

Mark3 Realtime Kernel

Generated by Doxygen 1.8.7

Mon Jan 11 2016 21:42:35

Contents

1	The Mark3 Realtime Kernel	1
2	License	3
2.1	License	3
3	Configuring The Mark3 Kernel	5
3.1	Overview	5
3.2	Timer Options	5
3.3	Blocking Objects	6
3.4	Inter-process/thread Communication	7
3.5	Debug Features	7
3.6	Enhancements, Security, Miscellaneous	7
4	Building Mark3	9
4.1	Source Layout	9
4.2	Building the kernel	9
4.3	Building on Windows	11
4.4	Exporting the kernel source	11
5	Getting Started With The Mark3 API	13
5.1	Kernel Setup	13
5.2	Threads	14
5.2.1	Thread Setup	14
5.2.2	Entry Functions	15
5.3	Timers	15
5.4	Semaphores	16
5.5	Mutexes	17
5.6	Event Flags	17
5.7	Messages	18
5.7.1	Message Objects	18
5.7.2	Global Message Pool	19
5.7.3	Message Queues	19
5.7.4	Messaging Example	19

5.8	Mailboxes	20
5.8.1	Mailbox Example	20
5.9	Notification Objects	21
5.9.1	Notification Example	21
5.10	Sleep	21
5.11	Round-Robin Quantum	22
6	Why Mark3?	23
7	When should you use an RTOS?	25
7.1	The reality of system code	25
7.2	Superloops, and their limitations	26
7.2.1	Intro to Superloops	26
7.2.2	The simplest loop	26
7.2.3	Interrupt-Driven Super-loop	27
7.2.4	Cooperative multi-tasking	28
7.2.5	Hybrid cooperative/preemptive multi-tasking	29
7.3	Problems with superloops	30
7.3.1	Hidden Costs	30
7.3.2	Tightly-coupled code	31
7.3.3	No blocking Calls	31
7.3.4	Difficult to guarantee responsiveness	31
7.3.5	Limited preemption capability	31
8	Can you afford an RTOS?	33
8.1	Intro	33
8.2	Application description	34
8.3	Runtime Overhead	35
8.4	Analysis	36
9	Mark3 Design Goals	37
9.1	Overview	37
9.1.1	Services Provided by an RTOS Kernel	37
9.1.2	Guiding Principles of Mark3	37
9.1.3	Be feature competitive	37
9.1.4	Be highly configuration	37
9.1.5	No external dependencies, no new language features	38
9.1.6	Target the most popular hobbyist platforms available	38
9.1.7	Maximize determinism – but be pragmatic	38
9.1.8	Apply engineering principles – and that means discipline, measurement and verification	38
9.1.9	Use Virtualization For Verification	38

10 Mark3 Kernel Architecture	41
10.1 Overview	41
10.2 Threads and Scheduling	43
10.2.1 A Bit About Threads	43
10.2.2 Thread States and ThreadLists	44
10.2.3 Blocking and Unblocking	44
10.2.4 Blocking Objects	45
10.3 Inside the Mark3 Scheduler	46
10.3.1 Considerations for Round-Robin Scheduling	47
10.3.2 Context Switching	47
10.3.3 Putting It All Together	48
10.4 Timers	48
10.4.1 Tick-based Timers	50
10.4.2 Tickless Timers	50
10.4.3 Timer Processing Algorithm	50
10.5 Synchronization and IPC	51
10.6 Blocking Objects	51
10.6.1 Semaphores	52
10.6.2 Mutex	52
10.6.3 Event Flags	52
10.6.4 Notification Objects	52
10.7 Messages and Global Message Queue	53
10.7.1 Messages	53
10.7.2 Message Objects	53
10.7.3 Global Message Pool	53
10.7.4 Message Queues	54
10.7.5 Mailboxes	54
10.7.6 Atomic Operations	54
10.7.7 Drivers	55
10.8 Kernel Proper and Porting	57
11 Build System	65
11.1 Introduction	65
11.2 Mark3 Build Process Overview	66
11.2.1 Pre-Build Phase:	66
11.2.2 Build Phase	67
11.2.3 Test and Profile	67
11.2.4 Release	68
12 Profiling Results	69
12.1 Date Performed	69

