads

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_THREADS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_THREADS` is the maximum number of POSIX API Threads that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

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 parameter `CONFIGURE_EXTRA_TASK_STACKS` is used to specify thread stack requirements *ABOVE* the minimum size required. See *Chapter 24 Section 12.7 - Reserve Task/Thread Stack Memory Above Minimum* (page 330) for more information about `CONFIGURE_EXTRA_TASK_STACKS`.

The maximum number of Classic API Tasks is specified by `CONFIGURE_MAXIMUM_TASKS`.

All POSIX threads have floating point enabled.

24.10.2 Specify Maximum POSIX API Mutexes

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_MUTEXES`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_MUTEXES` is the maximum number of POSIX API Mutexes that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.3 Specify Maximum POSIX API Condition Variables

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES` is the maximum number of POSIX API Condition Variables that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.4 Specify Maximum POSIX API Keys

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_KEYS`

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_POSIX_KEYS is the maximum number of POSIX API Keys that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.5 Specify Maximum POSIX API Timers

CONSTANT:

CONFIGURE_MAXIMUM_POSIX_TIMERS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_POSIX_TIMERS is the maximum number of POSIX API Timers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.6 Specify Maximum POSIX API Queued Signals

CONSTANT:

CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS is the maximum number of POSIX API Queued Signals that can be concurrently active.

NOTES:

None.

24.10.7 Specify Maximum POSIX API Message Queues

CONSTANT:

CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES is the maximum number of POSIX API Message Queues that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.8 Specify Maximum POSIX API Message Queue Descriptors

CONSTANT:

CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTOR

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

greater than or equal to CONFIGURE_MAXIMUM_POSIX_MESSAGES_QUEUES

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTOR is the maximum number of POSIX API Message Queue Descriptors that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

`CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTOR` should be greater than or equal to `CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES`.

24.10.9 Specify Maximum POSIX API Semaphores

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_SEMAPHORES`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_SEMAPHORES` is the maximum number of POSIX API Semaphores that can be concurrently active.

NOTES:

None.

24.10.10 Specify Maximum POSIX API Barriers

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_BARRIERS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_BARRIERS` is the maximum number of POSIX API Barriers that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.11 Specify Maximum POSIX API Spinlocks

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_SPINLOCKS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_SPINLOCKS` is the maximum number of POSIX API Spinlocks that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.10.12 Specify Maximum POSIX API Read/Write Locks

CONSTANT:

`CONFIGURE_MAXIMUM_POSIX_RWLOCKS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_POSIX_RWLOCKS` is the maximum number of POSIX API Read/Write Locks that can be concurrently active.

NOTES:

This object class can be configured in unlimited allocation mode.

24.11 POSIX Initialization Threads Table Configuration

The `<rtems/confdefs.h>` configuration system can automatically generate a POSIX Initialization Threads Table named `POSIX_Initialization_threads` with a single entry. The following parameters control the generation of that table.

24.11.1 Instantiate POSIX API Initialization Thread Table

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_TABLE`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This field is not defined by default, as the user **MUST** select their own API for initialization tasks.

DESCRIPTION:

`CONFIGURE_POSIX_INIT_THREAD_TABLE` is defined if the user wishes to use a POSIX API Initialization Threads Table. The table built by `<rtems/confdefs.h>` specifies the parameters for a single thread. This is sufficient for applications which initialize the system from a single task.

By default, this field is not defined as the user **MUST** select their own API for initialization tasks.

NOTES:

The application may choose to use the initialization tasks or threads table from another API.

A compile time error will be generated if the user does not configure any initialization tasks or threads.

24.11.2 Specifying POSIX API Initialization Thread Entry Point

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT`

DATA TYPE:

POSIX thread function pointer (`void *(*entry_point)(void *)`).

RANGE:

Undefined or a valid POSIX thread function pointer.

DEFAULT VALUE:

The default value is `POSIX_Init`.

DESCRIPTION:

`CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT` is the entry point (a.k.a. function name) of the single initialization thread defined by the POSIX API Initialization Threads Table.

NOTES:

The user must implement the function `POSIX_Init` or the function name provided in this configuration parameter.

24.11.3 Specifying POSIX API Initialization Thread Stack Size

CONSTANT:

`CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is `2 * RTEMS_MINIMUM_STACK_SIZE`.

DESCRIPTION:

`CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE` is the stack size of the single initialization thread defined by the POSIX API Initialization Threads Table.

NOTES:

If the stack size specified is greater than the configured minimum, it must be accounted for in `CONFIGURE_EXTRA_TASK_STACKS`. See *Chapter 24 Section 12.7 - Reserve*

Task/Thread Stack Memory Above Minimum
(page 330) for more information about
CONFIGURE_EXTRA_TASK_STACKS.

24.11.4 Not Using Generated POSIX Initialization Threads Table

CONSTANT:

CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE
is defined if the user wishes to define their
own POSIX API Initialization Threads
Table. This table should be named
POSIX_Initialization_threads.

NOTES:

This is a seldom used configuration parameter. The most likely use case is when an application desires to have more than one initialization task.

24.12 Basic System Information

This section defines the general system configuration parameters supported by `<rtems/confdefs.h>`.

24.12.1 Separate or Unified Work Areas

CONSTANT:

`CONFIGURE_UNIFIED_WORK_AREAS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, which specifies that the C Program Heap and the RTEMS Workspace will be separate.

DESCRIPTION:

When defined, the C Program Heap and the RTEMS Workspace will be one pool of memory.

When not defined, there will be separate memory pools for the RTEMS Workspace and C Program Heap.

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 RTEMS “unlimited” objects option. You will be able to create objects until you run out of all available memory rather than just until you run out of RTEMS Workspace.

24.12.2 Length of Each Clock Tick

CONSTANT:

`CONFIGURE_MICROSECONDS_PER_TICK`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

This is not defined by default. When not defined, the clock tick quantum is configured to be 10,000 microseconds which is ten (10) milliseconds.

DESCRIPTION:

This constant is used to specify the length of time between clock ticks.

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 parameter has no impact if the Clock Tick Device driver is not configured.

There may be BSP specific limits on the resolution or maximum value of a clock tick quantum.

24.12.3 Specifying Timeslicing Quantum

CONSTANT:

`CONFIGURE_TICKS_PER_TIMESLICE`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 50.

DESCRIPTION:

This configuration parameter specifies the length of the timeslice quantum in ticks for each task.

NOTES:

This configuration parameter has no impact if the Clock Tick Device driver is not configured.

24.12.4 Specifying the Number of Thread Priority Levels

CONSTANT:

CONFIGURE_MAXIMUM_PRIORITY

DATA TYPE:

Unsigned integer (uint8_t).

RANGE:

Valid values for this configuration parameter must be one (1) less than a power of two (2) between 4 and 256 inclusively. In other words, valid values are 3, 7, 31, 63, 127, and 255.

DEFAULT VALUE:

The default value is 255, because RTEMS must support 256 priority levels to be compliant with various standards. These priorities range from zero (0) to 255.

DESCRIPTION:

This configuration parameter specified the maximum numeric priority of any task in the system and one less than the number of priority levels in the system.

Reducing the number of priorities in the system reduces the amount of memory allocated from the RTEMS Workspace.

NOTES:

The numerically greatest priority is the logically lowest priority in the system and will thus be used by the IDLE task.

Priority zero (0) is reserved for internal use by RTEMS and is not available to applications.

With some schedulers, reducing the number of priorities can reduce the amount of memory used by the scheduler. For example, the Deterministic Priority Scheduler (DPS) used

by default uses three pointers of storage per priority level. Reducing the number of priorities from 256 levels to sixteen (16) can reduce memory usage by about three (3) kilobytes.

24.12.5 Specifying the Minimum Task Size

CONSTANT:

CONFIGURE_MINIMUM_TASK_STACK_SIZE

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Positive.

DEFAULT VALUE:

This is not defined by default, which sets the executive to the recommended minimum stack size for this processor.

DESCRIPTION:

The configuration parameter is set to the number of bytes the application wants the minimum stack size to be for every task or thread in the system.

Adjusting this parameter should be done with caution. Examining the actual usage using the Stack Checker Usage Reporting facility is recommended.

NOTES:

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 must 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.

24.12.6 Configuring the Size of the Interrupt Stack

CONSTANT:

CONFIGURE_INTERRUPT_STACK_SIZE

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is `CONFIGURE_MINIMUM_TASK_STACK_SIZE`, which is the minimum interrupt stack size.

DESCRIPTION:

`CONFIGURE_INTERRUPT_STACK_SIZE` is set to the size of the interrupt stack. The interrupt stack size is often set by the BSP but since this memory may be allocated from the RTEMS Workspace, it must be accounted for.

NOTES:

In some BSPs, changing this constant does NOT change the size of the interrupt stack, only the amount of memory reserved for it.

Patches which result in this constant only being used in memory calculations when the interrupt stack is intended to be allocated from the RTEMS Workspace would be welcomed by the RTEMS Project.

24.12.7 Reserve Task/Thread Stack Memory Above Minimum

CONSTANT:

`CONFIGURE_EXTRA_TASK_STACKS`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

This configuration parameter is set to 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 than the minimum, then that memory is NOT accounted for by `<rtems/confdefs.h>`.

24.12.8 Automatically Zeroing the RTEMS Workspace and C Program Heap

CONSTANT:

`CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, unless overridden by the BSP. The default is *NOT* to zero out the RTEMS Workspace or C Program Heap.

DESCRIPTION:

This macro indicates whether RTEMS should zero the RTEMS Workspace and C Program Heap as part of its initialization. If defined, the memory regions are zeroed. Otherwise, they are not.

NOTES:

Zeroing memory can add significantly to system boot time. It is not necessary for RTEMS but is often assumed by support libraries.

24.12.9 Enable The Task Stack Usage Checker

CONSTANT:

`CONFIGURE_STACK_CHECKER_ENABLED`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, and thus stack checking is disabled.

DESCRIPTION:

This configuration parameter is defined when the application wishes to enable runtime stack bounds checking.

NOTES:

In 4.9 and older, this configuration parameter was named `STACK_CHECKER_ON`.

This increases the time required to create tasks as well as adding overhead to each context switch.

24.12.10 Specify Application Specific User Extensions

CONSTANT:

CONFIGURE_INITIAL_EXTENSIONS

DATA TYPE:

List of user extension initializers (rtems_extensions_table).

RANGE:

Undefined or a list of one or more user extensions.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

If CONFIGURE_INITIAL_EXTENSIONS is defined by the application, then this application specific set of initial extensions will be placed in the initial extension table.

NOTES:

None.

24.13 Configuring Custom Task Stack Allocation

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.

24.13.1 Custom Task Stack Allocator Initialization

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR_INIT`

DATA TYPE:

Function pointer.

RANGE:

Undefined, NULL or valid function pointer.

DEFAULT VALUE:

The default value is NULL, which indicates that task stacks will be allocated from the RTEMS Workspace.

DESCRIPTION:

`CONFIGURE_TASK_STACK_ALLOCATOR_INIT` configures the initialization method for an application or BSP specific task stack allocation implementation.

NOTES:

A correctly configured system must configure the following to be consistent:

- `CONFIGURE_TASK_STACK_ALLOCATOR_INIT`
- `CONFIGURE_TASK_STACK_ALLOCATOR`
- `CONFIGURE_TASK_STACK_DEALLOCATOR`

24.13.2 Custom Task Stack Allocator

CONSTANT:

`CONFIGURE_TASK_STACK_ALLOCATOR`

DATA TYPE:

Function pointer.

RANGE:

Undefined or valid function pointer.

DEFAULT VALUE:

The default value is `_Workspace_Allocate`, which indicates that task stacks will be allocated from the RTEMS Workspace.

DESCRIPTION:

`CONFIGURE_TASK_STACK_ALLOCATOR` may point to a user provided routine to allocate task stacks.

NOTES:

A correctly configured system must configure the following to be consistent:

- `CONFIGURE_TASK_STACK_ALLOCATOR_INIT`
- `CONFIGURE_TASK_STACK_ALLOCATOR`
- `CONFIGURE_TASK_STACK_DEALLOCATOR`

24.13.3 Custom Task Stack Deallocator

CONSTANT:

`CONFIGURE_TASK_STACK_DEALLOCATOR`

DATA TYPE:

Function pointer.

RANGE:

Undefined or valid function pointer.

DEFAULT VALUE:

The default value is `_Workspace_Free`, which indicates that task stacks will be allocated from the RTEMS Workspace.

DESCRIPTION:

`CONFIGURE_TASK_STACK_DEALLOCATOR` may point to a user provided routine to free task stacks.

NOTES:

A correctly configured system must configure the following to be consistent:

- `CONFIGURE_TASK_STACK_ALLOCATOR_INIT`
- `CONFIGURE_TASK_STACK_ALLOCATOR`
- `CONFIGURE_TASK_STACK_DEALLOCATOR`

24.14 Configuring Memory for Classic API Message Buffers

This section describes the configuration parameters related to specifying the amount of memory reserved for Classic API Message Buffers.

24.14.1 Calculate Memory for a Single Classic Message API Message Queue

CONSTANT:

`CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE(max_messages, size_per)`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is None.

DESCRIPTION:

This is a helper macro which is used to assist in computing the total amount of memory required for message buffers. Each message queue will have its own configuration with maximum message size and maximum number of pending messages.

The interface for this macro is as follows:

```
1 CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE(max_
  ↳ messages, size_per)
```

Where `max_messages` is the maximum number of pending messages and `size_per` is the size in bytes of the user message.

NOTES:

This macro is only used in support of `CONFIGURE_MESSAGE_BUFFER_MEMORY`.

24.14.2 Reserve Memory for All Classic Message API Message Queues

CONSTANT:

`CONFIGURE_MESSAGE_BUFFER_MEMORY`

DATA TYPE:

integer summation macro

RANGE:

undefined (zero) or calculation resulting in a positive integer

DEFAULT VALUE:

This is not defined by default, and zero (0) memory is reserved.

DESCRIPTION:

This macro is set to the number of bytes the application requires to be reserved for pending Classic API Message Queue buffers.

NOTES:

The following illustrates how the help macro `CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE` 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 maximum of 24 pending messages with the message structure defined by the type `one_message_type`. The other message queue has maximum of 500 pending messages with the message structure defined by the type `other_message_type`.

```
1 #define CONFIGURE_MESSAGE_BUFFER_MEMORY \
2     (CONFIGURE_MESSAGE_BUFFERS_FOR_
3     ↳ QUEUE( \
4         24, sizeof(one_message_
5         ↳ type) \
6         ) + \
7     CONFIGURE_MESSAGE_BUFFERS_FOR_
8     ↳ QUEUE( \
9         500, sizeof(other_message_
10    ↳ type) \
11    )
```

24.15 Seldom Used Configuration Parameters

This section describes configuration parameters supported by `<rtems/confdefs.h>` which are seldom used by applications. These parameters tend to be oriented to debugging system configurations and providing workarounds when the memory estimated by `<rtems/confdefs.h>` is incorrect.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

This configuration parameter should only be defined if the application is providing their own complete set of configuration tables.

NOTES:

None.

24.15.1 Specify Memory Overhead

CONSTANT:

`CONFIGURE_MEMORY_OVERHEAD`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

This parameter is set to the number of kilobytes the application wishes to add to the requirements calculated by `<rtems/confdefs.h>`.

NOTES:

This configuration parameter 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.

24.15.2 Do Not Generate Configuration Information

CONSTANT:

`CONFIGURE_HAS_OWN_CONFIGURATION_TABLE`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

24.16 C Library Support Configuration

This section defines the file system and IO library related configuration parameters supported by `<rtems/confdefs.h>`.

24.16.1 Specify Maximum Number of File Descriptors

CONSTANT:

`CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

If `CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER` is defined, then the default value is 3, otherwise the default value is 0. Three file descriptors allows RTEMS to support standard input, output, and error I/O streams on `/dev/console`.

DESCRIPTION:

This configuration parameter is set to the maximum number of file like objects that can be concurrently open.

NOTES:

None.

24.16.2 Disable POSIX Termios Support

CONSTANT:

`CONFIGURE_TERMIO_DISABLED`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, and resources are reserved for the termios functionality.

DESCRIPTION:

This configuration parameter is defined if the software implementing POSIX termios

functionality is not going to be used by this application.

NOTES:

The termios support library should not be included in an application executable unless it is directly referenced by the application or a device driver.

24.16.3 Specify Maximum Termios Ports

CONSTANT:

`CONFIGURE_NUMBER_OF_TERMIO_PORTS`

DATA TYPE:

Unsigned integer.

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 1, so a console port can be used.

DESCRIPTION:

This configuration parameter is set to the number of ports using the termios functionality. Each concurrently active termios port requires resources.

NOTES:

If the application will be using serial ports including, but not limited to, the Console Device (e.g. `CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER`), then it is highly likely that this configuration parameter should NOT be is defined.

24.17 File System Configuration Parameters

This section defines File System related configuration parameters.

24.17.1 Providing Application Specific Mount Table

CONSTANT:

`CONFIGURE_HAS_OWN_MOUNT_TABLE`

DATA TYPE:

Undefined or an array of type `rtcms_filesystem_mount_table_t`.

RANGE:

Undefined or an array of type `rtcms_filesystem_mount_table_t`.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

This configuration parameter is defined when the application provides their own filesystem mount table. The mount table is an array of `rtcms_filesystem_mount_table_t` entries pointed to by the global variable `rtcms_filesystem_mount_table`. The number of entries in this table is in an integer variable named `rtcms_filesystem_mount_table_t`.

NOTES:

None.

24.17.2 Configure devFS as Root File System

CONSTANT:

`CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default. If no other root file system configuration parameters are

specified, the IMFS will be used as the root file system.

DESCRIPTION:

This configuration parameter is defined if the application wishes to use the device-only filesystem as the root file system.

NOTES:

The device-only filesystem supports only device nodes and is smaller in executable code size than the full IMFS and miniIMFS.

The devFS is comparable in functionality to the pseudo-filesystem name space provided before RTEMS release 4.5.0.

24.17.3 Specifying Maximum Devices for devFS

CONSTANT:

`CONFIGURE_MAXIMUM_DEVICES`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

If `BSP_MAXIMUM_DEVICES` is defined, then the default value is `BSP_MAXIMUM_DEVICES`, otherwise the default value is 4.

DESCRIPTION:

`CONFIGURE_MAXIMUM_DEVICES` is defined to the number of individual devices that may be registered in the device file system (devFS).

NOTES:

This option is specific to the device file system (devFS) and should not be confused with the `CONFIGURE_MAXIMUM_DRIVERS` option. This parameter only impacts the devFS and thus is only used by `<rtcms/confdefs.h>` when `CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM` is specified.

24.17.4 Disable File System Support

CONSTANT:

`CONFIGURE_APPLICATION_DISABLE_FILESYSTEM`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default. If no other root file system configuration parameters are specified, the IMFS will be used as the root file system.

DESCRIPTION:

This configuration parameter is defined if the application does not intend to use any kind of filesystem support. This includes the device infrastructure necessary to support `printf()`.

NOTES:

None.

24.17.5 Use a Root IMFS with a Minimalistic Feature Set

CONSTANT:

`CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the following configuration options will be defined as well

- `CONFIGURE_IMFS_DISABLE_CHMOD,`
- `CONFIGURE_IMFS_DISABLE_CHOWN,`
- `CONFIGURE_IMFS_DISABLE_UTIME,`
- `CONFIGURE_IMFS_DISABLE_LINK,`
- `CONFIGURE_IMFS_DISABLE_SYMLINK,`
- `CONFIGURE_IMFS_DISABLE_READLINK,`
- `CONFIGURE_IMFS_DISABLE_RENAME,` and
- `CONFIGURE_IMFS_DISABLE_UNMOUNT.`

24.17.6 Specify Block Size for IMFS

CONSTANT:

`CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK`

DATA TYPE:

Boolean feature macro.

RANGE:

Valid values for this configuration parameter are a power of two (2) between 16 and 512 inclusive. In other words, valid values are 16, 32, 64, 128, 256, and 512.

DEFAULT VALUE:

The default IMFS block size is 128 bytes.

DESCRIPTION:

This configuration parameter specifies the block size for in-memory files managed by the IMFS. 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.

24.17.7 Disable Change Owner Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_CHOWN

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to change the owner is disabled in the root IMFS.

24.17.8 Disable Change Mode Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_CHMOD

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to change the mode is disabled in the root IMFS.

24.17.9 Disable Change Times Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_UTIME

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to change times is disabled in the root IMFS.

24.17.10 Disable Create Hard Link Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_LINK

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to create hard links is disabled in the root IMFS.

24.17.11 Disable Create Symbolic Link Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_SYMLINK

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to create symbolic links is disabled in the root IMFS.