12.2 Compiler Information	69
12.3 Profiling Results	69
13 Code Size Profiling	71
13.1 Information	71
13.2 Compiler Version	71
13.3 Profiling Results	71
14 Hierarchical Index	73
14.1 Class Hierarchy	73
15 Class Index	75
15.1 Class List	75
16 File Index	77
16.1 File List	77
17 Class Documentation	81
17.1 BlockingObject Class Reference	81
17.1.1 Detailed Description	81
17.1.2 Member Function Documentation	82
17.1.2.1 Block	82
17.1.2.2 BlockPriority	82
17.1.2.3 Unblock	82
17.2 CircularLinkedList Class Reference	82
17.2.1 Detailed Description	83
17.2.2 Member Function Documentation	83
17.2.2.1 Add	83
17.2.2.2 InsertNodeBefore	83
17.2.2.3 PivotBackward	84
17.2.2.4 PivotForward	84
17.2.2.5 Remove	84
17.3 DevNull Class Reference	84
17.3.1 Detailed Description	85
17.3.2 Member Function Documentation	85
17.3.2.1 Close	85
17.3.2.2 Control	85
17.3.2.3 Init	86
17.3.2.4 Open	86
17.3.2.5 Read	86
17.3.2.6 Write	87
17.4 DoubleLinkedList Class Reference	87

17.4.1 Detailed Description	87
17.4.2 Constructor & Destructor Documentation	88
17.4.2.1 DoubleLinkedList	88
17.4.3 Member Function Documentation	88
17.4.3.1 Add	88
17.4.3.2 Remove	88
17.5 Driver Class Reference	88
17.5.1 Detailed Description	89
17.5.2 Member Function Documentation	89
17.5.2.1 Close	89
17.5.2.2 Control	90
17.5.2.3 GetPath	91
17.5.2.4 Init	91
17.5.2.5 Open	91
17.5.2.6 Read	91
17.5.2.7 SetName	92
17.5.2.8 Write	92
17.6 DriverList Class Reference	92
17.6.1 Detailed Description	93
17.6.2 Member Function Documentation	93
17.6.2.1 Add	93
17.6.2.2 FindByPath	93
17.6.2.3 Init	93
17.6.2.4 Remove	94
17.7 EventFlag Class Reference	95
17.7.1 Detailed Description	96
17.7.2 Member Function Documentation	96
17.7.2.1 Clear	96
17.7.2.2 GetMask	96
17.7.2.3 Set	96
17.7.2.4 Wait	97
17.7.2.5 Wait	98
17.7.2.6 Wait_i	98
17.7.2.7 WakeMe	99
17.8 FakeThread_t Struct Reference	99
17.8.1 Detailed Description	99
17.9 GlobalMessagePool Class Reference	100
17.9.1 Detailed Description	100
17.9.2 Member Function Documentation	100
17.9.2.1 Init	100

17.9.2.2 Pop	100
17.9.2.3 Push	101
17.10 Kernel Class Reference	101
17.10.1 Detailed Description	102
17.10.2 Member Function Documentation	102
17.10.2.1 GetIdleThread	102
17.10.2.2 Init	102
17.10.2.3 IsPanic	102
17.10.2.4 IsStarted	103
17.10.2.5 Panic	103
17.10.2.6 SetIdleFunc	103
17.10.2.7 SetPanic	103
17.10.2.8 Start	103
17.11 KernelAware Class Reference	104
17.11.1 Detailed Description	104
17.11.2 Member Function Documentation	105
17.11.2.1 ExitSimulator	105
17.11.2.2 IsSimulatorAware	105
17.11.2.3 Print	105
17.11.2.4 ProfileInit	105
17.11.2.5 ProfileReport	106
17.11.2.6 ProfileStart	106
17.11.2.7 ProfileStop	106
17.11.2.8 Trace	106
17.11.2.9 Trace	106
17.11.2.10 Trace	106
17.11.2.11 Trace_i	107
17.12 KernelAwareData_t Union Reference	107
17.12.1 Detailed Description	107
17.13 KernelSWI Class Reference	108
17.13.1 Detailed Description	108
17.13.2 Member Function Documentation	108
17.13.2.1 Clear	108
17.13.2.2 Config	108
17.13.2.3 DI	109
17.13.2.4 RI	109
17.13.2.5 Start	109
17.13.2.6 Stop	109
17.13.2.7 Trigger	109
17.14 KernelTimer Class Reference	109