24.17.12 Disable Read Symbolic Link Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_READLINK

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to read symbolic links is disabled in the root IMFS.

24.17.13 Disable Rename Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_RENAME

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to rename nodes is disabled in the root IMFS.

24.17.14 Disable Directory Read Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_READDIR

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to read a directory is disabled in the root IMFS. It is still possible to open nodes in a directory.

24.17.15 Disable Mount Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_MOUNT

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to mount other file systems is disabled in the root IMFS.

24.17.16 Disable Unmount Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_UNMOUNT

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to unmount file systems is disabled in the root IMFS.

24.17.17 Disable Make Nodes Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_MKNOD

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to make directories, devices, regular files and FIFOs is disabled in the root IMFS.

24.17.18 Disable Make Files Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_MKNOD_FILE

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to make regular files is disabled in the root IMFS.

24.17.19 Disable Remove Nodes Support of Root IMFS

CONSTANT:

CONFIGURE_IMFS_DISABLE_RMNOD

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

In case this configuration option is defined, then the support to remove nodes is disabled in the root IMFS.

24.18 Block Device Cache Configuration

This section defines Block Device Cache (bdbuf) related configuration parameters.

24.18.1 Enable Block Device Cache

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_LIBBLOCK`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

Provides a Block Device Cache configuration.

NOTES:

Each option of the Block Device Cache 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 file systems.

24.18.2 Size of the Cache Memory

CONSTANT:

`CONFIGURE_BDBUF_CACHE_MEMORY_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 32768 bytes.

DESCRIPTION:

Size of the cache memory in bytes.

NOTES:

None.

24.18.3 Minimum Size of a Buffer

CONSTANT:

`CONFIGURE_BDBUF_BUFFER_MIN_SIZE`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 512 bytes.

DESCRIPTION:

Defines the minimum size of a buffer in bytes.

NOTES:

None.

24.18.4 Maximum Size of a Buffer

CONSTANT:

`CONFIGURE_BDBUF_BUFFER_MAX_SIZE`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

It must be positive and an integral multiple of the buffer minimum size.

DEFAULT VALUE:

The default value is 4096 bytes.

DESCRIPTION:

Defines the maximum size of a buffer in bytes.

NOTES:

None.

24.18.5 Swapout Task Swap Period

CONSTANT:

`CONFIGURE_SWAPOUT_SWAP_PERIOD`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 250 milliseconds.

DESCRIPTION:

Defines the swapout task swap period in milliseconds.

NOTES:

None.

24.18.6 Swapout Task Maximum Block Hold Time

CONSTANT:

CONFIGURE_SWAPOUT_BLOCK_HOLD

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 1000 milliseconds.

DESCRIPTION:

Defines the swapout task maximum block hold time in milliseconds.

NOTES:

None.

24.18.7 Swapout Task Priority

CONSTANT:

CONFIGURE_SWAPOUT_TASK_PRIORITY

DATA TYPE:

Task priority (rtcms_task_priority).

RANGE:

Valid task priority.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

Defines the swapout task priority.

NOTES:

None.

24.18.8 Maximum Blocks per Read-Ahead Request

CONSTANT:

CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

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.

24.18.9 Maximum Blocks per Write Request

CONSTANT:

CONFIGURE_BDBUF_MAX_WRITE_BLOCKS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 16.

DESCRIPTION:

Defines the maximum blocks per write request.

NOTES:

None.

24.18.10 Task Stack Size of the Block Device Cache Tasks

CONSTANT:

CONFIGURE_BDBUF_TASK_STACK_SIZE

DATA TYPE:

Unsigned integer (size_t).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is RTEMS_MINIMUM_STACK_SIZE.

DESCRIPTION:

Defines the task stack size of the Block Device Cache tasks in bytes.

NOTES:

None.

24.18.11 Read-Ahead Task Priority

CONSTANT:

CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY

DATA TYPE:

Task priority (`rtems_task_priority`).

RANGE:

Valid task priority.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

Defines the read-ahead task priority.

NOTES:

None.

24.18.12 Swapout Worker Task Count

CONSTANT:

CONFIGURE_SWAPOUT_WORKER_TASKS

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

Defines the swapout worker task count.

NOTES:

None.

24.18.13 Swapout Worker Task Priority

CONSTANT:

CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY

DATA TYPE:

Task priority (`rtems_task_priority`).

RANGE:

Valid task priority.

DEFAULT VALUE:

The default value is 15.

DESCRIPTION:

Defines the swapout worker task priority.

NOTES:

None.

24.19 BSP Specific Settings

This section describes BSP specific configuration settings used by `<rtems/confdefs.h>`. The BSP specific configuration settings are defined in `<bsp.h>`.

24.19.1 Disable BSP Configuration Settings

CONSTANT:

`CONFIGURE_DISABLE_BSP_SETTINGS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

All BSP specific configuration settings can be disabled by the application with the `CONFIGURE_DISABLE_BSP_SETTINGS` option.

NOTES:

None.

24.19.2 Specify BSP Supports `sbrk()`

CONSTANT:

`CONFIGURE_MALLOC_BSP_SUPPORTS_SBRK`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

This configuration parameter is defined by a BSP to indicate that it does not allocate all available memory to the C Program Heap used by the Malloc Family of routines.

If defined, when `malloc()` is unable to allocate memory, it will call the BSP supplied `sbrk()` to obtain more memory.

NOTES:

This parameter should not be defined by the application. Only the BSP knows how it allocates memory to the C Program Heap.

24.19.3 Specify BSP Specific Idle Task

CONSTANT:

`BSP_IDLE_TASK_BODY`

DATA TYPE:

Function pointer.

RANGE:

Undefined or valid function pointer.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_IDLE_TASK_BODY` is defined by the BSP and `CONFIGURE_IDLE_TASK_BODY` is not defined by the application, then this BSP specific idle task body will be used.

NOTES:

As it has knowledge of the specific CPU model, system controller logic, and peripheral buses, a BSP specific IDLE task may be capable of turning components off to save power during extended periods of no task activity

24.19.4 Specify BSP Suggested Value for IDLE Task Stack Size

CONSTANT:

`BSP_IDLE_TASK_STACK_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_IDLE_TASK_STACK_SIZE` is defined by the BSP and `CONFIGURE_IDLE_TASK_STACK_SIZE` is not defined by the application, then this BSP suggested idle task stack size will be used.

NOTES:

The order of precedence for configuring the IDLE task stack size is:

- RTEMS default minimum stack size.
- If `CONFIGURE_MINIMUM_TASK_STACK_SIZE` defined, then `CONFIGURE_MINIMUM_TASK_STACK_SIZE`.
- If `BSP_IDLE_TASK_SIZE` defined, then the BSP specific `BSP_IDLE_TASK_SIZE`.
- If defined, then the application specified `CONFIGURE_IDLE_TASK_SIZE`.

24.19.5 Specify BSP Specific User Extensions

CONSTANT:

`BSP_INITIAL_EXTENSION`

DATA TYPE:

List of user extension initializers (`rtcms_extensions_table`).

RANGE:

Undefined or a list of user extension initializers.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_INITIAL_EXTENSION` is defined by the BSP, then this BSP specific initial extension will be placed as the last entry in the initial extension table.

NOTES:

None.

24.19.6 Specifying BSP Specific Interrupt Stack Size

CONSTANT:

`BSP_INTERRUPT_STACK_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_INTERRUPT_STACK_SIZE` is defined by the BSP and `CONFIGURE_INTERRUPT_STACK_SIZE` is not defined by the application, then this BSP specific interrupt stack size will be used.

NOTES:

None.

24.19.7 Specifying BSP Specific Maximum Devices

CONSTANT:

`BSP_MAXIMUM_DEVICES`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_MAXIMUM_DEVICES` is defined by the BSP and `CONFIGURE_MAXIMUM_DEVICES` is not defined by the application, then this BSP specific maximum device count will be used.

NOTES:

This option is specific to the device file system (`devFS`) and should not be confused with the `CONFIGURE_MAXIMUM_DRIVERS` option. This parameter only impacts the `devFS` and thus is only used by `<rtcms/confdefs.h>` when `CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM` is specified.

24.19.8 BSP Recommends RTEMS Workspace be Cleared

CONSTANT:

`BSP_ZERO_WORKSPACE_AUTOMATICALLY`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

If `BSP_ZERO_WORKSPACE_AUTOMATICALLY` is defined by the BSP and `CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY` is not defined by the application, then the workspace will be zeroed automatically.

NOTES:

Zeroing memory can add significantly to system boot time. It is not necessary for RTEMS but is often assumed by support libraries.

24.19.9 Specify BSP Prerequisite Drivers

CONSTANT:

`CONFIGURE_BSP_PREREQUISITE_DRIVERS`

DATA TYPE:

List of device driver initializers (`rtcms_driver_address_table`).

RANGE:

Undefined or array of device drivers.

DEFAULT VALUE:

This option is BSP specific.

DESCRIPTION:

`CONFIGURE_BSP_PREREQUISITE_DRIVERS` is defined if the BSP has device drivers it needs to include in the Device Driver Table. This should be defined to the set of device driver entries that will be placed in the table at the *FRONT* of the Device Driver Table and initialized before any other drivers *INCLUDING* any application prerequisite drivers.

NOTES:

`CONFIGURE_BSP_PREREQUISITE_DRIVERS` is typically used by BSPs to configure common infrastructure such as bus controllers or probe for devices.

24.20 Idle Task Configuration

This section defines the IDLE task related configuration parameters supported by `<rtems/confdefs.h>`.

24.20.1 Specify Application Specific Idle Task Body

CONSTANT:

`CONFIGURE_IDLE_TASK_BODY`

DATA TYPE:

Function pointer.

RANGE:

Undefined or valid function pointer.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_IDLE_TASK_BODY` is set to the function name corresponding to the application specific IDLE thread body. If not specified, the BSP or RTEMS default IDLE thread body will be used.

NOTES:

None.

24.20.2 Specify Idle Task Stack Size

CONSTANT:

`CONFIGURE_IDLE_TASK_STACK_SIZE`

DATA TYPE:

Unsigned integer (`size_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

The default value is `RTEMS_MINIMUM_STACK_SIZE`.

DESCRIPTION:

`CONFIGURE_IDLE_TASK_STACK_SIZE` is set to the desired stack size for the IDLE task.

NOTES:

None.

24.20.3 Specify Idle Task Performs Application Initialization

CONSTANT:

`CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, the user is assumed to provide one or more initialization tasks.

DESCRIPTION:

`CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION` is set 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 must make itself preemptible and enter an IDLE body loop.

The IDLE task must run at the lowest priority of all tasks in the system.

24.21 Scheduler Algorithm Configuration

This section defines the configuration parameters related to selecting a scheduling algorithm for an application. For the schedulers built into RTEMS, the configuration is straightforward. All that is required is to define the configuration macro which specifies which scheduler you want for in your application. The currently available schedulers are:

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 macros.

24.21.1 Use Deterministic Priority Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_PRIORITY`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is defined by default. This is the default scheduler and specifying this configuration parameter is redundant.

DESCRIPTION:

The Deterministic Priority Scheduler is the default scheduler in RTEMS for uni-processor applications and is designed for predictable performance under the highest loads. It can block or unblock a thread in a constant amount of time. This scheduler requires a variable amount of memory based upon the number of priorities configured in the system.

NOTES:

This scheduler may be explicitly selected by defining `CONFIGURE_SCHEDULER_PRIORITY` although this is equivalent to the default behavior.

24.21.2 Use Simple Priority Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_SIMPLE`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

When defined, the Simple Priority Scheduler is used at the thread scheduling algorithm. This is an alternative scheduler in RTEMS. It is designed to provide the same task scheduling behaviour as the Deterministic Priority Scheduler while being simpler in implementation and uses less memory for data management. It maintains a single sorted list of all ready threads. Thus blocking or unblocking a thread is not a constant time operation with this scheduler.

This scheduler may be explicitly selected by defining `CONFIGURE_SCHEDULER_SIMPLE`.

NOTES:

This scheduler is appropriate for use in small systems where RAM is limited.

24.21.3 Use Earliest Deadline First Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_EDF`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

The Earliest Deadline First Scheduler (EDF) is an alternative scheduler in RTEMS for uni-processor applications. The EDF schedules tasks with dynamic priorities equal to deadlines. The deadlines are declared using only Rate Monotonic manager which

handles periodic behavior. Period is always equal to deadline. If a task does not have any deadline declared or the deadline is cancelled, the task is considered a background task which is scheduled in case no deadline-driven tasks are ready to run. Moreover, multiple background tasks are scheduled according to their priority assigned upon initialization. All ready tasks reside in a single ready queue.

This scheduler may be explicitly selected by defining `CONFIGURE_SCHEDULER_EDF`.

NOTES:

None.

24.21.4 Use Constant Bandwidth Server Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_CBS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

The Constant Bandwidth Server Scheduler (CBS) is an alternative scheduler in RTEMS for uni-processor applications. The CBS is a budget aware extension of EDF scheduler. The goal of this scheduler is to ensure temporal isolation of tasks. The CBS is equipped with a set of additional rules and provides with an extensive API.

This scheduler may be explicitly selected by defining `CONFIGURE_SCHEDULER_CBS`.

NOTES:

None.

24.21.5 Use Deterministic Priority SMP Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_PRIORITY_SMP`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

The Deterministic Priority SMP Scheduler is derived from the Deterministic Priority Scheduler but is capable of scheduling threads across multiple processors.

In a configuration with SMP enabled at configure time, it may be explicitly selected by defining `CONFIGURE_SCHEDULER_PRIORITY_SMP`.

NOTES:

This scheduler is only available when RTEMS is configured with SMP support enabled.

This scheduler is currently the default in SMP configurations and is only selected when `CONFIGURE_SMP_APPLICATION` is defined.

24.21.6 Use Simple SMP Priority Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_SIMPLE_SMP`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

The Simple SMP Priority Scheduler is derived from the Simple Priority Scheduler but is capable of scheduling threads across multiple processors. It is designed to provide the same task scheduling behaviour as the Deterministic Priority Scheduler while distributing threads across multiple processors. Being based upon the Simple Priority Scheduler, it also maintains a single sorted list of

all ready threads. Thus blocking or unblocking a thread is not a constant time operation with this scheduler.

In addition, when allocating threads to processors, the algorithm is not constant time. This algorithm was not designed with efficiency as a primary design goal. Its primary design goal was to provide an SMP-aware scheduling algorithm that is simple to understand.

In a configuration with SMP enabled at configure time, it may be explicitly selected by defining `CONFIGURE_SCHEDULER_SIMPLE_SMP`.

NOTES:

This scheduler is only available when RTEMS is configured with SMP support enabled.

24.21.7 Configuring a Scheduler Name

CONSTANT:

`CONFIGURE_SCHEDULER_NAME`

DATA TYPE:

RTEMS Name (`rtems_name`).

RANGE:

Any value.

DEFAULT VALUE:

The default name is

- "UCBS" for the Uni-Processor CBS scheduler,
- "UEDF" for the Uni-Processor EDF scheduler,
- "UPD " for the Uni-Processor Deterministic Priority scheduler,
- "UPS " for the Uni-Processor Simple Priority scheduler,
- "MPA " for the Multi-Processor Priority Affinity scheduler, and
- "MPD " for the Multi-Processor Deterministic Priority scheduler, and
- "MPS " for the Multi-Processor Simple Priority scheduler.

DESCRIPTION:

Schedulers can be identified via `rtems_scheduler_ident`. The name of the scheduler is determined by the configuration.

NOTES:

None.

24.21.8 Configuring a User Provided Scheduler

CONSTANT:

`CONFIGURE_SCHEDULER_USER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

RTEMS allows the application to provide its own task/thread scheduling algorithm. In order to do this, one must define `CONFIGURE_SCHEDULER_USER` to indicate the application provides its own scheduling algorithm. If `CONFIGURE_SCHEDULER_USER` is defined then the following additional macros must be defined:

- `CONFIGURE_SCHEDULER_CONTEXT` must be defined to a static definition of the scheduler context of the user scheduler.
- `CONFIGURE_SCHEDULER_CONTROLS` must be defined to a scheduler control initializer for the user scheduler.
- `CONFIGURE_SCHEDULER_USER_PER_THREAD` must be defined to the type of the per-thread information of the user scheduler.

NOTES:

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/sapi/include/confdefs.h` for how these are defined for the Deterministic Priority Scheduler.

24.21.9 Configuring Clustered Schedulers

Clustered scheduling helps to control the worst-case latencies in a multi-processor system. The goal is to reduce the amount of shared state in the system and thus prevention of lock contention. Modern multi-processor 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 instance. 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 is used for which cluster?

CONFIGURATION:

The schedulers in an SMP system are statically configured on RTEMS. Firstly the application must select which scheduling algorithms are available with the following defines

- `CONFIGURE_SCHEDULER_PRIORITY_SMP`,
- `CONFIGURE_SCHEDULER_SIMPLE_SMP`,
and
- `CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP`

This is necessary to calculate the per-thread overhead introduced by the schedulers. After these definitions the configuration file must `#include <rtems/scheduler.h>` to have access to scheduler specific configuration macros. Each scheduler needs a context to store state information at run-time. To provide a context for each scheduler is the

next step. Use the following macros to create scheduler contexts

- `RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP(name, prio_...)`
- `RTEMS_SCHEDULER_CONTEXT_SIMPLE_SMP(name)`,
and
- `RTEMS_SCHEDULER_CONTEXT_PRIORITY_AFFINITY_SMP(name)`

The name parameter is used as part of a designator for a global variable, so the usual C/C++ designator rules apply. Additional parameters are scheduler specific. The schedulers are registered in the system via the scheduler table. To create the scheduler table define `CONFIGURE_SCHEDULER_CONTROLS` to a list of the following scheduler control initializers

- `RTEMS_SCHEDULER_CONTROL_PRIORITY_SMP(name, obj_name)`
- `RTEMS_SCHEDULER_CONTROL_SIMPLE_SMP(name, obj_name)`,
and
- `RTEMS_SCHEDULER_CONTROL_PRIORITY_AFFINITY_SMP(name)`

The name parameter must correspond to the parameter defining the scheduler context. The obj_name determines the scheduler object name and can be used in `rtems_scheduler_ident()` to get the scheduler object identifier.

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_SMP_MAXIMUM_PROCESSORS`).

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_SMP_SCHEDULER_ASSIGNMENTS` to a list of `RTEMS_SCHEDULER_ASSIGN(index,attr)` and `RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER` macros. The index parameter must be a valid index into the scheduler table. 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.

ERRORS:

In case one of the scheduler indices in "CONFIGURE_SMP_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. None of the errors is internal.

- 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_SMP_SCHEDULER_ASSIGNMENTS definition.
- SMP_FATAL_MULTITASKING_START_ON_UNASSIGNED - it is not allowed to start multitasking on a processor with no scheduler assigned.

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 scheduler instances 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 runtime 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.

```

1 #define CONFIGURE_SMP_MAXIMUM_PROCESSORS 8
2 #define CONFIGURE_MAXIMUM_PRIORITY 255
3 /* Make the scheduler algorithm available
   ↳ */
4 #define CONFIGURE_SCHEDULER_PRIORITY_SMP
5 #include <rtems/scheduler.h>
6 /* Create contexts for the two scheduler
   ↳ instances */
7 RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP(io,
   ↳ CONFIGURE_MAXIMUM_PRIORITY + 1);
8 RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP(work,
   ↳ CONFIGURE_MAXIMUM_PRIORITY + 1);
9 /* Define the scheduler table */
10 #define CONFIGURE_SCHEDULER_CONTROLS \
11     RTEMS_SCHEDULER_CONTROL_
12     ↳ PRIORITY_SMP( \
13         io, \
14         rtems_build_name('I', 'O',
15         ↳ ' ', ' ') \
16         ), \
17     RTEMS_SCHEDULER_CONTROL_
18     ↳ PRIORITY_SMP( \
19         work, \
20         rtems_build_name('W', 'O',
21         ↳ 'R', 'K') \
22         )
23 /* Define the processor to scheduler
   ↳ assignments */
24 #define CONFIGURE_SMP_SCHEDULER_
   ↳ ASSIGNMENTS \

```

```
21      RTEMS_SCHEDULER_ASSIGN(0, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
↳ \
22      RTEMS_SCHEDULER_ASSIGN_NO_
↳ SCHEDULER, \
23      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
↳ \
24      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_MANDATORY), \
↳ \
25      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
26      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
27      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL), \
28      RTEMS_SCHEDULER_ASSIGN(1, RTEMS_
↳ SCHEDULER_ASSIGN_PROCESSOR_OPTIONAL)
```

24.22 SMP Specific Configuration Parameters

When RTEMS is configured to support SMP target systems, there are other configuration parameters which apply.

24.22.1 Enable SMP Support for Applications

CONSTANT:

CONFIGURE_SMP_APPLICATION

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_SMP_APPLICATION must be defined to enable SMP support for the application.

NOTES:

This define may go away in the future in case all RTEMS components are SMP ready. This configuration define is ignored on uni-processor configurations.

24.22.2 Specify Maximum Processors in SMP System

CONSTANT:

CONFIGURE_SMP_MAXIMUM_PROCESSORS

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Defined or undefined.

DEFAULT VALUE:

The default value is 1, (if CONFIGURE_SMP_APPLICATION is defined).

DESCRIPTION:

CONFIGURE_SMP_MAXIMUM_PROCESSORS must be set to the number of processors in the SMP configuration.

NOTES:

If there are more processors available than configured, the rest will be ignored. This configuration define is ignored on uni-processor configurations.

24.23 Device Driver Table

This section defines the configuration parameters related to the automatic generation of a Device Driver Table. As `<rtcms/confdefs.h>` only is aware of a small set of standard device drivers, the generated Device Driver Table is suitable for simple applications with no custom device drivers.

Note that network device drivers are not configured in the Device Driver Table.

24.23.1 Specifying the Maximum Number of Device Drivers

CONSTANT:

`CONFIGURE_MAXIMUM_DRIVERS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

This is computed by default, and is set to the number of device drivers configured using the `CONFIGURE_APPLICATIONS_NEEDS_XXX_DRIVER` configuration parameters.

DESCRIPTION:

`CONFIGURE_MAXIMUM_DRIVERS` is defined as the number of device drivers per node.

NOTES:

If the application will dynamically install device drivers, then this configuration parameter must be larger than the number of statically configured device drivers. Drivers configured using the `CONFIGURE_APPLICATIONS_NEEDS_XXX_DRIVER` configuration parameters are statically installed.

24.23.2 Enable Console Device Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER` is defined if the application wishes to include the Console Device Driver.

NOTES:

This device driver is responsible for providing standard input and output using `/dev/console`.

BSPs should be constructed in a manner that allows `printk()` to work properly without the need for the console driver to be configured.

24.23.3 Enable Clock Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER` is defined if the application wishes to include the Clock Device Driver.

NOTES:

This device driver is responsible for providing a regular interrupt which invokes a clock tick directive.

If neither the Clock Driver nor Benchmark Timer is enabled and the configuration parameter `CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER` is not defined, then a compile time error will occur.

24.23.4 Enable the Benchmark Timer Driver

CONSTANT:

CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER is defined if the application wishes to include the Timer Driver. This device driver is used to benchmark execution times by the RTEMS Timing Test Suites.

NOTES:

If neither the Clock Driver nor Benchmark Timer is enabled and the configuration parameter CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER is not defined, then a compile time error will occur.

24.23.5 Specify Clock and Benchmark Timer Drivers Are Not Needed

CONSTANT:

CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER is defined when the application does *NOT* want the Clock Device Driver and is *NOT* using the Timer Driver. The inclusion or exclusion of the Clock Driver must be explicit in user applications.

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.

24.23.6 Enable Real-Time Clock Driver

CONSTANT:

CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER is defined if the application wishes to include the Real-Time Clock Driver.

NOTES:

Most BSPs do not include support for a real-time clock. 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.

24.23.7 Enable the Watchdog Device Driver

CONSTANT:

CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER is defined if the application wishes to include the Watchdog Driver.

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.

24.23.8 Enable the Graphics Frame Buffer Device Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER` is defined if the application wishes to include the BSP's Frame Buffer Device Driver.

NOTES:

Most BSPs do not include support for a Frame Buffer 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.

24.23.9 Enable Stub Device Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER`

is defined if the application wishes to include the Stub Device Driver.

NOTES:

This device driver simply provides entry points that return successful and is primarily a test fixture. It is supported by all BSPs.

24.23.10 Specify Application Prerequisite Device Drivers

CONSTANT:

`CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS`

DATA TYPE:

device driver entry structures

RANGE:

Undefined or set of device driver entry structures

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS` is defined if the application has device drivers it needs to include in the Device Driver Table. This should be defined to the set of device driver entries that will be placed in the table at the *FRONT* of the Device Driver Table and initialized before any other drivers *EXCEPT* any BSP prerequisite drivers.

NOTES:

In some cases, it is used by System On Chip BSPs to support peripheral buses beyond those normally found on the System On Chip. For example, this is used by one RTEMS system which has implemented a SPARC/ERC32 based board with VMEBus. The VMEBus Controller initialization is performed by a device driver configured via this configuration parameter.

24.23.11 Specify Extra Application Device Drivers

CONSTANT:

`CONFIGURE_APPLICATION_EXTRA_DRIVERS`

DATA TYPE:

device driver entry structures

RANGE:

Undefined or set of device driver entry structures

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_APPLICATION_EXTRA_DRIVERS` is defined if the application has device drivers it needs to include in the Device Driver Table. This should be defined to the set of device driver entries that will be placed in the table at the *END* of the Device Driver Table.

NOTES:

None.

24.23.12 Enable /dev/null Device Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

This configuration variable is specified to enable /dev/null device driver.

NOTES:

This device driver is supported by all BSPs.

24.23.13 Enable /dev/zero Device Driver

CONSTANT:

`CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

This configuration variable is specified to enable /dev/zero device driver.

NOTES:

This device driver is supported by all BSPs.

24.23.14 Specifying Application Defined Device Driver Table

CONSTANT:

`CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default, indicating the `<rtems/confdefs.h>` is providing the device driver table.

DESCRIPTION:

`CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE` is defined if the application wishes to provide their own Device Driver Table.

The table must be an array of `rtems_driver_address_table` entries named “`_IO_Driver_address_table`“. The application must also provide a const variable `_IO_Number_of_drivers` of type `size_t` indicating the number of entries in the `_IO_Driver_address_table`.

NOTES:

It is expected that there the application would only rarely need to do this.

24.24 Multiprocessing Configuration

This section defines the multiprocessing related system configuration parameters supported by `<rtems/confdefs.h>`. They are only used if the Multiprocessing Support (distinct from the SMP support) is enabled at configure time using the `--enable-multiprocessing` option.

Additionally, this class of Configuration Constants are only applicable if `CONFIGURE_MP_APPLICATION` is defined.

24.24.1 Specify Application Will Use Multiprocessing

CONSTANT:

`CONFIGURE_MP_APPLICATION`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

This configuration parameter must be defined to indicate that the application intends to be part of a multiprocessing configuration. Additional configuration parameters are assumed to be provided.

NOTES:

This has no impact unless RTEMS was configured and built using the `--enable-multiprocessing` option.

24.24.2 Configure Node Number in Multiprocessor Configuration

CONSTANT:

`CONFIGURE_MP_NODE_NUMBER`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is `NODE_NUMBER`, which is assumed to be set by the compilation environment.

DESCRIPTION:

`CONFIGURE_MP_NODE_NUMBER` is 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.

24.24.3 Configure Maximum Node in Multiprocessor Configuration

CONSTANT:

`CONFIGURE_MP_MAXIMUM_NODES`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 2.

DESCRIPTION:

`CONFIGURE_MP_MAXIMUM_NODES` is the maximum number of nodes in a multiprocessor system.

NOTES:

None.

24.24.4 Configure Maximum Global Objects in Multiprocessor Configuration

CONSTANT:

`CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Positive.

DEFAULT VALUE:

The default value is 32.

DESCRIPTION:

CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS is 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.

24.24.5 Configure Maximum Proxies in Multiprocessor Configuration

CONSTANT:

CONFIGURE_MP_MAXIMUM_PROXIES

DATA TYPE:

Unsigned integer (uint32_t).

RANGE:

Undefined or positive.

DEFAULT VALUE:

The default value is 32.

DESCRIPTION:

CONFIGURE_MP_MAXIMUM_PROXIES is 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.

24.24.6 Configure MPCPI in Multiprocessor Configuration

CONSTANT:

CONFIGURE_MP_MPCPI_TABLE_POINTER

DATA TYPE:

pointer to rtems_mpci_table

RANGE:

undefined or valid pointer

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_MP_MPCPI_TABLE_POINTER is the pointer to the MPCPI Configuration Table. The default value of this field is "&MPCPI_table".

NOTES:

RTEMS provides a Shared Memory MPCPI Device Driver which can be used on any Multiprocessor System assuming the BSP provides the proper set of supporting methods.

24.24.7 Do Not Generate Multiprocessor Configuration Table

CONSTANT:

CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE is defined if the application wishes to provide their own Multiprocessing Configuration Table. The generated table is named Multiprocessing_configuration.

NOTES:

This is a configuration parameter which is very unlikely to be used by an application. If you find yourself wanting to use it in an application, please reconsider and discuss this on the RTEMS Users mailing list.

24.25 Ada Tasks

This section defines the system configuration parameters supported by `<rtems/confdefs.h>` related to configuring RTEMS to support a task using Ada tasking with GNAT/RTEMS.

These configuration parameters are only available when RTEMS is built with the `--enable-ada` configure option and the application specifies `CONFIGURE_GNAT_RTEMS`.

Additionally RTEMS includes an Ada language binding to the Classic API which has a test suite. This test suite is enabled only when `--enable-tests` and `--enable-expada` are specified on the configure command.

24.25.1 Specify Application Includes Ada Code

CONSTANT:

`CONFIGURE_GNAT_RTEMS`

DATA TYPE:

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_GNAT_RTEMS` is defined to inform RTEMS that the GNAT Ada run-time is to be used by the application.

NOTES:

This configuration parameter is critical as it makes `<rtems/confdefs.h>` configure the resources (POSIX API Threads, Mutexes, Condition Variables, and Keys) used implicitly by the GNAT run-time.

24.25.2 Specify the Maximum Number of Ada Tasks.

CONSTANT:

`CONFIGURE_MAXIMUM_ADA_TASKS`

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Undefined or positive.

DEFAULT VALUE:

If `CONFIGURE_GNAT_RTEMS` is defined, then the default value is 20, otherwise the default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_ADA_TASKS` is the number of Ada tasks that can be concurrently active in the system.

NOTES:

None.

24.25.3 Specify the Maximum Fake Ada Tasks

CONSTANT:

DATA TYPE:

Unsigned integer (`uint32_t`).

RANGE:

Zero or positive.

DEFAULT VALUE:

The default value is 0.

DESCRIPTION:

`CONFIGURE_MAXIMUM_FAKE_ADA_TASKS` is the number of *fake* Ada tasks that can be concurrently active in the system. A *fake* Ada task is a non-Ada task that makes calls back into Ada code and thus implicitly uses the Ada run-time.

NOTES:

None.

24.26 PCI Library

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 result in link errors. Note also that in these modes Plug & Play is not performed.

24.27 Go Tasks

24.27.1 Specify Application Includes Go Code

CONSTANT:`CONFIGURE_ENABLE_GO`**DATA TYPE:**

Boolean feature macro.

RANGE:

Defined or undefined.

DEFAULT VALUE:

This is not defined by default.

DESCRIPTION:

`CONFIGURE_ENABLE_GO` is defined to inform RTEMS that the Go run-time is to be used by the application.

NOTES:

The Go language support is experimental

24.27.2 Specify the maximum number of Go routines

CONSTANT:`CONFIGURE_MAXIMUM_GOROUTINES`**DATA TYPE:**Unsigned integer (`uint32_t`).**RANGE:**

Zero or positive.

DEFAULT VALUE:

The default value is 400

DESCRIPTION:

`CONFIGURE_MAXIMUM_GOROUTINES` is defined to specify the maximum number of Go routines.

NOTES:

The Go language support is experimental

24.27.3 Specify the maximum number of Go Channels

CONSTANT:`CONFIGURE_MAXIMUM_GO_CHANNELS`**DATA TYPE:**Unsigned integer (`uint32_t`).**RANGE:**

Zero or positive.

DEFAULT VALUE:

The default value is 500

DESCRIPTION:

`CONFIGURE_MAXIMUM_GO_CHANNELS` is defined to specify the maximum number of Go channels.

NOTES:

The Go language support is experimental

24.28 Configuration Data Structures

It is recommended that applications be configured using `<rtems/confdefs.h>` as it is simpler and insulates applications from changes in the underlying data structures. However, it is sometimes important to understand the data structures that are automatically filled in by the configuration parameters. This section describes the primary configuration data structures.

If the user wishes to see the details of a particular data structure, they are advised to look at the source code. After all, that is one of the advantages of RTEMS.

MULTIPROCESSING MANAGER

25.1 Introduction

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 Manager are:

- *rtems_multiprocessing_announce* (page 375) - A multiprocessing communications packet has arrived

25.2 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 MPCCI layer.

25.2.1 Nodes

A processor in a RTEMS system is referred to as a node. Each node is assigned a unique non-zero 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 oper-

ations. Thus, the Multiprocessor Communications Interface Layer (MPCCI) must be able to route messages based on the node number.

25.2.2 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 in 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.

25.2.3 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.

25.2.4 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 MPCCI 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 MPCCI 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 MPCCI 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 MPCCI 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 MPCCI routine `RECEIVE_PACKET`, performs the requested

operation, builds an RR message, and returns it to the originating node.

8. The MPCCI 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 MPCCI 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 MPCCI layer, the fatal error handler should be invoked. RTEMS assumes the reliable transmission and reception of messages by the MPCCI and makes no attempt to detect or correct errors.

25.2.5 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.

25.2.6 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.

25.3 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.

25.3.1 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 adhere to the following prototype:

```
1 rtems_mpci_entry user_mpci_initialization(
2     rtems_configuration_table *configuration
3 );
```

where `configuration` is the address of the user's Configuration Table. 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.

25.3.2 GET_PACKET

The GET_PACKET component of the user-provided MPCPI 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:

```
1 rtems_mpci_entry user_mpci_get_packet(  
2   rtems_packet_prefix **packet  
3 );
```

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.

25.3.3 RETURN_PACKET

The RETURN_PACKET component of the user-provided MPCPI 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:

```
1 rtems_mpci_entry user_mpci_return_packet(  
2   rtems_packet_prefix *packet  
3 );
```

where packet is the address of a packet. If the packet cannot be successfully returned, the fatal error manager should be invoked.

25.3.4 RECEIVE_PACKET

The RECEIVE_PACKET component of the user-provided MPCPI layer is called when RTEMS needs to obtain a packet which has previously arrived. This component should be adhere to the following prototype:

```
1 rtems_mpci_entry user_mpci_receive_packet(  
2   rtems_packet_prefix **packet  
3 );
```

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.

25.3.5 SEND_PACKET

The SEND_PACKET component of the user-provided MPCPI 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:

```
1 rtems_mpci_entry user_mpci_send_packet(  
2   uint32_t          node,  
3   rtems_packet_prefix **packet  
4 );
```

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.

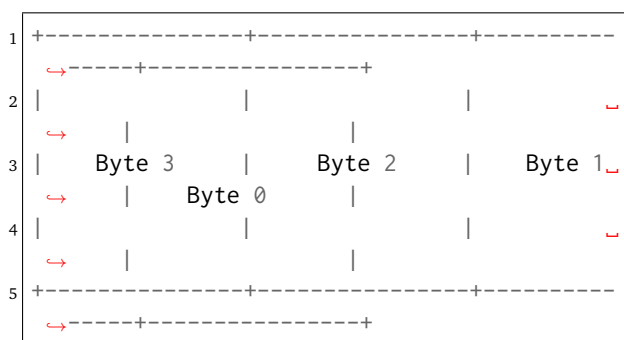
If node is set to zero, the packet is to be broadcasted to all other nodes in the system. Although some MPCPI layers will be built upon hardware which support a broadcast mechanism, others may be required to generate a copy of the packet for each node in the system.

Many MPCPI 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.

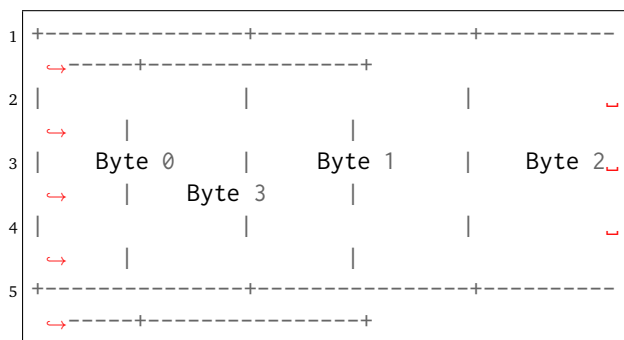
The to_convert field of the rtems_packet_prefix portion of the packet indicates how much of the packet in 32-bit units may require conversion in a heterogeneous system.

25.3.6 Supporting Heterogeneous Environments

Developing an MPCPI 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:



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:



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 MPCPI layer to perform all necessary conversion without burdening the developer with the details of the RTEMS packet format. As a result, the MPCPI 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 MPCPI layer.
- RTEMS makes no assumptions regarding the application data component of the packet.

25.4 Operations

25.4.1 Announcing a Packet

The `rtems_multiprocessing_announce` directive is called by the MPCIE 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.

25.5 Directives

This section details the additional directives required to support RTEMS in a multiprocessor configuration. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

25.5.1 MULTIPROCESSING_ANNOUNCE

- Announce the arrival of a packet

CALLING SEQUENCE:

```
1 void rtems_multiprocessing_announce( void_
  ↳);
```

DIRECTIVE STATUS CODES:

NONE

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 multiprocessor configurations.

NOTES:

This directive is typically called from an ISR.

This directive will almost certainly cause the calling task to be preempted.

This directive does not generate activity on remote nodes.

SYMMETRIC MULTIPROCESSING SERVICES

26.1 Introduction

The Symmetric Multiprocessing (SMP) support of the RTEMS 4.11.0 and later is available on

- ARM,
- PowerPC, and
- SPARC.

It must be explicitly enabled via the `--enable-smp` configure command line option. To enable SMP in the application configuration see *Chapter 24 Section 22.1 - Enable SMP Support for Applications* (page 354). The default scheduler for SMP applications supports up to 32 processors and is a global fixed priority scheduler, see also *Chapter 24 Section 21.9 - Configuring Clustered Schedulers* (page 351). For example applications see: `file:testsuites/smptests`.

Warning: The SMP support in the release of RTEMS is a work in progress. Before you start using this RTEMS version for SMP ask on the RTEMS mailing list.

This chapter describes the services related to Symmetric Multiprocessing provided by RTEMS.

The application level services currently provided are:

- *rtems_task_get_affinity* (page 395) - Get task processor affinity
- *rtems_task_set_affinity* (page 396) - Set task processor affinity
- *rtems_get_processor_count* (page 389) - Get processor count
- *rtems_get_current_processor* (page 390) - Get current processor index
- *rtems_scheduler_ident* (page 391) - Get ID of a scheduler
- *rtems_scheduler_get_processor_set* (page 392) - Get processor set of a scheduler
- *rtems_task_get_scheduler* (page 393) - Get scheduler of a task
- *rtems_task_set_scheduler* (page 394) - Set scheduler of a task

26.2 Background

26.2.1 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 interrupt service routines are allowed to nest. From a tasking viewpoint, it is the responsibility of the real-time operating 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.

26.2.2 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 arbitrary processor affinities uses a proof of concept implementation. See <https://devel.rtems.org/ticket/2510>.

26.2.3 Task Migration

With more than one processor in the system tasks can migrate from one processor to another. There are three reasons why tasks migrate in RTEMS.

- The scheduler changes explicitly via `rtems_task_set_scheduler()` 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 *Migratory Priority Inheritance* or the *Multiprocessor Resource Sharing Protocol*.

Task migration should be avoided so that the working set of a task can stay on the most local cache level.

The current implementation of task migration in RTEMS has some implications with respect to the interrupt latency. It is crucial to preserve the system invariant that a task can execute on at most one processor in the system at a time. This is accomplished with a boolean indicator in the task context. The processor architecture specific low-level task context switch code will mark that a task context is no longer executing and waits that the heir context stopped execution before it restores the heir context and resumes execution of the heir task. So there is one point in time in which a processor is without a task. This is essential to avoid cyclic dependencies in case multiple tasks migrate at once. Otherwise some supervising entity is necessary to prevent life-locks. Such a global supervisor would lead to scalability problems so this approach is not used. Currently the thread dispatch is performed with interrupts disabled. So in case the heir task is currently executing on another processor then this prolongs the time of disabled interrupts since one processor has to wait for another processor to make progress.

It is difficult to avoid this issue with the interrupt latency since interrupts normally store the context of the interrupted task on its stack. In case a task is marked as not executing we must not use its task stack to store such an interrupt context. We cannot use the heir stack before it stopped execution on another processor. So if we enable interrupts during this transition we have to provide an alternative task inde-

pendent stack for this time frame. This issue needs further investigation.

26.2.4 Clustered Scheduling

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.