17.14.1 Detailed Description	110
17.14.2 Member Function Documentation	110
17.14.2.1 ClearExpiry	110
17.14.2.2 Config	110
17.14.2.3 DI	111
17.14.2.4 EI	111
17.14.2.5 GetOvertime	111
17.14.2.6 Read	111
17.14.2.7 RI	111
17.14.2.8 SetExpiry	111
17.14.2.9 Start	112
17.14.2.10 Stop	112
17.14.2.11 SubtractExpiry	112
17.14.2.12 TimeToExpiry	112
17.15 LinkList Class Reference	113
17.15.1 Detailed Description	113
17.15.2 Member Function Documentation	113
17.15.2.1 Add	113
17.15.2.2 GetHead	114
17.15.2.3 GetTail	114
17.15.2.4 Init	114
17.15.2.5 Remove	114
17.16 LinkListNode Class Reference	114
17.16.1 Detailed Description	115
17.16.2 Member Function Documentation	115
17.16.2.1 ClearNode	115
17.16.2.2 GetNext	116
17.16.2.3 GetPrev	116
17.17 Mailbox Class Reference	116
17.17.1 Detailed Description	117
17.17.2 Member Function Documentation	118
17.17.2.1 CopyData	118
17.17.2.2 GetHeadPointer	118
17.17.2.3 GetTailPointer	118
17.17.2.4 Init	118
17.17.2.5 MoveHeadBackward	119
17.17.2.6 MoveHeadForward	119
17.17.2.7 MoveTailBackward	119
17.17.2.8 MoveTailForward	119
17.17.2.9 Receive	119

17.17.2.10	Receive	119
17.17.2.11	Receive_i	120
17.17.2.12	ReceiveTail	120
17.17.2.13	ReceiveTail	120
17.17.2.14	Send	121
17.17.2.15	Send	121
17.17.2.16	Send_i	121
17.17.2.17	SendTail	122
17.17.2.18	SendTail	122
17.17.3	Member Data Documentation	122
17.17.3.1	m_clSendSem	122
17.18	Message Class Reference	122
17.18.1	Detailed Description	123
17.18.2	Member Function Documentation	123
17.18.2.1	GetCode	123
17.18.2.2	GetData	124
17.18.2.3	Init	124
17.18.2.4	SetCode	124
17.18.2.5	SetData	124
17.19	MessageQueue Class Reference	125
17.19.1	Detailed Description	125
17.19.2	Member Function Documentation	126
17.19.2.1	GetCount	126
17.19.2.2	Init	126
17.19.2.3	Receive	126
17.19.2.4	Receive	126
17.19.2.5	Receive_i	127
17.19.2.6	Send	127
17.20	Mutex Class Reference	127
17.20.1	Detailed Description	128
17.20.2	Member Function Documentation	128
17.20.2.1	Claim	128
17.20.2.2	Claim	129
17.20.2.3	Claim_i	129
17.20.2.4	Init	129
17.20.2.5	Release	129
17.20.2.6	WakeMe	130
17.20.2.7	WakeNext	130
17.21	Notify Class Reference	130
17.21.1	Detailed Description	131