Clustered scheduling helps to control the worst-case latencies in multi-processor systems, see *Brandenburg, Bjorn B.: Scheduling and Locking in Multiprocessor Real-Time Operating Systems. PhD thesis, 2011. <http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf>*. The goal is to reduce the amount of shared state in the system and thus prevention of lock contention. Modern multi-processor 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. It is easy to implement. 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 four means available

- events,
- message queues,
- semaphores using the *Chapter 12 Section 2.3 - Priority Inheritance* (page 169) protocol (priority boosting), and
- semaphores using the *Chapter 12 Section 2.5 - Multiprocessor Resource Sharing Protocol* (page 170) (MrsP).

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. This is work in progress.

For the configuration of clustered schedulers

see *Chapter 24 Section 21.9 - Configuring Clustered Schedulers* (page 351).

To set the scheduler of a task see `SCHEDULER_IDENT` - Get ID of a scheduler and *Chapter 26 Section 4.6 - TASK_SET_SCHEDULER - Set scheduler of a task* (page 394).

26.2.5 Task Priority Queues

Due to the support for clustered scheduling the task priority queues need special attention. It makes no sense to compare the priority values of two different scheduler instances. Thus, it is not possible to simply use one plain priority queue for tasks of different scheduler instances.

One solution to this problem is to use two levels of queues. The top level queue provides FIFO ordering and contains priority queues. Each priority queue is associated with a scheduler instance and contains only tasks of this scheduler instance. Tasks are enqueued in the priority queue corresponding to their scheduler instance. In case this priority queue was empty, then it is appended to the FIFO. To dequeue a task the highest priority task of the first priority queue in the FIFO is selected. Then the first priority queue is removed from the FIFO. In case the previously first priority queue is not empty, then it is appended to the FIFO. So there is FIFO fairness with respect to the highest priority task of each scheduler instances. See also *Brandenburg, Bjorn B.: 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), pages 292-302, 2013.*<http://www.mpi-sws.org/~bbb/papers/pdf/ecrts13b.pdf>.

Such a two level queue may need a considerable amount of memory if fast enqueue and dequeue operations are desired (depends on the scheduler instance count). To mitigate this problem an approach of the FreeBSD kernel was implemented in RTEMS. We have the invariant that a task can be enqueued on at most one task queue. Thus, we need only as many queues as we have tasks. Each task is equipped with spare task queue which it can give to an

object on demand. The task queue uses a dedicated memory space independent of the other memory used for the task itself. In case a task needs to block, then there are two options

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

In case the task is dequeued, then there are two options

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

Since there are usually more objects than tasks, this actually reduces the memory demands. In addition the objects contain only a pointer to the task queue structure. This helps to hide implementation details and makes it possible to use self-contained synchronization objects in Newlib and GCC (C++ and OpenMP run-time support).

26.2.6 Scheduler Helping Protocol

The scheduler provides a helping protocol to support locking protocols like *Migratory Priority Inheritance* or the *Multiprocessor Resource Sharing Protocol*. Each ready task can use at least one scheduler node at a time to gain access to a processor. Each scheduler node has an owner, a user and an optional idle task. The owner of a scheduler node is determined a task creation and never changes during the life time of a scheduler node. The user of a scheduler node may change due to the scheduler helping protocol. A scheduler node is in one of the four scheduler help states:

help yourself

This scheduler node is solely used by the

owner task. This task owns no resources using a helping protocol and thus does not take part in the scheduler helping protocol. No help will be provided for other tasks.

help active owner

This scheduler node is owned by a task actively owning a resource and can be used to help out tasks. In case this scheduler node changes its state from ready to scheduled and the task executes using another node, then an idle task will be provided as a user of this node to temporarily execute on behalf of the owner task. Thus lower priority tasks are denied access to the processors of this scheduler instance. In case a task actively owning a resource performs a blocking operation, then an idle task will be used also in case this node is in the scheduled state.

help active rival

This scheduler node is owned by a task actively obtaining a resource currently owned by another task and can be used to help out tasks. The task owning this node is ready and will give away its processor in case the task owning the resource asks for help.

help passive

This scheduler node is owned by a task obtaining a resource currently owned by another task and can be used to help out tasks. The task owning this node is blocked.

The following scheduler operations return a task in need for help

- unblock,
- change priority,
- yield, and
- ask for help.

A task in need for help is a task that encounters a scheduler state change from scheduled to ready (this is a pre-emption by a higher priority task) or a task that cannot be scheduled in an unblock operation. Such a task can ask tasks which depend on resources owned by this task for help.

In case it is not possible to schedule a task in need for help, then the scheduler nodes available for the task will be placed into the set

of ready scheduler nodes of the corresponding scheduler instances. Once a state change from ready to scheduled happens for one of scheduler nodes it will be used to schedule the task in need for help.

The ask for help scheduler operation is used to help tasks in need for help returned by the operations mentioned above. This operation is also used in case the root of a resource sub-tree owned by a task changes.

The run-time of the ask for help procedures depend on the size of the resource tree of the task needing help and other resource trees in case tasks in need for help are produced during this operation. Thus the worst-case latency in the system depends on the maximum resource tree size of the application.

26.2.7 Critical Section Techniques and SMP

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. This primarily means the use of binary semaphores or mutexes to implement critical sections.

26.2.7.1 Disable Interrupts and Interrupt Locks

A low overhead means to ensure mutual exclusion in uni-processor configurations is to disable interrupts around a critical section. This is commonly used in device driver code and throughout the operating system core. On 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_flush()`

are disabled on SMP configurations and its use will lead to compiler warnings and linker errors. In the unlikely case that interrupts must be disabled on the current processor, then the

- `rtems_interrupt_local_disable()`, and
- `rtems_interrupt_local_enable()`

macros are now available in all configurations.

Since disabling of interrupts is not enough to ensure system wide mutual exclusion on SMP, a new low-level synchronization primitive was added - the interrupt locks. They 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. On uni-processor configurations they degenerate to simple interrupt disable/enable sequences. 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.

```

1 void legacy_code_with_interrupt_disable
  ↪enable( void )
2 {
3     rtems_interrupt_level level;
4     rtems_interrupt_disable( level );
5     /* Some critical stuff */
6     rtems_interrupt_enable( level );
7 }
8
9 RTEMS_INTERRUPT_LOCK_DEFINE( static, lock, ↪
  ↪"Name" );
10
11 void smp_ready_code_with_interrupt_lock( ↪
  ↪void )
12 {
13     rtems_interrupt_lock_context lock_
  ↪context;
14     rtems_interrupt_lock_acquire( &lock, &
  ↪lock_context );
15     /* Some critical stuff */
16     rtems_interrupt_lock_release( &lock, &
  ↪lock_context );
17 }

```

The `rtems_interrupt_lock` structure is empty on uni-processor configurations.

Empty structures have a different size in C (implementation-defined, zero in case of GCC) and C++ (implementation-defined non-zero value, one in case of GCC). Thus the `RTEMS_INTERRUPT_LOCK_DECLARE()`, `RTEMS_INTERRUPT_LOCK_DEFINE()`, `RTEMS_INTERRUPT_LOCK_MEMBER()`, and `RTEMS_INTERRUPT_LOCK_REFERENCE()` macros are provided to ensure ABI compatibility.

26.2.7.2 Highest Priority Task Assumption

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 binary semaphore or 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.

26.2.7.3 Disable Preemption

On a uniprocessor system, disabling preemption in a task is very similar to making the highest priority task assumption. While preemption is disabled, no task context switches will occur unless the task initiates them voluntarily. And, just as with the highest priority task assumption, there are N-1 processors also running tasks. Thus the assumption that no other tasks will run while the task has preemption disabled is violated.

26.2.8 Task Unique Data and SMP

Per task variables are a service commonly provided by real-time operating systems for application use. They work by allowing the application to specify a location in memory (typically a void *) which is logically added to the context of a task. On each task switch, the location in memory is stored and each task can have a unique value in the same memory location. This memory location is directly accessed as a variable in a program.

This works well in a uniprocessor environment because there is one task executing and one memory location containing a task-specific value. But it is fundamentally broken on an SMP system because there are always N tasks executing. With only one location in memory, N-1 tasks will not have the correct value.

This paradigm for providing task unique data values is fundamentally broken on SMP systems.

26.2.8.1 Classic API Per Task Variables

The Classic API provides three directives to support per task variables. These are:

- `rtems_task_variable_add` - Associate per task variable
- `rtems_task_variable_get` - Obtain value of a per task variable
- `rtems_task_variable_delete` - Remove per task variable

As task variables are unsafe for use on SMP systems, the use of these services must be eliminated in all software that is to be used in an SMP environment. The task variables API is disabled on SMP. Its use will lead to compile-time and link-time errors. It is recommended that the application developer consider the use of POSIX Keys or Thread Local Storage (TLS). POSIX Keys are available in all RTEMS configurations. For the availability of TLS on a particular architecture please consult the *RTEMS CPU Architecture Supplement*.

The only remaining user of task variables in the RTEMS code base is the Ada support. So

basically Ada is not available on RTEMS SMP.

26.2.9 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.

```

1 #include <stdlib.h>
2 void __attribute__((constructor(1000))) _
   ↪ config_libgomp( void )
3 {
4     setenv( "OMP_DISPLAY_ENV", "VERBOSE", 1
   ↪ );
5     setenv( "GOMP_SPINCOUNT", "30000", 1 );
6     setenv( "GOMP_RTEMS_THREAD_POOLS", "1
   ↪ $2@SCHD", 1 );
7 }

```

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 ac-

ording 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 `I0`, `WRK0`, and `WRK1` with `GOMP_RTEMS_THREAD_POOLS` set to `"1@WRK0:3@WRK1"`. Then there are no thread pool restrictions for scheduler instance `I0`. 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.

26.2.10 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.

On SMP systems, scheduling decisions on one processor must be propagated to other processors through inter-processor interrupts. So, a thread dispatch which must be carried out on another processor happens not instantaneous. 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
- an boolean flag indicating if a thread dispatch is necessary or not.

Updates of the heir thread and the thread dispatch necessary indicator are synchronized via explicit memory barriers without the use of locks. A thread can be an heir thread on at most one processor in the system. 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. The thread post-switch actions use a per-processor lock. This implementation turned out to be quite efficient and no lock contention was observed in the test suite.

The current implementation of thread dispatching has some implications with respect to the interrupt latency. It is crucial to preserve the system invariant that a thread can execute on at most one processor in the system at a time. This is accomplished with a boolean indicator in the thread context. The processor architecture specific context switch code will mark that a thread context is no longer executing and waits that the heir context stopped execution before it restores the heir context and resumes execution of the heir thread (the boolean indicator is basically a TTAS lock). So, there is one point in time in which a processor is without a thread. This is essential to avoid cyclic dependencies in case multiple threads migrate at once. Otherwise some supervising entity is necessary to prevent deadlocks. Such a global supervisor would lead to scalability problems so this approach is not used. Currently the context switch is performed with interrupts disabled. Thus in case the heir thread is currently executing on another processor, the time of disabled interrupts is prolonged since one processor has to wait for another processor to make progress.

It is difficult to avoid this issue with the interrupt latency since interrupts normally store the

context of the interrupted thread on its stack. In case a thread is marked as not executing, we must not use its thread stack to store such an interrupt context. We cannot use the heir stack before it stopped execution on another processor. If we enable interrupts during this transition, then we have to provide an alternative thread independent stack for interrupts in this time frame. This issue needs further investigation.

The problematic situation occurs in case we have a thread which executes with thread dispatching disabled and should execute on another processor (e.g. it is an heir thread on another processor). In this case the interrupts on this other processor are disabled until the thread enables thread dispatching and starts the thread dispatch sequence. The scheduler (an exception is the scheduler with thread processor affinity support) tries to avoid such a situation and checks if a new scheduled thread already executes on a processor. In case the assigned processor differs from the processor on which the thread already executes and this processor is a member of the processor set managed by this scheduler instance, it will re-assign the processors to keep the already executing thread in place. Therefore normal scheduler requests will not lead to such a situation. Explicit thread migration requests, however, can lead to this situation. Explicit thread migrations may occur due to the scheduler helping protocol or explicit scheduler instance changes. The situation can also be provoked by interrupts which suspend and resume threads multiple times and produce stale asynchronous thread dispatch requests in the system.

26.3 Operations

26.3.1 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`.

```
1 #include <rtems.h>
2 #include <assert.h>
3
4 void pin_to_processor(rtems_id task_id, int_
   ↪processor_index)
5 {
6     rtems_status_code sc;
7     cpu_set_t          cpuset;
8     CPU_ZERO(&cpuset);
9     CPU_SET(processor_index, &cpuset);
10    sc = rtems_task_set_affinity(task_id, ↪
   ↪sizeof(cpuset), &cpuset);
11    assert(sc == RTEMS_SUCCESSFUL);
12 }
```

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.

26.4 Directives

This section details the symmetric multiprocessing services. A subsection is dedicated to each of these services and describes the calling sequence, related constants, usage, and status codes.

26.4.1 GET_PROCESSOR_COUNT - Get processor count

CALLING SEQUENCE:

```
1 uint32_t rtems_get_processor_count(void);
```

DIRECTIVE STATUS CODES:

The count of processors in the system.

DESCRIPTION:

On uni-processor configurations a value of one will be returned.

On SMP configurations this returns the value of a global variable set during system initialization to indicate the count of utilized processors. The processor count depends on the physically or virtually available processors and application configuration. The value will always be less than or equal to the maximum count of application configured processors.

NOTES:

None.

26.4.2 GET_CURRENT_PROCESSOR - Get current processor index

CALLING SEQUENCE:

```
1 uint32_t      rtems_get_current_  
   ↪processor(void);
```

DIRECTIVE STATUS CODES:

The index of the current processor.

DESCRIPTION:

On uni-processor configurations a value of zero will be returned.

On SMP configurations an architecture specific method is used to obtain the index of the current processor in the system. The set of processor indices is the range of integers starting with zero up to the processor count minus one.

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.

NOTES:

None.

26.4.3 SCHEDULER_IDENT - Get ID of a scheduler

CALLING SEQUENCE:

```

1 rtems_status_code rtems_scheduler_ident(
2   rtems_name name,
3   rtems_id *id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	successful operation
RTEMS_ INVALID_ ADDRESS	id is NULL
RTEMS_ INVALID_ NAME	invalid scheduler name
RTEMS_ UNSATISFIED	a scheduler with this name exists, but the processor set of this scheduler is empty

DESCRIPTION:

Identifies a scheduler by its name. The scheduler name is determined by the scheduler configuration. See *Chapter 24 - Configuring a System* (page 307).

NOTES:

None.

26.4.4 SCHEDULER_GET_PROCESSOR_SET

- Get processor set of a scheduler

CALLING SEQUENCE:

```

1 rtems_status_code    rtems_scheduler_get_
  ↪processor_set(
2   rtems_id    scheduler_id,
3   size_t      cpusetsize,
4   cpu_set_t  *cpuset
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	successful operation
RTEMS_ INVALID_ ADDRESS	cpuset is NULL
RTEMS_ INVALID_ ID	invalid scheduler id
RTEMS_ INVALID_ NUMBER	the affinity set buffer is too small for set of processors owned by the scheduler

DESCRIPTION:

Returns the processor set owned by the scheduler in cpuset. A set bit in the processor set means that this processor is owned by the scheduler and a cleared bit means the opposite.

NOTES:

None.

26.4.5 TASK_GET_SCHEDULER - Get scheduler of a task

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_task_get_
  ↪ scheduler(
2   rtems_id task_id,
3   rtems_id *scheduler_id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successful operation
RTEMS_INVALID_ADDRESS	scheduler_id is NULL
RTEMS_INVALID_ID	invalid task id

DESCRIPTION:

Returns the scheduler identifier of a task identified by task_id in scheduler_id.

NOTES:

None.

26.4.6 TASK_SET_SCHEDULER - Set scheduler of a task

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_set_
  ↪ scheduler(
2   rtems_id task_id,
3   rtems_id scheduler_id
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successful operation
RTEMS_INVALID_ID	invalid task or scheduler id
RTEMS_INCORRECT_STATE	the task is in the wrong state to perform a scheduler change

DESCRIPTION:

Sets the scheduler of a task identified by `task_id` to the scheduler identified by `scheduler_id`. The scheduler of a task is initialized to the scheduler of the task that created it.

NOTES:

None.

EXAMPLE:

```

1 #include <rtems.h>
2 #include <assert.h>
3
4 void task(rtems_task_argument arg);
5
6 void example(void)
7 {
8   rtems_status_code sc;
9   rtems_id task_id;
10  rtems_id scheduler_id;
11  rtems_name scheduler_name;
12
13  scheduler_name = rtems_build_name('W',
  ↪ 'O', 'R', 'K');
14
15  sc = rtems_scheduler_ident(scheduler_
  ↪ name, &scheduler_id);
16  assert(sc == RTEMS_SUCCESSFUL);
17
18  sc = rtems_task_create(
19  ↪ rtems_build_name('T', 'A', 'S',
  ↪ 'K'),

```

```

20     1,
21     RTEMS_MINIMUM_STACK_SIZE,
22     RTEMS_DEFAULT_MODES,
23     RTEMS_DEFAULT_ATTRIBUTES,
24     &task_id
25   );
26   assert(sc == RTEMS_SUCCESSFUL);
27
28   sc = rtems_task_set_scheduler(task_id,
  ↪ scheduler_id);
29   assert(sc == RTEMS_SUCCESSFUL);
30
31   sc = rtems_task_start(task_id, task,
  ↪ 0);
32   assert(sc == RTEMS_SUCCESSFUL);
33 }

```

26.4.7 TASK_GET_AFFINITY - Get task processor affinity

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_get_affinity(
2   rtems_id id,
3   size_t cpusetsize,
4   cpu_set_t *cpuset
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successful operation
RTEMS_INVALID_ADDRESS	cpuset is NULL
RTEMS_INVALID_ID	invalid task id
RTEMS_INVALID_NUMBER	the affinity set buffer is too small for the current processor affinity set of the task

DESCRIPTION:

Returns the current processor affinity set of the task in cpuset. A set bit in the affinity set means that the task can execute on this processor and a cleared bit means the opposite.

NOTES:

None.

26.4.8 TASK_SET_AFFINITY - Set task processor affinity

CALLING SEQUENCE:

```

1 rtems_status_code rtems_task_set_affinity(
2   rtems_id      id,
3   size_t        cpusetsize,
4   const cpu_set_t *cpuset
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	successful operation
RTEMS_INVALID_ADDRESS	cpuset is NULL
RTEMS_INVALID_ID	invalid task id
RTEMS_INVALID_NUMBER	invalid processor affinity set

DESCRIPTION:

Sets the processor affinity set for the task specified by `cpuset`. A set bit in the affinity set means that the task can execute on this processor and a cleared bit means the opposite.

NOTES:

This function will not change the scheduler of the task. The intersection of the processor affinity set and the set of processors owned by the scheduler of the task must be non-empty. It is not an error if the processor affinity set contains processors that are not part of the set of processors owned by the scheduler instance of the task. A task will simply not run under normal circumstances on these processors since the scheduler ignores them. Some locking protocols may temporarily use processors that are not included in the processor affinity set of the task. It is also not an error if the processor affinity set contains processors that are not part of the system.

PCI LIBRARY

27.1 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 debugging 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`)

27.2 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 interrupt 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.

27.2.1 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

27.2.2 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.

27.2.2.1 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.

27.2.2.2 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.

27.2.2.3 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.

27.2.2.4 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`.

27.2.2.5 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

27.2.3 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".

27.2.3.1 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).

```

1 /* Configuration Space Access Read Routines
   ↳ */
2 extern int pci_cfg_r8(pci_dev_t dev, int ofs,
   ↳ uint8_t *data);
3 extern int pci_cfg_r16(pci_dev_t dev, int
   ↳ ofs, uint16_t *data);
4 extern int pci_cfg_r32(pci_dev_t dev, int
   ↳ ofs, uint32_t *data);
5
6 /* Configuration Space Access Write Routines
   ↳ */
7 extern int pci_cfg_w8(pci_dev_t dev, int ofs,
   ↳ uint8_t data);
8 extern int pci_cfg_w16(pci_dev_t dev, int
   ↳ ofs, uint16_t data);
9 extern int pci_cfg_w32(pci_dev_t dev, int
   ↳ ofs, uint32_t data);

```

27.2.3.2 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 402) for how addresses are translated.

```

1 /* Read a register over PCI I/O Space */
2 extern uint8_t pci_io_r8(uint32_t adr);
3 extern uint16_t pci_io_r16(uint32_t adr);
4 extern uint32_t pci_io_r32(uint32_t adr);
5

```

```

6 /* Write a register over PCI I/O Space */
7 extern void pci_io_w8(uint32_t adr, uint8_t
   ↳ data);
8 extern void pci_io_w16(uint32_t adr, uint16_t
   ↳ data);
9 extern void pci_io_w32(uint32_t adr, uint32_t
   ↳ data);

```

27.2.3.3 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.

```

1 static inline uint16_t pci_ld_le16(volatile
   ↳ uint16_t *addr);
2 static inline void pci_st_le16(volatile
   ↳ uint16_t *addr, uint16_t val);
3 static inline uint32_t pci_ld_le32(volatile
   ↳ uint32_t *addr);
4 static inline void pci_st_le32(volatile
   ↳ uint32_t *addr, uint32_t val);
5 static inline uint16_t pci_ld_be16(volatile
   ↳ uint16_t *addr);
6 static inline void pci_st_be16(volatile
   ↳ uint16_t *addr, uint16_t val);
7 static inline uint32_t pci_ld_be32(volatile
   ↳ uint32_t *addr);
8 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.

27.2.3.4 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.

```

1 /* Get Read/Write function for accessing a
   ↪ register over PCI Memory Space
2  * (non-inline functions).
3  *
4  * Arguments
5  * wr          0(Read), 1(Write)
6  * size        1(Byte), 2(Word), 4(Double
   ↪ Word)
7  * func        Where function pointer
   ↪ will be stored
8  * endian      PCI_LITTLE_ENDIAN or PCI_
   ↪ BIG_ENDIAN
9  * type        1(I/O), 3(REG over MEM),
   ↪ 4(CFG)
10 *
11 * Return
12 * 0           Found function
13 * others      No such function defined
   ↪ by host driver or BSP
14 */
15 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 endianness, this is mostly 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.

27.2.3.5 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.

```

1 /* Translate PCI address into CPU accessible
   ↪ address */
2 static inline int pci_pci2cpu(uint32_t
   ↪ *address, int type);
3
4 /* Translate CPU accessible address into PCI
   ↪ address (for DMA) */
5 static inline int pci_cpu2pci(uint32_t
   ↪ *address, int type);

```

27.2.4 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 controller, 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.

27.2.5 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.

STACK BOUNDS CHECKER

28.1 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_blow*
(page 409) - Has the Current Task Blown its Stack
- *rtems_stack_checker_report_usage*
(page 409) - Report Task Stack Usage

28.2 Background

28.2.1 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.

28.2.2 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.

28.3 Operations

28.3.1 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:

```
1 #define CONFIGURE_STACK_CHECKER_ENABLED
2 ...
3 #include <rtems/confdefs.h>
```

28.3.2 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_blowed` method. This method return `FALSE` if the task is operating within its stack bounds and has not damaged its pattern area.

28.3.3 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	

```
5 0x08010004 TA3 0x003e0c40 0x003e3047
   →9096 1104
6 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.

28.3.4 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
2 stack covers range 0x003e5750 - 0x003e7b57
   →(9224 bytes)
3 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.

28.4 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.

28.4.1 `STACK_CHECKER_IS_BLOWN` - Has Current Task Blown Its Stack

CALLING SEQUENCE:

```
1 bool rtems_stack_checker_is_blowed( 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.

28.4.2 `STACK_CHECKER_REPORT_USAGE` - Report Task Stack Usage

CALLING SEQUENCE:

```
1 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

CPU USAGE STATISTICS

29.1 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 416) - Report CPU Usage Statistics
- *rtems_cpu_usage_reset* (page 417) - Reset CPU Usage Statistics

29.2 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.

29.3 Operations

29.3.1 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:

```

1 +-----+
2 |CPU USAGE BY THREAD                                     |
3 +-----+
4 |ID          | NAME          | SECONDS  | PERCENT  |
5 +-----+-----+-----+-----+
6 |0x04010001 | IDLE         | 0        | 0.000   |
7 +-----+-----+-----+-----+
8 |0x08010002 | TA1         | 1203    | 0.748   |
9 +-----+-----+-----+-----+
10|0x08010003 | TA2         | 203     | 0.126   |
11+-----+-----+-----+-----+
12|0x08010004 | TA3         | 202     | 0.126   |
13+-----+-----+-----+-----+
14|TICKS SINCE LAST SYSTEM RESET:                       |
15|TOTAL UNITS:                                         |
16+-----+-----+-----+-----+

```

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 us-

age. 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.

29.3.2 Reset CPU Usage Statistics

Invoking the `rtems_cpu_usage_reset` routine resets the CPU usage statistics for all tasks in the system.

29.4 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.

29.4.1 `cpu_usage_report` - Report CPU Usage Statistics

CALLING SEQUENCE:

```
1 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.

29.4.2 `cpu_usage_reset` - Reset CPU Usage Statistics

CALLING SEQUENCE:

```
1 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. default state which is when zero period executions have occurred.

NOTES:

NONE

OBJECT SERVICES

30.1 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 object related services provided by RTEMS are:

- *build_id*
- *rtems_build_name* (page 424) - build object name from characters
- *rtems_object_get_classic_name* (page 425) - lookup name from Id
- *rtems_object_get_name* (page 426) - obtain object name as string
- *rtems_object_set_name* (page 427) - set object name
- *rtems_object_id_get_api* (page 428) - obtain API from Id
- *rtems_object_id_get_class* (page 429) - obtain class from Id
- *rtems_object_id_get_node* (page 430) - obtain node from Id
- *rtems_object_id_get_index* (page 431) - obtain index from Id
- *rtems_build_id* (page 432) - build object id from components
- *rtems_object_id_api_minimum* (page 433) - obtain minimum API value
- *rtems_object_id_api_maximum* (page 434) - obtain maximum API value
- *rtems_object_id_api_minimum_class* (page 437) - obtain minimum class value
- *rtems_object_id_api_maximum_class* (page 438) - obtain maximum class value
- *rtems_object_get_api_name* (page 439) - obtain API name
- *rtems_object_get_api_class_name* (page 440) - obtain class name
- *rtems_object_get_class_information* (page 441) - obtain class information

30.2 Background

30.2.1 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.

30.2.2 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.

30.2.3 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.

30.3 Operations

30.3.1 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.

```

1 void printObjectId(rtems_id id)
2 {
3     printf(
4         "API=%d Class=%d Node=%d Index=%d\n",
5         rtems_object_id_get_api(id),
6         rtems_object_id_get_class(id),
7         rtems_object_id_get_node(id),
8         rtems_object_id_get_index(id)
9     );
10 }

```

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.”

```

1 rtems_id nextObjectId(rtems_id id)
2 {
3     return rtems_build_id(
4         rtems_object_id_get_api(id),
5         rtems_object_id_get_class(id),
6         rtems_object_id_get_node(id),
7         rtems_object_id_get_index(id)
8         + 1
9     );
10 }

```

Note that this Id may not be valid in this system or associated with an allocated object.

30.3.2 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.”

```

1 void prettyPrintObjectId(rtems_id id)
2 {
3     int tmpAPI, tmpClass;
4
5     tmpAPI = rtems_object_id_get_api(id),
6     tmpClass = rtems_object_id_get_class(id),
7
8     printf(
9         "API=%s Class=%s Node=%d Index=%d\n",
10        rtems_object_get_api_name(tmpAPI),
11        rtems_object_get_api_class_
12        ↪name(tmpAPI, tmpClass),
13        rtems_object_id_get_node(id),
14        rtems_object_id_get_index(id)
15    );
16 }

```

30.4 Directives

30.4.1 BUILD_NAME - Build object name from characters

CALLING SEQUENCE:

```
1 rtems_name rtems_build_name(  
2     uint8_t c1,  
3     uint8_t c2,  
4     uint8_t c3,  
5     uint8_t c4  
6 );
```

DIRECTIVE STATUS CODES:

Returns a name constructed from the four characters.

DESCRIPTION:

This service takes the four characters provided as arguments and constructs a thirty-two bit object name with c1 in the most significant byte and c4 in the least significant byte.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.2 OBJECT_GET_CLASSIC_NAME - Lookup name from id

CALLING SEQUENCE:

```

1 rtems_status_code      rtems_object_get_
  ↳classic_name(
2   rtems_id      id,
3   rtems_name   *name
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_ SUCCESSFUL	name looked up successfully
RTEMS_INVALID_ ADDRESS	invalid name pointer
RTEMS_INVALID_ ID	invalid object id

DESCRIPTION:

This service looks up the name for the object id specified and, if found, places the result in *name.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.3 OBJECT_GET_NAME - Obtain object name as string

CALLING SEQUENCE:

```
1 char* rtems_object_get_name(  
2     rtems_id      id,  
3     size_t        length,  
4     char          *name  
5 );
```

DIRECTIVE STATUS CODES:

Returns a pointer to the name if successful or NULL otherwise.

DESCRIPTION:

This service looks up the name of the object specified by `id` and places it in the memory pointed to by `name`. Every attempt is made to return name as a printable string even if the object has the Classic API thirty-two bit style name.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.4 OBJECT_SET_NAME - Set object name

CALLING SEQUENCE:

```

1 rtems_status_code rtems_object_set_name(
2   rtems_id      id,
3   const char    *name
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	name looked up successfully
RTEMS_INVALID_ADDRESS	invalid name pointer
RTEMS_INVALID_ID	invalid object id

DESCRIPTION:

This service sets the name of `id` to that specified by the string located at `name`.

NOTES:

This directive is strictly local and does not impact task scheduling.

If the object specified by `id` is of a class that has a string name, this method 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 thirty-two bit integer style name, then the first four characters in `*name` will be used to construct the name. `name` to the RTEMS Workspace and allocate enough memory from the RTEMS Workspace to make a copy of the string

30.4.5 OBJECT_ID_GET_API - Obtain API from Id

CALLING SEQUENCE:

```
1 int rtems_object_id_get_api(  
2   rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

Returns the API portion of the object Id.

DESCRIPTION:

This directive returns the API portion of the provided object id.

NOTES:

This directive is strictly local and does not impact task scheduling.

This directive does NOT validate the id provided.

30.4.6 OBJECT_ID_GET_CLASS - Obtain Class from Id

CALLING SEQUENCE:

```
1 int rtems_object_id_get_class(  
2     rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

Returns the class portion of the object Id.

DESCRIPTION:

This directive returns the class portion of the provided object id.

NOTES:

This directive is strictly local and does not impact task scheduling.

This directive does NOT validate the id provided.

30.4.7 OBJECT_ID_GET_NODE - Obtain Node from Id

CALLING SEQUENCE:

```
1 int rtems_object_id_get_node(  
2   rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

Returns the node portion of the object Id.

DESCRIPTION:

This directive returns the node portion of the provided object id.

NOTES:

This directive is strictly local and does not impact task scheduling.

This directive does NOT validate the id provided.

30.4.8 OBJECT_ID_GET_INDEX - Obtain Index from Id

CALLING SEQUENCE:

```
1 int rtems_object_id_get_index(  
2     rtems_id id  
3 );
```

DIRECTIVE STATUS CODES:

Returns the index portion of the object Id.

DESCRIPTION:

This directive returns the index portion of the provided object id.

NOTES:

This directive is strictly local and does not impact task scheduling.

This directive does NOT validate the id provided.

30.4.9 BUILD_ID - Build Object Id From Components

CALLING SEQUENCE:

```
1 rtems_id rtems_build_id(  
2     int the_api,  
3     int the_class,  
4     int the_node,  
5     int the_index  
6 );
```

DIRECTIVE STATUS CODES:

Returns an object Id constructed from the provided arguments.

DESCRIPTION:

This service constructs an object Id from the provided the_api, the_class, the_node, and the_index.

NOTES:

This directive is strictly local and does not impact task scheduling.

This directive does NOT validate the arguments provided or the Object id returned.

30.4.10 OBJECT_ID_API_MINIMUM - Obtain Minimum API Value

CALLING SEQUENCE:

```
1 int rtems_object_id_api_minimum(void);
```

DIRECTIVE STATUS CODES:

Returns the minimum valid for the API portion of an object Id.

DESCRIPTION:

This service returns the minimum valid for the API portion of an object Id.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.11 OBJECT_ID_API_MAXIMUM - Obtain Maximum API Value

CALLING SEQUENCE:

```
1 int rtems_object_id_api_maximum(void);
```

DIRECTIVE STATUS CODES:

Returns the maximum valid for the API portion of an object Id.

DESCRIPTION:

This service returns the maximum valid for the API portion of an object Id.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.12 OBJECT_API_MINIMUM_CLASS

- Obtain Minimum Class Value

CALLING SEQUENCE:

```
1 int rtems_object_api_minimum_class(  
2     int api  
3 );
```

DIRECTIVE STATUS CODES:

If api is not valid, -1 is returned.

If successful, this service returns the minimum valid for the class portion of an object Id for the specified api.

DESCRIPTION:

This service returns the minimum valid for the class portion of an object Id for the specified api.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.13 OBJECT_API_MAXIMUM_CLASS - Obtain Maximum Class Value

CALLING SEQUENCE:

```
1 int rtems_object_api_maximum_class(  
2     int api  
3 );
```

DIRECTIVE STATUS CODES:

If api is not valid, -1 is returned.

If successful, this service returns the maximum valid for the class portion of an object Id for the specified api.

DESCRIPTION:

This service returns the maximum valid for the class portion of an object Id for the specified api.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.14 OBJECT_ID_API_MINIMUM_CLASS

- Obtain Minimum Class Value
for an API

CALLING SEQUENCE:

```
1 int rtems_object_get_id_api_minimum_class(  
2     int api  
3 );
```

DIRECTIVE STATUS CODES:

If api is not valid, -1 is returned.

If successful, this service returns the index corresponding to the first object class of the specified api.

DESCRIPTION:

This service returns the index for the first object class associated with the specified api.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.15 OBJECT_ID_API_MAXIMUM_CLASS

- Obtain Maximum Class Value
for an API

CALLING SEQUENCE:

```
1 int rtems_object_get_api_maximum_class(  
2     int api  
3 );
```

DIRECTIVE STATUS CODES:

If api is not valid, -1 is returned.

If successful, this service returns the index corresponding to the last object class of the specified api.

DESCRIPTION:

This service returns the index for the last object class associated with the specified api.

NOTES:

This directive is strictly local and does not impact task scheduling.

30.4.16 OBJECT_GET_API_NAME - Obtain API Name

CALLING SEQUENCE:

```
1 const char* rtems_object_get_api_name(  
2     int api  
3 );
```

DIRECTIVE STATUS CODES:

If api is not valid, the string "BAD API" is returned.

If successful, this service returns a pointer to a string containing the name of the specified api.

DESCRIPTION:

This service returns the name of the specified api.

NOTES:

This directive is strictly local and does not impact task scheduling.

The string returned is from constant space. Do not modify or free it.

30.4.17 OBJECT_GET_API_CLASS_NAME - Obtain Class Name

CALLING SEQUENCE:

```
1 const char *rtems_object_get_api_class_  
   ↪name(  
2     int the_api,  
3     int the_class  
4 );
```

DIRECTIVE STATUS CODES:

If the_api is not valid, the string "BAD API" is returned.

If the_class is not valid, the string "BAD CLASS" is returned.

If successful, this service returns a pointer to a string containing the name of the specified the_api / the_class pair.

DESCRIPTION:

This service returns the name of the object class indicated by the specified the_api and the_class.

NOTES:

This directive is strictly local and does not impact task scheduling.

The string returned is from constant space. Do not modify or free it.

30.4.18 OBJECT_GET_CLASS_INFORMATION - Obtain Class Information

CALLING SEQUENCE:

```

1 rtems_status_code rtems_object_get_class_
  ↳information(
2   int
  ↳the_api,
3   int
  ↳the_class,
4   rtems_object_api_class_information
  ↳*info
5 );

```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL	information obtained successfully
RTEMS_INVALID_ADDRESS	info is NULL
RTEMS_INVALID_NUMBER	invalid api or the_class

If successful, the structure located at `info` will be filled in with information about the specified `api` / `the_class` pairing.

DESCRIPTION:

This service returns information about the object class indicated by the specified `api` and `the_class`. This structure is defined as follows:

```

1 typedef struct {
2   rtems_id minimum_id;
3   rtems_id maximum_id;
4   int      maximum;
5   bool     auto_extend;
6   int      unallocated;
7 } rtems_object_api_class_information;

```

NOTES:

This directive is strictly local and does not impact task scheduling.

CHAINS

31.1 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 448) - initialize the chain with nodes
- *rtems_chain_initialize_empty* (page 449) - initialize the chain as empty
- *rtems_chain_is_null_node* (page 450) - Is the node NULL ?
- *rtems_chain_head* (page 451) - Return the chain's head
- *rtems_chain_tail* (page 452) - Return the chain's tail
- *rtems_chain_are_nodes_equal* (page 453) - Are the node's equal ?
- *rtems_chain_is_empty* (page 454) - Is the chain empty ?
- *rtems_chain_is_first* (page 455) - Is the Node the first in the chain ?
- *rtems_chain_is_last* (page 456) - Is the Node the last in the chain ?
- *rtems_chain_has_only_one_node* (page 457) - Does the node have one node ?
- *rtems_chain_node_count_unprotected* (page 458) - Returns the node count of the chain (unprotected)
- *rtems_chain_is_head* (page 459) - Is the node the head ?
- *rtems_chain_is_tail* (page 460) - Is the node the tail ?
- *rtems_chain_extract* (page 461) - Extract the node from the chain
- *rtems_chain_extract_unprotected* (page 462) - Extract the node from the chain (unprotected)
- *rtems_chain_get* (page 463) - Return the first node on the chain
- *rtems_chain_get_unprotected* (page 464) - Return the first node on the chain (unprotected)
- *rtems_chain_insert* (page 465) - Insert the node into the chain
- *rtems_chain_insert_unprotected* (page 466) - Insert the node into the chain (unprotected)
- *rtems_chain_append* (page 467) - Append the node to chain
- *rtems_chain_append_unprotected* (page 468) - Append the node to chain (unprotected)
- *rtems_chain_prepend* (page 469) - Prepend the node to the end of the chain
- *rtems_chain_prepend_unprotected* (page 470) - Prepend the node to chain (unprotected)

31.2 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 traverse 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.

31.2.1 Nodes

A chain is made up from nodes that originate 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:

```

1 typedef struct foo
2 {
3     rtems_chain_node node;
4     int bar;
5 } foo;

```

creates a type `foo` that can be placed on a chain. To get the `foo` structure from the list you perform the following:

```

1 foo* get_foo(rtems_chain_control* control)
2 {
3     return (foo*) rtems_chain_get(control);
4 }

```

The node is placed at the start of the user's structure to allow the node address on the chain to be easily cast to the user's structure address.

31.2.2 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.

```

1 [Control]
2 first ----->
3 permanent_null <----- [NODE]
4 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.

31.3 Operations

31.3.1 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.

31.3.2 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:

```

1 typedef struct foo
2 {
3     rtems_chain_node node;
4     uint8_t char* data;
5 } foo;
6 rtems_chain_control chain;

```

Add nodes with the following code:

```

1 rtems_chain_initialize_empty (&chain);
2
3 for (i = 0; i < count; i++)
4 {
5     foo* bar = malloc (sizeof (foo));
6     if (!bar)
7         return -1;
8     bar->data = malloc (size);
9     rtems_chain_append (&chain, &bar->node);
10 }

```

The chain is initialized and the nodes allocated and appended to the chain. This is an example of a pool of buffers.

31.3.3 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:

```

1 void foobar (const char* match,
2             rtems_chain_control* chain,
3             rtems_chain_control* out)
4 {
5     rtems_chain_node* node;
6     foo* bar;
7
8     rtems_chain_initialize_empty (out);
9
10    node = chain->first;
11    while (!rtems_chain_is_tail (chain, node))
12    {
13        bar = (foo*) node;
14        rtems_chain_node* next_node = node->
15        next;
16        if (strcmp (match, bar->data) == 0)
17        {
18            rtems_chain_extract (node);
19            rtems_chain_append (out, node);
20        }
21        node = next_node;
22    }

```

31.4 Directives

The section details the Chains directives.

31.4.1 Initialize Chain With Nodes

CALLING SEQUENCE:

```
1 void rtems_chain_initialize(  
2     rtems_chain_control *the_chain,  
3     void                *starting_address,  
4     size_t              number_nodes,  
5     size_t              node_size  
6 )
```

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 initialize any user data on each node.

31.4.2 Initialize Empty

CALLING SEQUENCE:

```
1 void rtems_chain_initialize_empty(  
2     rtems_chain_control *the_chain  
3 );
```

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.

31.4.3 Is Null Node ?

CALLING SEQUENCE:

```
1 bool rtems_chain_is_null_node(  
2     const rtems_chain_node *the_node  
3 );
```

RETURNS:

Returns true if the node pointer 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.