17.21.2 Member Function Documentation	131
17.21.2.1 Init	131
17.21.2.2 Signal	131
17.21.2.3 Wait	131
17.21.2.4 Wait	132
17.21.2.5 WakeMe	132
17.22 Profiler Class Reference	132
17.22.1 Detailed Description	133
17.22.2 Member Function Documentation	133
17.22.2.1 GetEpoch	133
17.22.2.2 Init	133
17.22.2.3 Process	133
17.22.2.4 Read	133
17.22.2.5 Start	133
17.22.2.6 Stop	134
17.23 ProfileTimer Class Reference	134
17.23.1 Detailed Description	135
17.23.2 Member Function Documentation	135
17.23.2.1 ComputeCurrentTicks	135
17.23.2.2 GetAverage	135
17.23.2.3 GetCurrent	135
17.23.2.4 Init	135
17.23.2.5 Start	136
17.23.2.6 Stop	136
17.24 Quantum Class Reference	136
17.24.1 Detailed Description	136
17.24.2 Member Function Documentation	137
17.24.2.1 AddThread	137
17.24.2.2 ClearInTimer	137
17.24.2.3 RemoveThread	137
17.24.2.4 SetInTimer	137
17.24.2.5 SetTimer	137
17.24.2.6 UpdateTimer	137
17.25 Scheduler Class Reference	138
17.25.1 Detailed Description	138
17.25.2 Member Function Documentation	139
17.25.2.1 Add	139
17.25.2.2 GetCurrentThread	139
17.25.2.3 GetNextThread	139
17.25.2.4 GetStopList	139

17.25.2.5 GetThreadList	139
17.25.2.6 Init	140
17.25.2.7 IsEnabled	140
17.25.2.8 QueueScheduler	140
17.25.2.9 Remove	140
17.25.2.10Schedule	140
17.25.2.11SetScheduler	141
17.26Semaphore Class Reference	141
17.26.1 Detailed Description	142
17.26.2 Member Function Documentation	142
17.26.2.1 GetCount	142
17.26.2.2 Init	142
17.26.2.3 Pend	143
17.26.2.4 Pend	143
17.26.2.5 Pend_i	143
17.26.2.6 Post	143
17.26.2.7 WakeMe	144
17.26.2.8 WakeNext	144
17.27Thread Class Reference	144
17.27.1 Detailed Description	147
17.27.2 Member Function Documentation	147
17.27.2.1 ContextSwitchSWI	147
17.27.2.2 Exit	147
17.27.2.3 GetCurPriority	147
17.27.2.4 GetCurrent	147
17.27.2.5 GetEventFlagMask	148
17.27.2.6 GetEventFlagMode	148
17.27.2.7 GetExpired	148
17.27.2.8 GetID	148
17.27.2.9 GetOwner	148
17.27.2.10GetPriority	149
17.27.2.11GetQuantum	149
17.27.2.12GetStackSlack	149
17.27.2.13GetState	149
17.27.2.14InheritPriority	149
17.27.2.15Init	150
17.27.2.16InitIdle	150
17.27.2.17SetCurrent	150
17.27.2.18SetEventFlagMask	150
17.27.2.19SetEventFlagMode	151

17.27.2.20	SetExpired	151
17.27.2.21	SetID	151
17.27.2.22	SetOwner	151
17.27.2.23	SetPriority	151
17.27.2.24	SetPriorityBase	152
17.27.2.25	SetQuantum	152
17.27.2.26	SetState	152
17.27.2.27	Sleep	152
17.27.2.28	Start	153
17.27.2.29	Stop	153
17.27.2.30	USleep	153
17.27.2.31	Yield	153
17.28	ThreadList Class Reference	153
17.28.1	Detailed Description	154
17.28.2	Constructor & Destructor Documentation	154
17.28.2.1	ThreadList	154
17.28.3	Member Function Documentation	155
17.28.3.1	Add	155
17.28.3.2	Add	155
17.28.3.3	AddPriority	155
17.28.3.4	HighestWaiter	155
17.28.3.5	Remove	155
17.28.3.6	SetFlagPointer	156
17.28.3.7	SetPriority	156
17.29	ThreadPort Class Reference	156
17.29.1	Detailed Description	157
17.29.2	Member Function Documentation	157
17.29.2.1	InitStack	157
17.29.2.2	StartThreads	157
17.30	Timer Class Reference	157
17.30.1	Detailed Description	159
17.30.2	Constructor & Destructor Documentation	159
17.30.2.1	Timer	159
17.30.3	Member Function Documentation	159
17.30.3.1	GetInterval	159
17.30.3.2	Init	159
17.30.3.3	SetCallback	159
17.30.3.4	SetData	159
17.30.3.5	SetFlags	160
17.30.3.6	SetIntervalMSeconds	160