31.4.4 Head

CALLING SEQUENCE:

```
1 rtems_chain_node *rtems_chain_head(  
2     rtems_chain_control *the_chain  
3 )
```

RETURNS:

Returns the permanent head node of the chain.

DESCRIPTION:

This function returns a pointer to the first node on the chain.

31.4.5 Tail

CALLING SEQUENCE:

```
1 rtems_chain_node *rtems_chain_tail(  
2     rtems_chain_control *the_chain  
3 );
```

RETURNS:

Returns the permanent tail node of the chain.

DESCRIPTION:

This function returns a pointer to the last node on the chain.

31.4.6 Are Two Nodes Equal ?

CALLING SEQUENCE:

```
1 bool rtems_chain_are_nodes_equal(  
2     const rtems_chain_node *left,  
3     const rtems_chain_node *right  
4 );
```

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.

31.4.7 Is the Chain Empty

CALLING SEQUENCE:

```
1 bool rtems_chain_is_empty(  
2     rtems_chain_control *the_chain  
3 );
```

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.

31.4.8 Is this the First Node on the Chain ?

CALLING SEQUENCE:

```
1 bool rtems_chain_is_first(  
2     const rtems_chain_node *the_node  
3 );
```

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.

31.4.9 Is this the Last Node on the Chain ?

CALLING SEQUENCE:

```
1 bool rtems_chain_is_last(  
2     const rtems_chain_node *the_node  
3 );
```

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.

31.4.10 Does this Chain have only One Node ?

CALLING SEQUENCE:

```
1 bool rtems_chain_has_only_one_node(  
2     const rtems_chain_control *the_chain  
3 );
```

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.

31.4.11 Returns the node count of the chain (unprotected)

CALLING SEQUENCE:

```
1 size_t rtems_chain_node_count_unprotected(  
2     const rtems_chain_control *the_chain  
3 );
```

RETURNS:

This function returns the node count of the chain.

DESCRIPTION:

This function returns the node count of the chain.

31.4.12 Is this Node the Chain Head ?

CALLING SEQUENCE:

```
1 bool rtems_chain_is_head(  
2     rtems_chain_control *the_chain,  
3     rtems_const chain_node *the_node  
4 );
```

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.

31.4.13 Is this Node the Chain Tail ?

CALLING SEQUENCE:

```
1 bool rtems_chain_is_tail(  
2     rtems_chain_control *the_chain,  
3     const rtems_chain_node *the_node  
4 )
```

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.

31.4.14 Extract a Node

CALLING SEQUENCE:

```
1 void rtems_chain_extract(  
2     rtems_chain_node *the_node  
3 );
```

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.

31.4.15 Extract a Node (unprotected)

CALLING SEQUENCE:

```
1 void rtems_chain_extract_unprotected(  
2     rtems_chain_node *the_node  
3 );
```

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.

31.4.16 Get the First Node

CALLING SEQUENCE:

```
1 rtems_chain_node *rtems_chain_get(  
2     rtems_chain_control *the_chain  
3 );
```

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.

31.4.17 Get the First Node (unprotected)

CALLING SEQUENCE:

```
1 rtems_chain_node      *rtems_chain_get_  
  ↪unprotected(  
2   rtems_chain_control *the_chain  
3 );
```

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.

31.4.18 Insert a Node

CALLING SEQUENCE:

```
1 void rtems_chain_insert(  
2     rtems_chain_node *after_node,  
3     rtems_chain_node *the_node  
4 );
```

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.

31.4.19 Insert a Node (unprotected)

CALLING SEQUENCE:

```
1 void rtems_chain_insert_unprotected(  
2     rtems_chain_node *after_node,  
3     rtems_chain_node *the_node  
4 );
```

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.

31.4.20 Append a Node

CALLING SEQUENCE:

```
1 void rtems_chain_append(  
2     rtems_chain_control *the_chain,  
3     rtems_chain_node   *the_node  
4 );
```

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.

31.4.21 Append a Node (unprotected)

CALLING SEQUENCE:

```
1 void rtems_chain_append_unprotected(  
2     rtems_chain_control *the_chain,  
3     rtems_chain_node   *the_node  
4 );
```

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.

31.4.22 Prepend a Node

CALLING SEQUENCE:

```
1 void rtems_chain_prepend(  
2     rtems_chain_control *the_chain,  
3     rtems_chain_node   *the_node  
4 );
```

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.

31.4.23 Prepend a Node (unprotected)

CALLING SEQUENCE:

```
1 void rtems_chain_prepend_unprotected(  
2     rtems_chain_control *the_chain,  
3     rtems_chain_node   *the_node  
4 );
```

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.

RED-BLACK TREES

32.1 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_rbtrees_node` - Red-Black Tree node embedded in another struct
- `rtems_rtems_rbtrees_control` - Red-Black Tree control node for an entire tree
- `rtems_rtems_rbtrees_initialize` - initialize the red-black tree with nodes
- `rtems_rtems_rbtrees_initialize_empty` - initialize the red-black tree as empty
- `rtems_rtems_rbtrees_set_off_tree` - Clear a node's links
- `rtems_rtems_rbtrees_root` - Return the red-black tree's root node
- `rtems_rtems_rbtrees_min` - Return the red-black tree's minimum node
- `rtems_rtems_rbtrees_max` - Return the red-black tree's maximum node
- `rtems_rtems_rbtrees_left` - Return a node's left child node
- `rtems_rtems_rbtrees_right` - Return a node's right child node
- `rtems_rtems_rbtrees_parent` - Return a node's parent node
- `rtems_rtems_rbtrees_are_nodes_equal` - Are the node's equal ?
- `rtems_rtems_rbtrees_is_empty` - Is the red-black tree empty ?
- `rtems_rtems_rbtrees_is_min` - Is the Node the minimum in the red-black tree ?
- `rtems_rtems_rbtrees_is_max` - Is the Node the maximum in the red-black tree ?
- `rtems_rtems_rbtrees_is_root` - Is the Node the root of the red-black tree ?
- `rtems_rtems_rbtrees_find` - Find the node with a matching key in the red-black tree
- `rtems_rtems_rbtrees_predecessor` - Return the in-order predecessor of a node.
- `rtems_rtems_rbtrees_successor` - Return the in-order successor of a node.
- `rtems_rtems_rbtrees_extract` - Remove the node from the red-black tree
- `rtems_rtems_rbtrees_get_min` - Remove the minimum node from the red-black tree
- `rtems_rtems_rbtrees_get_max` - Remove the maximum node from the red-black tree
- `rtems_rtems_rbtrees_peek_min` - Returns the minimum node from the red-black tree
- `rtems_rtems_rbtrees_peek_max` - Returns the maximum node from the red-black tree
- `rtems_rtems_rbtrees_insert` - Add the node to the red-black tree

32.2 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.

32.2.1 Nodes

A red-black tree is made up from nodes that originate from a red-black tree control object. A node is of type `rtems_rtems_rbtrees_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.

32.2.2 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_rbtrees_control` structure is treated as a `rtems_rtems_rbtrees_node` structure with a NULL parent and left child pointing to the root.

32.3 Operations

Examples for using the red-black trees can be found in the testsuites/sptests/sprbtree01/init.c file.

32.4 Directives

32.4.1 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`.

TIMESPEC HELPERS

33.1 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 482) - Set timespec's value
- *rtems_timespec_zero* (page 483) - Zero timespec's value
- *rtems_timespec_is_valid* (page 484) - Check if timespec is valid
- *rtems_timespec_add_to* (page 485) - Add two timespecs
- *rtems_timespec_subtract* (page 486) - Subtract two timespecs
- *rtems_timespec_divide* (page 487) - Divide two timespecs
- *rtems_timespec_divide_by_integer* (page 488) - Divide timespec by integer
- *rtems_timespec_less_than* (page 489) - Less than operator
- *rtems_timespec_greater_than* (page 490) - Greater than operator
- *rtems_timespec_equal_to* (page 491) - Check if two timespecs are equal
- *rtems_timespec_get_seconds* (page 492) - Obtain seconds portion of timespec
- *rtems_timespec_get_nanoseconds* (page 493) - Obtain nanoseconds portion of timespec
- *rtems_timespec_to_ticks* (page 494) - Convert timespec to number of ticks
- *rtems_timespec_from_ticks* (page 495) - Convert ticks to timespec

33.2 Background

33.2.1 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 similar 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.

33.3 Operations

33.3.1 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.

33.3.2 Timespec Math

A user can perform multiple operations on `struct timespec` instances. The helpers in this manager assist in adding, subtracting, and performing division 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 timespec` 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.

33.3.3 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.

33.3.4 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.

33.4 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.

33.4.1 TIMESPEC_SET - Set struct timespec Instance

CALLING SEQUENCE:

```
1 void rtems_timespec_set(  
2     struct timespec *time,  
3     time_t          seconds,  
4     uint32_t        nanoseconds  
5 );
```

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.

33.4.2 TIMESPEC_ZERO - Zero struct timespec Instance

CALLING SEQUENCE:

```
1 void rtems_timespec_zero(  
2     struct timespec *time  
3 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This routine sets the contents of the struct timespec instance time to zero.

NOTES:

NONE

33.4.3 TIMESPEC_IS_VALID - Check validity of a struct timespec instance

CALLING SEQUENCE:

```
1 bool rtems_timespec_is_valid(  
2     const struct timespec *time  
3 );
```

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 instance is in allowed range (less than the maximum number of nanoseconds per second).

NOTES:

NONE

33.4.4 TIMESPEC_ADD_TO - Add Two struct timespec Instances

CALLING SEQUENCE:

```
1 uint32_t rtems_timespec_add_to(  
2     struct timespec *time,  
3     const struct timespec *add  
4 );
```

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

33.4.5 TIMESPEC_SUBTRACT - Subtract Two struct timespec Instances

CALLING SEQUENCE:

```
1 void rtems_timespec_subtract(  
2     const struct timespec *start,  
3     const struct timespec *end,  
4     struct timespec      *result  
5 );
```

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!

33.4.6 TIMESPEC_DIVIDE - Divide Two struct timespec Instances

CALLING SEQUENCE:

```
1 void rtems_timespec_divide(  
2     const struct timespec *lhs,  
3     const struct timespec *rhs,  
4     uint32_t               *ival_percentage,  
5     uint32_t               *fval_percentage  
6 );
```

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.

33.4.7 TIMESPEC_DIVIDE_BY_INTEGER

- Divide a struct timespec Instance
by an Integer

CALLING SEQUENCE:

```
1 int rtems_timespec_divide_by_integer(  
2     const struct timespec *time,  
3     uint32_t             iterations,  
4     struct timespec      *result  
5 );
```

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.

33.4.8 TIMESPEC_LESS_THAN - Less than operator

CALLING SEQUENCE:

```
1 bool rtems_timespec_less_than(  
2     const struct timespec *lhs,  
3     const struct timespec *rhs  
4 );
```

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

33.4.9 TIMESPEC_GREATER_THAN - Greater than operator

CALLING SEQUENCE:

```
1 bool rtems_timespec_greater_than(  
2     const struct timespec *_lhs,  
3     const struct timespec *_rhs  
4 );
```

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

33.4.10 TIMESPEC_EQUAL_TO - Check equality of timespecs

CALLING SEQUENCE:

```
1 bool rtems_timespec_equal_to(  
2     const struct timespec *lhs,  
3     const struct timespec *rhs  
4 );
```

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

33.4.11 TIMESPEC_GET_SECONDS - Get Seconds Portion of struct timespec Instance

CALLING SEQUENCE:

```
1 time_t rtems_timespec_get_seconds(  
2     struct timespec *time  
3 );
```

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

33.4.12 TIMESPEC_GET_NANOSECONDS

- Get Nanoseconds Portion of the struct timespec Instance

CALLING SEQUENCE:

```
1 uint32_t rtems_timespec_get_nanoseconds(  
2     struct timespec *_time  
3 );
```

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

33.4.13 TIMESPEC_TO_TICKS - Convert struct timespec Instance to Ticks

CALLING SEQUENCE:

```
1 uint32_t rtems_timespec_to_ticks(  
2     const struct timespec *time  
3 );
```

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

33.4.14 TIMESPEC_FROM_TICKS - Convert Ticks to struct timespec Representation

CALLING SEQUENCE:

```
1 void rtems_timespec_from_ticks(  
2     uint32_t      ticks,  
3     struct timespec *time  
4 );
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This routine converts the ticks to the corresponding struct timespec representation and stores it in time.

NOTES:

NONE

CONSTANT BANDWIDTH SERVER SCHEDULER API

34.1 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 504) - Initialize the CBS library
- *rtems_cbs_cleanup* (page 505) - Cleanup the CBS library
- *rtems_cbs_create_server* (page 506) - Create a new bandwidth server
- *rtems_cbs_attach_thread* (page 507) - Attach a thread to server
- *rtems_cbs_detach_thread* (page 508) - Detach a thread from server
- *rtems_cbs_destroy_server* (page 509) - Destroy a bandwidth server
- *rtems_cbs_get_server_id* (page 510) - Get an ID of a server
- *rtems_cbs_get_parameters* (page 511) - Get scheduling parameters of a server
- *rtems_cbs_set_parameters* (page 512) - Set scheduling parameters of a server
- *rtems_cbs_get_execution_time* (page 513) - Get elapsed execution time
- *rtems_cbs_get_remaining_budget* (page 514) - Get remainig execution time
- *rtems_cbs_get_approved_budget* (page 515) - Get scheduler approved execution time

34.2 Background

34.2.1 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 extension 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 type³ denoting an id of a server and `rtems_id` a type for id of tasks.

34.2.2 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`.

34.2.3 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 priority). 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
);
```

34.2.4 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 voluntarily 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 unblocking, you should be always sure that the ratio between remaining computation time and remaining deadline is not higher than the utilization you have agreed with the scheduler.

34.3 Operations

34.3.1 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.

34.3.2 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.

34.3.3 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.

34.3.4 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.

```

1 void overrun_handler (
2     rtems_cbs_server_id server_id
3 )
4 {
5     printk( "Budget overrun, fixing the
6     ↪task\n" );
7     return;
8 }
9 rtems_task Tasks_Periodic(
10    rtems_task_argument argument
11 )
12 {
13    rtems_id          rmid;
14    rtems_cbs_server_id server_id;
15    rtems_cbs_parameters params;
16
17    params.deadline = 10;
18    params.budget = 4;
19
20    rtems_cbs_initialize();
21    rtems_cbs_create_server( &params, &
22    ↪overrun_handler, &server_id )
23    rtems_cbs_attach_thread( server_id, SELF_
24    ↪);
25    rtems_rate_monotonic_create( argument, &
26    ↪rmid );
27
28    while ( 1 ) {
29        if (rtems_rate_monotonic_period(rmid,
30    ↪ params.deadline) == RTEMS_TIMEOUT)
31            break;
32        /* Perform some periodic action */
33    }
34
35    rtems_rate_monotonic_delete( rmid );
36    rtems_cbs_cleanup();
37    exit( 1 );

```

```
34 }
```

34.4 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.

34.4.1 CBS_INITIALIZE - Initialize the CBS library

CALLING SEQUENCE:

```
1 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.

34.4.2 CBS_CLEANUP - Cleanup the CBS library

CALLING SEQUENCE:

```
1 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.

34.4.3 CBS_CREATE_SERVER - Create a new bandwidth server

CALLING SEQUENCE:

```

1 int rtems_cbs_create_server (
2     rtems_cbs_parameters *params,
3     rtems_cbs_budget_overrun budget_
4     ↪overrun_callback,
5     rtems_cbs_server_id *server_id
6 );

```

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.

34.4.4 CBS_ATTACH_THREAD - Attach a thread to server

CALLING SEQUENCE:

```

1 int rtems_cbs_attach_thread (
2     rtems_cbs_server_id server_id,
3     rtems_id           task_id
4 );

```

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 pre-emptible.

34.4.5 CBS_DETACH_THREAD - Detach a thread from server

CALLING SEQUENCE:

```
1 int rtems_cbs_detach_thread (  
2     rtems_cbs_server_id server_id,  
3     rtems_id          task_id  
4 );
```

DIRECTIVE STATUS CODES:

RTEMS_CBS_OK	successfully detached
RTEMS_CBS_ERROR_INVALID_PARAMETER	invalid input argument
RTEMS_CBS_ERROR_NOSEVER	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.

34.4.6 CBS_DESTROY_SERVER - Destroy a bandwidth server

CALLING SEQUENCE:

```
1 int rtems_cbs_destroy_server (  
2     rtems_cbs_server_id server_id  
3 );
```

DIRECTIVE STATUS CODES:

RTEMS_CBS_OK	successfully destroyed
RTEMS_CBS_ERROR_INVALID_PARAMETER	invalid input argument
RTEMS_CBS_ERROR_NOSEVER	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.

34.4.7 CBS_GET_SERVER_ID - Get an ID of a server

CALLING SEQUENCE:

```
1 int rtems_cbs_get_server_id (  
2     rtems_id          task_id,  
3     rtems_cbs_server_id *server_id  
4 );
```

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.

34.4.8 CBS_GET_PARAMETERS - Get scheduling parameters of a server

CALLING SEQUENCE:

```
1 rtems_cbs_get_parameters (
2   rtems_cbs_server_id  server_id,
3   rtems_cbs_parameters *params
4 );
```

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.

34.4.9 CBS_SET_PARAMETERS - Set scheduling parameters

CALLING SEQUENCE:

```

1 int rtems_cbs_set_parameters (
2     rtems_cbs_server_id server_id,
3     rtems_cbs_parameters *params
4 );

```

DIRECTIVE STATUS CODES:

RTEMS_CBS_OK	successful
RTEMS_CBS_ERROR_INVALID_PARAMETER	invalid input argument
RTEMS_CBS_ERROR_NO_SERVER	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 immediately, 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.

34.4.10 CBS_GET_EXECUTION_TIME - Get elapsed execution time

CALLING SEQUENCE:

```

1 int rtems_cbs_get_execution_time (
2     rtems_cbs_server_id  server_id,
3     time_t               *exec_time,
4     time_t               *abs_time
5 );

```

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.

34.4.11 CBS_GET_REMAINING_BUDGET

- Get remaining execution time

CALLING SEQUENCE:

```
1 int rtems_cbs_get_remaining_budget (  
2     rtems_cbs_server_id server_id,  
3     time_t                *remaining_budget  
4 );
```

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.

34.4.12 CBS_GET_APPROVED_BUDGET - Get scheduler approved execution time

CALLING SEQUENCE:

```
1 int rtems_cbs_get_approved_budget (  
2     rtems_cbs_server_id server_id,  
3     time_t                *appr_budget  
4 );
```

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.

DIRECTIVE STATUS CODES

35.1 Introduction

The directive status code directives are:

- *rtems_status_text* (page 520) - Return the name for the status code

35.2 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
RTEMS_INTERNAL_ERROR	RTEMS inconsistency detected
RTEMS_NO_MEMORY	could not get enough memory

35.2.1 STATUS_TEXT - Returns the enumeration name for a status code

CALLING SEQUENCE:

```
1 const char *rtems_status_text(  
2     rtems_status_code code  
3 );
```

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.

LINKER SETS

36.1 Introduction

Linker sets are a flexible means to create arrays of items out of a set of object files at link-time. For example its 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 525) - Designator of the linker set begin marker
- *RTEMS_LINKER_SET_END* (page 526) - Designator of the linker set end marker
- *RTEMS_LINKER_SET_SIZE* (page 527) - The linker set size in characters
- *RTEMS_LINKER_ROSET_DECLARE* (page 528) - Declares a read-only linker set
- *RTEMS_LINKER_ROSET* (page 529) - Defines a read-only linker set
- *RTEMS_LINKER_ROSET_ITEM_DECLARE* (page 530) - Declares a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM_REFERENCE* (page 531) - References a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM* (page 532) - Defines a read-only linker set item
- *RTEMS_LINKER_ROSET_ITEM_ORDERED* (page 533) - Defines an ordered read-only linker set item
- *RTEMS_LINKER_RWSET_DECLARE* (page 534) - Declares a read-write linker set
- *RTEMS_LINKER_RWSET* (page 535) - Defines a read-write linker set
- *RTEMS_LINKER_RWSET_ITEM_DECLARE* (page 536) - Declares a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM_REFERENCE* (page 537) - References a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM* (page 538) - Defines a read-write linker set item
- *RTEMS_LINKER_RWSET_ITEM_ORDERED* (page 539) - Defines an ordered read-write linker set item

36.2 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 `ctrend.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 creates 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.

```

1 /* To be placed in a read-only memory region.
   ↳*/
2 .rtemsroset : {
3     KEEP (*(SORT(.rtemsroset.*)))
4 }
5 /* To be placed in a read-write memory region.
   ↳*/
6 .rtemswset : {
7     KEEP (*(SORT(.rtemswset.*)))
8 }

```

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? It can be used to ini-

tialize 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`.

36.3 Directives

36.3.1 RTEMS_LINKER_SET_BEGIN - Designator of the linker set begin marker

CALLING SEQUENCE:

```
1 volatile 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.

36.3.2 RTEMS_LINKER_SET_END - Designator of the linker set end marker

CALLING SEQUENCE:

```
1 volatile 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.

36.3.3 RTEMS_LINKER_SET_SIZE - The linker set size in characters

CALLING SEQUENCE:

```
1 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.

36.3.4 RTEMS_LINKER_ROSET_DECLARE

- Declares a read-only linker set

CALLING SEQUENCE:

```
1 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.

36.3.5 RTEMS_LINKER_ROSET - Defines a read-only linker set

CALLING SEQUENCE:

```
1 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.

36.3.6 RTEMS_LINKER_ROSET_ITEM_DECLARE

- Declares a read-only linker set item

CALLING SEQUENCE:

```
1 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`. 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.

36.3.7 RTEMS_LINKER_ROSET_ITEM_REFERENCE

- References a read-only linker set item

CALLING SEQUENCE:

```
1 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.

36.3.8 RTEMS_LINKER_ROSET_ITEM - Defines a read-only linker set item

CALLING SEQUENCE:

```
1 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.

36.3.9 RTEMS_LINKER_ROSET_ITEM_ORDERED

- Defines an ordered read-only linker set item

CALLING SEQUENCE:

```
1 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()`).

```
1 #include <rtems/linkersets.h>
2
3 typedef struct {
4     int foo;
5 } xyz_item;
6
7 /* The XYZ-order defines */
8 #define XYZ_ORDER_FIRST 0x00001000
9 #define XYZ_ORDER_AND_SO_ON 0x00002000
10
11 /* Defines an ordered XYZ-item */
12 #define XYZ_ITEM( item, order ) \
13     enum { xyz_##item = order - ↵
   ↵order }; \
14     RTEMS_LINKER_ROSET_ITEM_
   ↵ORDERED( \
15     xyz, const xyz_item *, ↵
   ↵item, order \
```

```
16     ) = { &item }
17
18 /* Example item */
19 static const xyz_item some_item = { 123 };
20 XYZ_ITEM( some_item, XYZ_ORDER_FIRST );
```

36.3.10 RTEMS_LINKER_RWSET_DECLARE

- Declares a read-write linker set

CALLING SEQUENCE:

```
1 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.

36.3.11 RTEMS_LINKER_RWSET - Defines a read-write linker set

CALLING SEQUENCE:

```
1 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.

36.3.12 RTEMS_LINKER_RWSET_ITEM_DECLARE

- Declares a read-write linker set item

CALLING SEQUENCE:

```
1 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`. 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.

36.3.13 RTEMS_LINKER_RWSET_ITEM_REFERENCE

- References a read-write linker set item

CALLING SEQUENCE:

```
1 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.

36.3.14 RTEMS_LINKER_RWSET_ITEM - Defines a read-write linker set item

CALLING SEQUENCE:

```
1 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.

36.3.15 RTEMS_LINKER_RWSET_ITEM_ORDERED

- Defines an ordered read-write linker set item

```

16         ) = { &item }
17 /* Example item */
18 static const xyz_item some_item = { 123 };
19 XYZ_ITEM( some_item, XYZ_ORDER_FIRST );

```

CALLING SEQUENCE:

```

1 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()`).

```

1 #include <rtems/linkersets.h>
2
3 typedef struct {
4     int foo;
5 } xyz_item;
6
7 /* The XYZ-order defines */
8 #define XYZ_ORDER_FIRST 0x00001000
9 #define XYZ_ORDER_AND_SO_ON 0x00002000
10
11 /* Defines an ordered XYZ-item */
12 #define XYZ_ITEM( item, order ) \
13     enum { xyz_##item = order -_
14     ↪order }; \
15     RTEMS_LINKER_RWSET_ITEM_
16     ↪ORDERED( \
17     ↪xyz, const xyz_item *,_
18     ↪item, order \

```


EXAMPLE APPLICATION

```

1  /*
2  * This file contains an example of a simple
   ↳RTEMS
3  * application. It instantiates the RTEMS
   ↳Configuration
4  * Information using confdef.h and contains
   ↳two tasks:
5  * a user initialization task and a simple
   ↳task.
6  */
7
8  #include <rtems.h>
9
10 rtems_task      user_application(rtems_task_
   ↳argument argument);
11
12 rtems_task init_task(
13     rtems_task_argument ignored
14 )
15 {
16     rtems_id      tid;
17     rtems_status_code status;
18     rtems_name    name;
19
20     name = rtems_build_name( 'A', 'P', 'P',
   ↳'1' )
21
22     status = rtems_task_create(
23         name, 1, RTEMS_MINIMUM_STACK_SIZE,
24         RTEMS_NO_PREEMPT, RTEMS_FLOATING_
   ↳POINT, &tid
25     );
26     if ( status != RTEMS_STATUS_SUCCESSFUL_
   ↳) {
27         printf( "rtems_task_create failed_
   ↳with status of %d.\n", status );
28         exit( 1 );
29     }
30
31     status = rtems_task_start( tid, user_
   ↳application, 0 );
32     if ( status != RTEMS_STATUS_SUCCESSFUL_
   ↳) {
33         printf( "rtems_task_start failed_
   ↳with status of %d.\n", status );
34         exit( 1 );
35     }
36
37     status = rtems_task_delete( SELF ); /
   ↳* should not return */
38
39     printf( "rtems_task_delete returned with_
   ↳status of %d.\n", status );
40     exit( 1 );
41 }
42
43 rtems_task      user_application(rtems_task_
   ↳argument argument)
44 {
45     /* application specific initialization_
   ↳goes here */
46     while ( 1 ) { /* infinite_
   ↳loop */
47         /* APPLICATION CODE GOES HERE
48          *
49          * This code will typically include_
   ↳at least one
50          * directive which causes the_
   ↳calling task to
51          * give up the processor.
52          */
53     }
54 }
55
56 /* The Console Driver supplies Standard I/O.
   ↳ */
57 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_
   ↳DRIVER
58 /* The Clock Driver supplies the clock tick.
   ↳ */
59 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_
   ↳DRIVER
60 #define CONFIGURE_MAXIMUM_TASKS 2
61 #define CONFIGURE_INIT_TASK_NAME    rtems_
   ↳build_name( 'E', 'X', 'A', 'M' )
62 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
63 #define CONFIGURE_INIT
64 #include <rtems/confdefs.h>

```


GLOSSARY

active

A term used to describe an object which has been created by an application.

aperiodic task

A task which must execute only at irregular intervals and has only a soft deadline.

application

In this document, software which makes use of RTEMS.

ASR

see Asynchronous Signal Routine.

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 *ISO/IEC 9899:2011*.

awakened

A term used to describe a task that has been unblocked and may be scheduled to the CPU.

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.

broadcast

To simultaneously send a message to a logical set of destinations.

BSP

see Board Support Package.

Board Support Package

A collection of device initialization and control routines specific to a particular type of board or collection of boards.

buffer

A fixed length block of memory allocated from a partition.

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.

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.

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:dfn: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

An acronym for Central Processing Unit.

critical section

A section of code which must be executed indivisibly.

CRT

An acronym for Cathode Ray Tube. Normally used in reference to the man-machine interface.

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.

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.

dormant

The state entered by a task after it is created and before it has been started.

Device Driver Table

A table which contains the entry points for each of the configured device drivers.

dual-ported

A term used to describe memory which can be accessed at two different addresses.

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.

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.

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.

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. On 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 refer 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.

external address

The address used to access dual-ported memory by all the nodes in a system which do not own the memory.

FIFO

An acronym for First In First Out.

First In First Out

A discipline for manipulating entries in a data structure.

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.

freed

A resource that has been released by the application to RTEMS.

Giant lock

The *Giant lock* is a recursive SMP lock protecting most parts of the operating system state. Virtually every operating system service must acquire and release the Giant lock during its operation.

global

An object that has been created with the GLOBAL attribute and exported to all nodes in a multiprocessor system.

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 work 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.

heterogeneous

A multiprocessor computer system composed of dissimilar processors.

homogeneous

A multiprocessor computer system composed of a single type of processor.

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.

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 Routine

An ISR is invoked by the CPU to process a pending interrupt.

I/O

An acronym for Input/Output.

ISR

An acronym for Interrupt Service Routine.

kernel

In this document, this term is used as a synonym for executive.

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.

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.

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.

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

An acronym for Multiprocessor Communications Interface Layer.

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 characteris-

tics 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.

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.

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.

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

An RTEMS object which is used to allocate and deallocate fixed size blocks of memory from an dynamically specified area of memory.

partition

Clusters with a cardinality of one are *partitions*.

Partition Control Block

A data structure associated with each partition used by RTEMS to manage that partition.

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.

posting

The act of sending an event, message, semaphore, or signal to a task.

preempt

The act of forcing a task to relinquish the processor and dispatching to another task.

priority

A mechanism used to represent the relative importance of an element in a set of items.

RTEMS uses priority to determine which task should execute.

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, the pseudo-interrupt priority.

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 task 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

An acronym for Partition Control Block.

PXCB

An acronym for Proxy Control Block.

quantum

The application defined unit of time in which the processor is allocated.

queue

Alternate term for message queue.

QCB

An acronym for Message Queue Control Block.

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-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.

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.

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.

return code

Also known as error code or return value.

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

A value returned by RTEMS directives to indicate the completion status of the directive.

RNCB

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.

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.

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.

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 *Chapter 24 - Configuring a System* (page 307).

segments

Variable sized memory blocks allocated from a region.

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.

SMCB

An acronym for Semaphore Control Block.

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.

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.

soft real-time system

A real-time system in which a missed deadline does not compromise the integrity of the

system.

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

Also known as error code or return value.

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.

target

The system on which the application will ultimately execute.

task

A logically complete thread of execution. It consists normally of a set of registers and a stack. The terms *task* and *thread* are synonym in RTEMS. The scheduler assigns processors to a subset of the ready tasks.

Task Control Block

A data structure associated with each task used by RTEMS to manage that task.

task migration

Task migration happens in case a task stops execution on one processor and resumes execution on another processor.

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

An acronym for Task Control Block.

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.

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.

timeslice

The application defined unit of time in which the processor is allocated.

TMCB

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.

user extensions

Software routines provided by the application to enhance the functionality of RTEMS.

User Extension Table

A table which contains the entry points for each user extensions.

User Initialization Tasks Table

A table which contains the information needed to create and start each of the user initialization tasks.

user-provided

Alternate term for user-supplied. This term is used to designate any software routines which must be written by the application designer.

user-supplied

Alternate term for user-provided. This term is used to designate any software routines which must be written by the application designer.

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.

yield

When a task voluntarily releases control of the processor.

- genindex
- search

INDEX

- /dev/null, 358
- /dev/zero, 358
- _Internal_errors_What_happened, 280
- <rtems/confdefs.h>, 308
- “CONFIGURE_MAXIMUM_FAKE_ADA_TASKS“, 361

- add memory, 246
- add memory to a region, 246
- announce, 282, 283
- announce arrival of package, 375
- announce fatal error, 282, 283
- ASR, 221
- ASR mode, 221
- ASR vs. ISR, 221
- asynchronous signal routine, 221
- attach a thread to server, 507

- barrier, 185
- binary semaphores, 167
- Board Support Packages, 287
- broadcast message to a queue, 207
- BSP, 288
- BSP_IDLE_TASK_BODY, 344
- BSP_IDLE_TASK_STACK_SIZE, 344
- BSP_INITIAL_EXTENSION, 345
- BSP_INTERRUPT_STACK_SIZE, 345
- BSP_MAXIMUM_DEVICES, 345
- BSP_ZERO_WORKSPACE_AUTOMATICALLY, 345
- BSPs, 287
- buffers, 229
- build object id from components, 432
- build object name, 424
- building, 62, 213, 221, 229, 239

- C Program Heap, 328
- cancel a period, 158
- cancel a timer, 136
- cbs, 497
- CBS limitations, 499

- CBS overrun handler, 499
- CBS parameters, 499
- CBS periodic tasks, 499
- chain append a node, 467
- chain append a node unprotected, 468
- chain extract a node, 461
- chain extract a node unprotected, 462
- chain get first node, 463, 464
- chain get head, 451
- chain get tail, 452
- chain initialize, 448
- chain initialize empty, 449
- chain insert a node, 465
- chain insert a node unprotected, 466
- chain is chain empty, 454
- chain is node null, 450
- chain is node the first, 455
- chain is node the head, 459
- chain is node the last, 456
- chain is node the tail, 460
- chain iterate, 446
- chain only one node, 457, 458
- chains, 443
- chare are nodes equal, 453
- cleanup the CBS library, 505
- clear C Program Heap, 330
- clear RTEMS Workspace, 330
- clock, 109
- clock get nanoseconds uptime, 128
- clock get uptime, 125
- clock get uptime interval, 126
- clock get uptime seconds, 127
- close a device, 273
- communication and synchronization, 27
- conclude current period, 160
- confdefs.h, 308
- configure message queue buffer memory, 333
- CONFIGURE_APPLICATION_DISABLE_FILESYSTEM, 336
- CONFIGURE_APPLICATION_DOES_NOT_

- NEED_CLOCK_DRIVER, 356
- CONFIGURE_APPLICATION_EXTRA_DRIVERS, 357
- CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER, 355
- CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER, 355
- CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER, 357
- CONFIGURE_APPLICATION_NEEDS_LIBBLOCK, 341
- CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER, 358
- CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER, 356
- CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER, 357
- CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER, 356
- CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER, 356
- CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER, 358
- CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS, 357
- CONFIGURE_BDBUF_BUFFER_MAX_SIZE, 341
- CONFIGURE_BDBUF_BUFFER_MIN_SIZE, 341
- CONFIGURE_BDBUF_CACHE_MEMORY_SIZE, 341
- CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS, 342
- CONFIGURE_BDBUF_MAX_WRITE_BLOCKS, 342
- CONFIGURE_BDBUF_READ_AHEAD_TASK_PRIORITY, 343
- CONFIGURE_BDBUF_TASK_STACK_SIZE, 342
- CONFIGURE_BSP_PREREQUISITE_DRIVERS, 346
- CONFIGURE_DISABLE_BSP_SETTINGS, 344
- CONFIGURE_ENABLE_CLASSIC_API_NOTEPADS, 317
- CONFIGURE_ENABLE_GO, 363
- CONFIGURE_EXTRA_TASK_STACKS, 330
- CONFIGURE_GNAT RTEMS, 361
- CONFIGURE_HAS_OWN_CONFIGURATION_TABLE, 334
- CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE, 358
- CONFIGURE_HAS_OWN_INIT_TASK_TABLE, 322
- CONFIGURE_HAS_OWN_MOUNT_TABLE, 336
- CONFIGURE_HAS_OWN_MULTIPROCESSING_TABLE, 360
- CONFIGURE_IDLE_TASK_BODY, 347
- CONFIGURE_IDLE_TASK_INITIALIZES_APPLICATION, 347
- CONFIGURE_IDLE_TASK_STACK_SIZE, 347
- CONFIGURE_IMFS_DISABLE_CHMOD, 338
- CONFIGURE_IMFS_DISABLE_CHOWN, 338
- CONFIGURE_IMFS_DISABLE_LINK, 338
- CONFIGURE_IMFS_DISABLE_MKNOD, 339
- CONFIGURE_IMFS_DISABLE_MKNOD_FILE, 340
- CONFIGURE_IMFS_DISABLE_MOUNT, 339
- CONFIGURE_IMFS_DISABLE_READDIR, 339
- CONFIGURE_IMFS_DISABLE_READLINK, 338
- CONFIGURE_IMFS_DISABLE_RENAME, 339
- CONFIGURE_IMFS_DISABLE_RMNOD, 340
- CONFIGURE_IMFS_DISABLE_SYMLINK, 338
- CONFIGURE_IMFS_DISABLE_UNMOUNT, 339
- CONFIGURE_IMFS_DISABLE_UTIME, 338
- CONFIGURE_IMFS_MEMFILE_BYTES_PER_BLOCK, 337
- CONFIGURE_INIT_TASK_ARGUMENTS, 321
- CONFIGURE_INIT_TASK_ATTRIBUTES, 321
- CONFIGURE_INIT_TASK_ENTRY_POINT, 320
- CONFIGURE_INIT_TASK_INITIAL_MODES, 321
- CONFIGURE_INIT_TASK_NAME, 320
- CONFIGURE_INIT_TASK_PRIORITY, 321
- CONFIGURE_INIT_TASK_STACK_SIZE, 320
- CONFIGURE_INITIAL_EXTENSIONS, 331
- CONFIGURE_INTERRUPT_STACK_SIZE, 329
- CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS, 335
- CONFIGURE_MALLOC_BSP_SUPPORTS_SBRK, 344
- CONFIGURE_MAXIMUM_ADA_TASKS, 361
- CONFIGURE_MAXIMUM_BARRIERS, 318
- CONFIGURE_MAXIMUM_DEVICES, 336
- CONFIGURE_MAXIMUM_DRIVERS, 355
- CONFIGURE_MAXIMUM_FAKE_ADA_TASKS, 361
- CONFIGURE_MAXIMUM_GO_CHANNELS, 363
- CONFIGURE_MAXIMUM_GOROUTINES, 363
- CONFIGURE_MAXIMUM_MESSAGE_QUEUES, 318

- CONFIGURE_MAXIMUM_MRSP_SEMAPHORES, 318
- CONFIGURE_MAXIMUM_PARTITIONS, 319
- CONFIGURE_MAXIMUM_PERIODS, 318
- CONFIGURE_MAXIMUM_PORTS, 319
- CONFIGURE_MAXIMUM_POSIX_BARRIERS, 325
- CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES, 323
- CONFIGURE_MAXIMUM_POSIX_KEYS, 323
- CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUE_DESCRIPTOR, 324
- CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES, 324
- CONFIGURE_MAXIMUM_POSIX_MUTEXES, 323
- CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS, 324
- CONFIGURE_MAXIMUM_POSIX_RWLOCKS, 325
- CONFIGURE_MAXIMUM_POSIX_SEMAPHORES, 325
- CONFIGURE_MAXIMUM_POSIX_SPINLOCKS, 325
- CONFIGURE_MAXIMUM_POSIX_THREADS, 323
- CONFIGURE_MAXIMUM_POSIX_TIMERS, 324
- CONFIGURE_MAXIMUM_PRIORITY, 329
- CONFIGURE_MAXIMUM_REGIONS, 319
- CONFIGURE_MAXIMUM_SEMAPHORES, 318
- CONFIGURE_MAXIMUM_TASKS, 317
- CONFIGURE_MAXIMUM_TIMERS, 317
- CONFIGURE_MAXIMUM_USER_EXTENSIONS, 319
- CONFIGURE_MEMORY_OVERHEAD, 334
- CONFIGURE_MESSAGE_BUFFER_MEMORY, 333
- CONFIGURE_MESSAGE_BUFFERS_FOR_QUEUE, 333
- CONFIGURE_MICROSECONDS_PER_TICK, 328
- CONFIGURE_MINIMUM_TASK_STACK_SIZE, 329
- CONFIGURE_MP_APPLICATION, 359
- CONFIGURE_MP_MAXIMUM_GLOBAL_OBJECTS, 359
- CONFIGURE_MP_MAXIMUM_NODES, 359
- CONFIGURE_MP_MAXIMUM_PROXIES, 360
- CONFIGURE_MP_MPCI_TABLE_POINTER, 360
- CONFIGURE_MP_NODE_NUMBER, 359
- CONFIGURE_NUMBER_OF_TERMIOS_PORTS, 335
- CONFIGURE_POSIX_HAS_OWN_INIT_THREAD_TABLE, 327
- CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT, 326
- CONFIGURE_POSIX_INIT_THREAD_STACK_SIZE, 326
- CONFIGURE_POSIX_INIT_THREAD_TABLE, 326
- CONFIGURE_RTEMS_INIT_TASKS_TABLE, 320
- CONFIGURE_SCHEDULER_CBS, 349
- CONFIGURE_SCHEDULER_EDF, 348
- CONFIGURE_SCHEDULER_NAME, 350
- CONFIGURE_SCHEDULER_PRIORITY, 348
- CONFIGURE_SCHEDULER_PRIORITY_SMP, 349
- CONFIGURE_SCHEDULER_SIMPLE, 348
- CONFIGURE_SCHEDULER_SIMPLE_SMP, 349
- CONFIGURE_SCHEDULER_USER, 350
- CONFIGURE_SMP_APPLICATION, 354
- CONFIGURE_SMP_MAXIMUM_PROCESSORS, 354
- CONFIGURE_STACK_CHECKER_ENABLED, 330
- CONFIGURE_SWAPOUT_BLOCK_HOLD, 342
- CONFIGURE_SWAPOUT_SWAP_PERIOD, 341
- CONFIGURE_SWAPOUT_TASK_PRIORITY, 342
- CONFIGURE_SWAPOUT_WORKER_TASK_PRIORITY, 343
- CONFIGURE_SWAPOUT_WORKER_TASKS, 343
- CONFIGURE_TASK_STACK_ALLOCATOR, 332
- CONFIGURE_TASK_STACK_ALLOCATOR_INIT, 332
- CONFIGURE_TASK_STACK_DEALLOCATOR, 332
- CONFIGURE_TERMIOS_DISABLED, 335
- CONFIGURE_TICKS_PER_TIMESLICE, 328
- CONFIGURE_UNIFIED_WORK_AREAS, 328
- CONFIGURE_UNLIMITED_OBJECTS, 316
- CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM, 336
- CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM, 337
- CONFIGURE_ZERO_WORKSPACE_AUTOMATICALLY, 330
- configuring a system, 307