17.30.3.7 SetIntervalSeconds	160
17.30.3.8 SetIntervalTicks	160
17.30.3.9 SetIntervalUSeconds	160
17.30.3.10SetOwner	161
17.30.3.11SetTolerance	161
17.30.3.12Start	161
17.30.3.13Start	161
17.30.3.14Stop	162
17.31TimerList Class Reference	162
17.31.1 Detailed Description	163
17.31.2 Member Function Documentation	163
17.31.2.1 Add	163
17.31.2.2 Init	163
17.31.2.3 Process	163
17.31.2.4 Remove	163
17.32TimerScheduler Class Reference	164
17.32.1 Detailed Description	164
17.32.2 Member Function Documentation	164
17.32.2.1 Add	164
17.32.2.2 Init	164
17.32.2.3 Process	164
17.32.2.4 Remove	165
18 File Documentation	167
18.1 /home/mo/mark3-source/embedded/kernel/atomic.cpp File Reference	167
18.1.1 Detailed Description	167
18.2 atomic.cpp	167
18.3 /home/mo/mark3-source/embedded/kernel/autoalloc.cpp File Reference	169
18.3.1 Detailed Description	169
18.4 autoalloc.cpp	169
18.5 /home/mo/mark3-source/embedded/kernel/blocking.cpp File Reference	170
18.5.1 Detailed Description	170
18.6 blocking.cpp	170
18.7 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp File Reference	171
18.7.1 Detailed Description	172
18.8 kernelprofile.cpp	172
18.9 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp File Reference	173
18.9.1 Detailed Description	173
18.10kernelswi.cpp	173
18.11/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp File Reference	174

18.11.1 Detailed Description	174
18.12kerneltimer.cpp	174
18.13/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h File Reference	176
18.13.1 Detailed Description	177
18.14kernelprofile.h	177
18.15/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h File Reference	177
18.15.1 Detailed Description	178
18.16kernelswi.h	178
18.17/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h File Reference	178
18.17.1 Detailed Description	179
18.18kerneltimer.h	179
18.19/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h File Reference	179
18.19.1 Detailed Description	180
18.19.2 Macro Definition Documentation	180
18.19.2.1 CS_ENTER	180
18.20threadport.h	181
18.21/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp File Reference	182
18.21.1 Detailed Description	183
18.22threadport.cpp	183
18.23/home/mo/mark3-source/embedded/kernel/driver.cpp File Reference	185
18.23.1 Detailed Description	186
18.23.2 Function Documentation	186
18.23.2.1 DrvCmp	186
18.24driver.cpp	186
18.25/home/mo/mark3-source/embedded/kernel/eventflag.cpp File Reference	187
18.25.1 Detailed Description	188
18.25.2 Function Documentation	188
18.25.2.1 TimedEventFlag_Callback	188
18.26eventflag.cpp	188
18.27/home/mo/mark3-source/embedded/kernel/kernel.cpp File Reference	192
18.27.1 Detailed Description	192
18.28kernel.cpp	193
18.29/home/mo/mark3-source/embedded/kernel/kernelaware.cpp File Reference	194
18.29.1 Detailed Description	194
18.29.2 Variable Documentation	194
18.29.2.1 g_bIsKernelAware	195
18.29.2.2 g_stKAData	195
18.30kernelaware.cpp	195