- constant bandwidth server scheduling, 40
- convert external to internal address, 259
- convert internal to external address, 260
- counting semaphores, 167
- create a barrier, 190
- create a message queue, 202
- create a new bandwidth server, 506
- create a partition, 232
- create a period, 156
- create a port, 256
- create a region, 243
- create a semaphore, 176
- create a task, 68
- create a timer, 134
- create an extension set, 304
- current task mode, 79
- current task priority, 78

- definition, 59, 148, 213, 229, 239, 253, 288, 367, 368, 370
- delay a task for an interval, 82
- delay a task until a wall time, 83
- delays, 111
- delete a barrier, 192
- delete a message queue, 204
- delete a partition, 234
- delete a period, 159
- delete a port, 258
- delete a region, 245
- delete a semaphore, 179
- delete a timer, 137
- delete an extension set, 306
- deleting a task, 74
- destroy a bandwidth server, 509
- detach a thread from server, 508
- device driver interface, 264
- Device Driver Table, 263
- device drivers, 261
- device names, 263
- disable interrupts, 97, 100
- disabling interrupts, 91
- dispatching, 43
- dual ported memory, 251, 253

- earliest deadline first scheduling, 40
- enable interrupts, 98, 101
- establish an ASR, 225
- establish an ISR, 96
- event condition, 213
- event flag, 213
- event set, 213
- events, 211

- exception frame, 284
- extension set, 297
- external addresses, 253

- fatal error, 282, 283, 285, 286
- fatal error detection, 279
- fatal error processing, 279
- fatal error user extension, 279
- fatal errors, 277
- fire a task-based timer at wall time, 142
- fire a timer after an interval, 138
- fire a timer at wall time, 139
- fire task-based a timer after an interval, 141
- flash interrupts, 99
- floating point, 61
- flush a semaphore, 182
- flush messages on a queue, 210

- get an ID of a server, 510
- get buffer from partition, 235
- get class from object ID, 26
- get current ticks counter value, 121
- get elapsed execution time, 513
- get ID of a barrier, 191
- get ID of a message queue, 203
- get ID of a partition, 233
- get ID of a period, 157
- get ID of a port, 257
- get ID of a region, 244
- get ID of a semaphore, 178
- get ID of a task, 70
- get ID of an extension set, 305
- get index from object ID, 26
- get name from id, 425
- get node from object ID, 26
- get number of pending messages, 209
- get object name as string, 426
- get per-task variable, 86
- get remaining execution time, 514
- get scheduler approved execution time, 515
- get scheduling parameters of a server, 511
- get segment from region, 247
- get size of segment, 249
- get statistics of period, 162
- get status of period, 161
- get task mode, 79
- get task notepad entry, 80
- get task preemption mode, 79
- get task priority, 78
- global objects, 367
- global objects table, 367

- heterogeneous multiprocessing, 372
- initialization tasks, 49
- initialize a device driver, 269
- initialize RTEMS, 54
- initialize the CBS library, 504
- initiate the Timer Server, 140
- install an ASR, 225
- install an ISR, 96
- int16_t, 34
- int32_t, 34
- int64_t, 34
- int8_t, 34
- internal addresses, 253
- interrupt level, 60
- interrupt levels, 91
- interrupt processing, 91
- interrupt stack size, 329
- IO Control, 276
- IO Manager, 261
- is interrupt in progress, 107
- is task suspended, 77
- ISR vs. ASR, 221
- iterate over all threads, 84
- libpci, 397
- linkersets, 521
- lock a barrier, 193
- lock a semaphore, 180
- lookup device major and minor number, 271
- major device number, 263
- manual round robin, 42
- maximum file descriptors, 335
- maximum priority, 329
- memory for a single message queue's buffers, 333
- memory for task tasks, 330
- memory management, 29
- message queue attributes, 197
- message queues, 195
- messages, 195
- minimum task stack size, 329
- minor device number, 263
- MPCI, 370
- MPCI and remote operations, 368
- MPCI entry points, 370
- multiprocessing, 365
- multiprocessing topologies, 367
- mutual exclusion, 167
- nodes, 367
- number of priority levels, 329
- object ID, 25
- object ID composition, 25
- object manipulation, 419
- object name, 25
- objects, 25
- obtain a barrier, 193
- obtain a semaphore, 180
- obtain API from id, 428
- obtain API name, 439
- obtain buffer from partition, 235
- obtain class from object id, 429
- obtain class information, 441
- obtain class name, 440
- obtain ID of a barrier, 191
- obtain ID of a partition, 233
- obtain ID of a period, 157
- obtain ID of a port, 257
- obtain ID of a region, 244
- obtain ID of a semaphore, 178
- obtain ID of an extension set, 305
- obtain ID of caller, 71
- obtain index from object id, 431
- obtain maximum API value, 434
- obtain maximum class value, 436
- obtain maximum class value for an API, 438
- obtain minimum API value, 433
- obtain minimum class value, 435
- obtain minimum class value for an API, 437
- obtain name from id, 425
- obtain node from object id, 430
- obtain object name as string, 426
- obtain per-task variable, 86
- obtain seconds since epoch, 119, 120
- obtain statistics of period, 162
- obtain status of period, 161
- obtain task mode, 79
- obtain task priority, 78
- obtain the ID of a timer, 135
- obtain the time of day, 116–118
- obtain ticks since boot, 121
- obtaining class from object ID, 26
- obtaining index from object ID, 26
- obtaining node from object ID, 26
- open a device, 272
- partition, 229
- partition attribute set, 229
- partitions, 227
- PCI_LIB_AUTO, 362
- PCI_LIB_PERIPHERAL, 362
- PCI_LIB_READ, 362
- PCI_LIB_STATIC, 362

- per task variables, 61
- per-task variable, 85, 87
- period initiation, 160
- period statistics report, 165
- periodic task, 148
- periodic tasks, 145
- ports, 251
- preemption, 42, 60
- prepend node, 469
- prepend node unprotected, 470
- print period statistics report, 165
- priority, 59
- priority scheduling, 39
- proxy, 368
- put message at front of queue, 206

- rate mononitonic tasks, 145
- Rate Monotonic Scheduling Algorithm, 148
- rbtree doc, 475
- rbtrees, 471
- read from a device, 274
- receive event condition, 217
- receive message from a queue, 208
- region, 239, 246
- region attribute set, 239
- regions, 237
- register a device driver, 267
- register device, 270
- release a barrier, 194
- release a semaphore, 181
- reset a timer, 143
- reset statistics of all periods, 164
- reset statistics of period, 163
- resize segment, 250
- restarting a task, 73
- resuming a task, 76
- return buffer to partition, 236
- return segment to region, 248
- RMS Algorithm, 148
- RMS First Deadline Rule, 150
- RMS Processor Utilization Rule, 149
- RMS schedulability analysis, 149
- round robin scheduling, 42
- rtems extensions table index, 298
- RTEMS Workspace, 328
- rtems_address, 33
- rtems_asr, 33, 223, 499
- rtems_asr_entry, 33
- rtems_attribute, 33
- rtems_barrier_create, 190
- rtems_barrier_delete, 192
- rtems_barrier_ident, 191
- rtems_barrier_release, 194
- rtems_barrier_wait, 193
- rtems_boolean, 33
- rtems_build_id, 432
- rtems_build_name, 424
- rtems_cbs_attach_thread, 507
- rtems_cbs_cleanup, 505
- rtems_cbs_create_server, 506
- rtems_cbs_destroy_server, 509
- rtems_cbs_detach_thread, 508
- rtems_cbs_get_approved_budget, 515
- rtems_cbs_get_execution_time, 513
- rtems_cbs_get_parameters, 511
- rtems_cbs_get_remaining_budget, 514
- rtems_cbs_get_server_id, 510
- rtems_cbs_initialize, 504
- rtems_cbs_parameters, 499
- rtems_cbs_set_parameters, 512
- rtems_chain_append, 467
- rtems_chain_append_unprotected, 468
- rtems_chain_are_nodes_equal, 453
- rtems_chain_extract, 461
- rtems_chain_extract_unprotected, 462
- rtems_chain_get, 463
- rtems_chain_get_unprotected, 464
- rtems_chain_has_only_one_node, 457
- rtems_chain_head, 451
- rtems_chain_initialize, 448
- rtems_chain_initialize_empty, 449
- rtems_chain_insert, 465
- rtems_chain_insert_unprotected, 466
- rtems_chain_is_empty, 454
- rtems_chain_is_first, 455
- rtems_chain_is_head, 459
- rtems_chain_is_last, 456
- rtems_chain_is_null_node, 450
- rtems_chain_is_tail, 460
- rtems_chain_node_count_unprotected, 458
- rtems_chain_prepend, 469
- rtems_chain_prepend_unprotected, 470
- rtems_chain_tail, 452
- rtems_clock_get, 116
- rtems_clock_get_options, 113, 116
- rtems_clock_get_seconds_since_epoch, 119
- rtems_clock_get_ticks_per_second, 120
- rtems_clock_get_ticks_since_boot, 121
- rtems_clock_get_tod, 117
- rtems_clock_get_tod_timeval, 118
- rtems_clock_get_uptime, 125
- rtems_clock_get_uptime_nanoseconds, 128
- rtems_clock_get_uptime_seconds, 127

rtems_clock_get_uptime_timeval, 126
 rtems_clock_set, 115
 rtems_clock_tick_before, 124
 rtems_clock_tick_later, 122
 rtems_clock_tick_later_usec, 123
 rtems_context, 33
 rtems_context_fp, 33
 rtems_device_driver, 33
 rtems_device_driver_entry, 33
 rtems_device_major_number, 33, 263
 rtems_device_minor_number, 33, 263
 rtems_double, 33
 rtems_event_receive, 217
 rtems_event_send, 216
 rtems_event_set, 33, 213
 rtems_exception_frame_print, 284
 rtems_extension, 33, 298
 rtems_extension_create, 304
 rtems_extension_delete, 306
 rtems_extension_ident, 305
 rtems_fatal, 283
 rtems_fatal_error_occurred, 282
 rtems_fatal_extension, 33, 300
 rtems_fatal_source_text, 285
 rtems_id, 25, 33
 rtems_initialize_executive, 54
 rtems_internal_error_text, 286
 rtems_interrupt_catch, 96
 rtems_interrupt_disable, 97
 rtems_interrupt_enable, 98
 rtems_interrupt_flash, 99
 rtems_interrupt_frame, 33
 rtems_interrupt_is_in_progress, 107
 rtems_interrupt_level, 33
 rtems_interrupt_local_disable, 100
 rtems_interrupt_local_enable, 101
 rtems_interrupt_lock_acquire, 103
 rtems_interrupt_lock_acquire_isr, 105
 rtems_interrupt_lock_initialize, 102
 rtems_interrupt_lock_release, 104
 rtems_interrupt_lock_release_isr, 106
 rtems_interval, 28, 33
 rtems_io_close, 273
 rtems_io_control, 276
 rtems_io_initialize, 269
 rtems_io_lookup_name, 271
 rtems_io_open, 272
 rtems_io_read, 274
 rtems_io_register_driver, 267
 rtems_io_register_name, 270
 rtems_io_unregister_driver, 268
 rtems_io_write, 275
 rtems_isr, 33, 91
 rtems_isr_entry, 33
 rtems_iterate_over_all_threads, 84
 RTEMS_LINKER_ROSET, 529
 RTEMS_LINKER_ROSET_DECLARE, 528
 RTEMS_LINKER_ROSET_ITEM, 532
 RTEMS_LINKER_ROSET_ITEM_DECLARE, 530
 RTEMS_LINKER_ROSET_ITEM_ORDERED, 533
 RTEMS_LINKER_ROSET_ITEM_REFERENCE, 531
 RTEMS_LINKER_RWSET, 535
 RTEMS_LINKER_RWSET_DECLARE, 534
 RTEMS_LINKER_RWSET_ITEM, 538
 RTEMS_LINKER_RWSET_ITEM_DECLARE, 536
 RTEMS_LINKER_RWSET_ITEM_ORDERED, 539
 RTEMS_LINKER_RWSET_ITEM_REFERENCE, 537
 RTEMS_LINKER_SET_BEGIN, 525
 RTEMS_LINKER_SET_END, 526
 RTEMS_LINKER_SET_SIZE, 527
 rtems_message_queue_broadcast, 207
 rtems_message_queue_create, 202
 rtems_message_queue_delete, 204
 rtems_message_queue_flush, 210
 rtems_message_queue_get_number_pending, 209
 rtems_message_queue_ident, 203
 rtems_message_queue_receive, 208
 rtems_message_queue_send, 205
 rtems_message_queue_urgent, 206
 rtems_mode, 33
 rtems_mp_packet_classes, 33
 rtems_mpci_entry, 34, 370
 rtems_mpci_get_packet_entry, 34
 rtems_mpci_initialization_entry, 34
 rtems_mpci_receive_packet_entry, 34
 rtems_mpci_return_packet_entry, 34
 rtems_mpci_send_packet_entry, 34
 rtems_mpci_table, 34
 rtems_multiprocessing_announce, 375
 rtems_name, 25, 34
 rtems_object_api_maximum_class, 436
 rtems_object_api_minimum_class, 435
 rtems_object_get_api_class_name, 440
 rtems_object_get_api_name, 439
 rtems_object_get_class_information, 441

rtems_object_get_classic_name, 425
 rtems_object_get_name, 25, 426
 rtems_object_id_api_maximum, 434
 rtems_object_id_api_maximum_class, 438
 rtems_object_id_api_minimum, 433
 rtems_object_id_api_minimum_class, 437
 rtems_object_id_get_api, 26, 428
 rtems_object_id_get_class, 26, 429
 rtems_object_id_get_index, 26, 431
 rtems_object_id_get_node, 26, 430
 rtems_object_set_name, 427
 rtems_option, 34
 rtems_packet_prefix, 34
 rtems_partition_create, 232
 rtems_partition_delete, 234
 rtems_partition_get_buffer, 235
 rtems_partition_ident, 233
 rtems_partition_return_buffer, 236
 rtems_port_create, 256
 rtems_port_delete, 258
 rtems_port_external_to_internal, 259
 rtems_port_ident, 257
 rtems_port_internal_to_external, 260
 rtems_rate_monotonic_cancel, 158
 rtems_rate_monotonic_create, 156
 rtems_rate_monotonic_delete, 159
 rtems_rate_monotonic_get_statistics, 162
 rtems_rate_monotonic_get_status, 161
 rtems_rate_monotonic_ident, 157
 rtems_rate_monotonic_period, 160
 rtems_rate_monotonic_period_statistics, 162,
 165
 rtems_rate_monotonic_period_status, 161
 rtems_rate_monotonic_report_statistics, 165
 rtems_rate_monotonic_reset_all_statistics,
 164
 rtems_rate_monotonic_reset_statistics, 163
 rtems_region_delete, 245
 rtems_region_extend, 246
 rtems_region_get_segment, 247
 rtems_region_get_segment_size, 249
 rtems_region_ident, 244
 rtems_region_resize_segment, 250
 rtems_region_return_segment, 248
 rtems_resource_is_unlimited, 316
 rtems_resource_maximum_per_allocation,
 316
 rtems_resource_unlimited, 315
 rtems_semaphore_create, 176
 rtems_semaphore_delete, 179
 rtems_semaphore_flush, 182
 rtems_semaphore_ident, 178
 rtems_semaphore_obtain, 180
 rtems_semaphore_release, 181
 rtems_semaphore_set_priority, 183
 rtems_shutdown_executive, 55
 rtems_signal_catch, 225
 rtems_signal_send, 226
 rtems_signal_set, 34, 221
 rtems_single, 34
 rtems_status_codes, 34
 rtems_status_text, 520
 rtems_task, 34, 60
 rtems_task_argument, 34
 rtems_task_begin_extension, 35, 299
 rtems_task_create, 68
 rtems_task_create_extension, 35, 298
 rtems_task_delete, 74
 rtems_task_delete_extension, 35, 299
 rtems_task_entry, 35
 rtems_task_exitted_extension, 35, 300
 rtems_task_get_note, 65, 80
 rtems_task_ident, 70
 rtems_task_is_suspended, 77
 rtems_task_mode, 60, 79
 rtems_task_priority, 35, 59
 rtems_task_restart, 73
 rtems_task_restart_extension, 35, 299
 rtems_task_resume, 76
 rtems_task_self, 71
 rtems_task_set_note, 65, 81
 rtems_task_set_priority, 78
 rtems_task_start, 72
 rtems_task_start_extension, 35, 298
 rtems_task_suspend, 75
 rtems_task_switch_extension, 35, 299
 rtems_task_variable_add, 85
 rtems_task_variable_delete, 87
 rtems_task_variable_get, 86
 rtems_task_wake_after, 82
 rtems_task_wake_when, 83
 rtems_tcb, 35
 rtems_time_of_day, 28, 35, 111
 rtems_timer_cancel, 136
 rtems_timer_create, 134
 rtems_timer_delete, 137
 rtems_timer_fire_after, 138
 rtems_timer_fire_when, 139
 rtems_timer_ident, 135
 rtems_timer_initiate_server, 140
 rtems_timer_reset, 143
 rtems_timer_server_fire_after, 141

- rtems_timer_server_fire_when, 142
- rtems_timer_service_routine, 35, 131
- rtems_timer_service_routine_entry, 35
- rtems_timespec_add_to, 485
- rtems_timespec_divide, 487
- rtems_timespec_divide_by_integer, 488
- rtems_timespec_equal_to, 491
- rtems_timespec_from_ticks, 495
- rtems_timespec_get_nanoseconds, 493
- rtems_timespec_get_seconds, 492
- rtems_timespec_greater_than, 490
- rtems_timespec_is_valid, 484
- rtems_timespec_less_than, 489
- rtems_timespec_set, 482
- rtems_timespec_subtract, 486
- rtems_timespec_to_ticks, 494
- rtems_timespec_zero, 483
- rtems_vector_number, 35, 91
- runtime driver registration, 263
- scheduling, 37
- scheduling algorithms, 39
- scheduling mechanisms, 42
- segment, 239
- semaphores, 167
- send event set to a task, 216
- send message to a queue, 205
- send signal set, 226
- separate work areas, 328
- set object name, 427
- set priority by scheduler for a semaphore, 183
- set scheduling parameters, 512
- set struct timespec instance, 482
- set task mode, 79
- set task notepad entry, 81
- set task preemption mode, 79
- set task priority, 78
- set the time of day, 115
- shutdown RTEMS, 55
- signal set, 221
- signals, 219
- special device services, 276
- sporadic task, 148
- stack, 405
- start current period, 160
- start multitasking, 54
- starting a task, 72
- suspending a task, 75
- task, 59, 60
- task affinity, 379
- task arguments, 60
- task attributes, 62
- task migration, 379
- task mode, 60, 62
- task priority, 42, 59
- task private data, 85, 87
- task private variable, 85, 87
- task prototype, 60
- task scheduling, 37
- task stack allocator, 332
- task stack deallocator, 332
- task state transitions, 44
- task states, 59
- tasks, 57
- TCB extension area, 297
- thread affinity, 379
- thread migration, 379
- tick quantum, 328
- ticks per timeslice, 328
- time, 28
- timeouts, 111
- timers, 129
- timeslicing, 42, 60, 111
- uint16_t, 35
- uint32_t, 35
- uint64_t, 35
- uint8_t, 35
- uintptr_t, 35
- unblock all tasks waiting on a semaphore, 182
- unified work areas, 328
- unlock a semaphore, 181
- unregister a device driver, 268
- uptime, 125–128
- user extensions, 295
- wait at a barrier, 194
- wake up after an interval, 82
- wake up at a wall time, 83
- write to a device, 275
- zero C Program Heap, 330
- zero RTEMS Workspace, 330