18.31/home/mo/mark3-source/embedded/kernel/ksemaphore.cpp File Reference	197
18.31.1 Detailed Description	197
18.31.2 Function Documentation	197
18.31.2.1 TimedSemaphore_Callback	197
18.32ksemaphore.cpp	197
18.33/home/mo/mark3-source/embedded/kernel/ll.cpp File Reference	200
18.33.1 Detailed Description	200
18.34ll.cpp	200
18.35/home/mo/mark3-source/embedded/kernel/mailbox.cpp File Reference	203
18.35.1 Detailed Description	203
18.36mailbox.cpp	203
18.37/home/mo/mark3-source/embedded/kernel/message.cpp File Reference	207
18.37.1 Detailed Description	207
18.38message.cpp	207
18.39/home/mo/mark3-source/embedded/kernel/mutex.cpp File Reference	209
18.39.1 Detailed Description	209
18.39.2 Function Documentation	209
18.39.2.1 TimedMutex_Callback	209
18.40mutex.cpp	210
18.41/home/mo/mark3-source/embedded/kernel/notify.cpp File Reference	213
18.41.1 Detailed Description	213
18.42notify.cpp	213
18.43/home/mo/mark3-source/embedded/kernel/profile.cpp File Reference	215
18.43.1 Detailed Description	215
18.44profile.cpp	215
18.45/home/mo/mark3-source/embedded/kernel/public/atomic.h File Reference	217
18.45.1 Detailed Description	217
18.46atomic.h	217
18.47/home/mo/mark3-source/embedded/kernel/public/autoalloc.h File Reference	218
18.47.1 Detailed Description	218
18.48autoalloc.h	218
18.49/home/mo/mark3-source/embedded/kernel/public/blocking.h File Reference	219
18.49.1 Detailed Description	219
18.50blocking.h	219
18.51/home/mo/mark3-source/embedded/kernel/public/buffalogger.h File Reference	220
18.51.1 Detailed Description	220
18.52buffalogger.h	220
18.53/home/mo/mark3-source/embedded/kernel/public/driver.h File Reference	221
18.53.1 Detailed Description	221
18.53.2 Intro	221

18.53.3 Driver Design	221
18.53.4 Driver API	221
18.54driver.h	222
18.55/home/mo/mark3-source/embedded/kernel/public/eventflag.h File Reference	223
18.55.1 Detailed Description	223
18.56eventflag.h	223
18.57/home/mo/mark3-source/embedded/kernel/public/kernel.h File Reference	224
18.57.1 Detailed Description	224
18.58kernel.h	225
18.59/home/mo/mark3-source/embedded/kernel/public/kernelaware.h File Reference	225
18.59.1 Detailed Description	226
18.59.2 Enumeration Type Documentation	226
18.59.2.1 KernelAwareCommand_t	226
18.60kernelaware.h	227
18.61/home/mo/mark3-source/embedded/kernel/public/kerneldebug.h File Reference	228
18.61.1 Detailed Description	228
18.62kerneldebug.h	228
18.63/home/mo/mark3-source/embedded/kernel/public/kerneltypes.h File Reference	231
18.63.1 Detailed Description	232
18.63.2 Enumeration Type Documentation	232
18.63.2.1 EventFlagOperation_t	232
18.64kerneltypes.h	232
18.65/home/mo/mark3-source/embedded/kernel/public/ksemaphore.h File Reference	233
18.65.1 Detailed Description	233
18.66ksemaphore.h	233
18.67/home/mo/mark3-source/embedded/kernel/public/ll.h File Reference	234
18.67.1 Detailed Description	234
18.68ll.h	234
18.69/home/mo/mark3-source/embedded/kernel/public/mailbox.h File Reference	236
18.69.1 Detailed Description	236
18.70mailbox.h	236
18.71/home/mo/mark3-source/embedded/kernel/public/manual.h File Reference	238
18.71.1 Detailed Description	238
18.72manual.h	238
18.73/home/mo/mark3-source/embedded/kernel/public/mark3.h File Reference	239
18.73.1 Detailed Description	239
18.74mark3.h	239
18.75/home/mo/mark3-source/embedded/kernel/public/mark3cfg.h File Reference	240
18.75.1 Detailed Description	241
18.75.2 Macro Definition Documentation	241

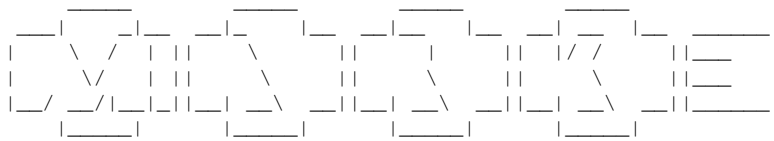
18.75.2.1 GLOBAL_MESSAGE_POOL_SIZE	241
18.75.2.2 KERNEL_AWARE_SIMULATION	242
18.75.2.3 KERNEL_ENABLE_LOGGING	242
18.75.2.4 KERNEL_ENABLE_USER_LOGGING	242
18.75.2.5 KERNEL_TIMERS_TICKLESS	242
18.75.2.6 KERNEL_USE_ATOMIC	242
18.75.2.7 KERNEL_USE_AUTO_ALLOC	242
18.75.2.8 KERNEL_USE_DYNAMIC_THREADS	243
18.75.2.9 KERNEL_USE_EVENTFLAG	243
18.75.2.10 KERNEL_USE_IDLE_FUNC	243
18.75.2.11 KERNEL_USE_MAILBOX	243
18.75.2.12 KERNEL_USE_MESSAGE	243
18.75.2.13 KERNEL_USE_PROFILER	243
18.75.2.14 KERNEL_USE_QUANTUM	243
18.75.2.15 KERNEL_USE_SEMAPHORE	244
18.75.2.16 KERNEL_USE_THREADNAME	244
18.75.2.17 KERNEL_USE_TIMEOUTS	244
18.75.2.18 KERNEL_USE_TIMERS	244
18.75.2.19 SAFE_UNLINK	244
18.75.2.20 THREAD_QUANTUM_DEFAULT	244
18.76 mark3cfg.h	245
18.77 /home/mo/mark3-source/embedded/kernel/public/message.h File Reference	246
18.77.1 Detailed Description	246
18.77.2 using Messages, Queues, and the Global Message Pool	246
18.78 message.h	247
18.79 /home/mo/mark3-source/embedded/kernel/public/mutex.h File Reference	248
18.79.1 Detailed Description	249
18.79.2 Initializing	249
18.79.3 Resource protection example	249
18.80 mutex.h	249
18.81 /home/mo/mark3-source/embedded/kernel/public/notify.h File Reference	250
18.81.1 Detailed Description	250
18.82 notify.h	250
18.83 /home/mo/mark3-source/embedded/kernel/public/paniccodes.h File Reference	251
18.83.1 Detailed Description	251
18.84 paniccodes.h	251
18.85 /home/mo/mark3-source/embedded/kernel/public/profile.h File Reference	251
18.85.1 Detailed Description	252
18.86 profile.h	252
18.87 /home/mo/mark3-source/embedded/kernel/public/quantum.h File Reference	253

18.87.1 Detailed Description	253
18.88quantum.h	253
18.89/home/mo/mark3-source/embedded/kernel/public/scheduler.h File Reference	254
18.89.1 Detailed Description	254
18.90scheduler.h	255
18.91/home/mo/mark3-source/embedded/kernel/public/thread.h File Reference	256
18.91.1 Detailed Description	256
18.92thread.h	256
18.93/home/mo/mark3-source/embedded/kernel/public/threadlist.h File Reference	259
18.93.1 Detailed Description	259
18.94threadlist.h	259
18.95/home/mo/mark3-source/embedded/kernel/public/timer.h File Reference	260
18.95.1 Detailed Description	261
18.95.2 Macro Definition Documentation	261
18.95.2.1 TIMERLIST_FLAG_EXPIRED	261
18.95.3 Typedef Documentation	261
18.95.3.1 TimerCallback_t	261
18.96timer.h	261
18.97/home/mo/mark3-source/embedded/kernel/public/timerlist.h File Reference	263
18.97.1 Detailed Description	263
18.98timerlist.h	263
18.99/home/mo/mark3-source/embedded/kernel/public/timerscheduler.h File Reference	264
18.99.1 Detailed Description	264
18.100timerscheduler.h	264
18.101/home/mo/mark3-source/embedded/kernel/public/tracebuffer.h File Reference	265
18.101.1 Detailed Description	265
18.102tracebuffer.h	265
18.103/home/mo/mark3-source/embedded/kernel/quantum.cpp File Reference	266
18.103.1 Detailed Description	266
18.103.2 Function Documentation	266
18.103.2.1 QuantumCallback	266
18.104quantum.cpp	267
18.105/home/mo/mark3-source/embedded/kernel/scheduler.cpp File Reference	268
18.105.1 Detailed Description	269
18.105.2 Variable Documentation	269
18.105.2.1 aucCLZ	269
18.106scheduler.cpp	269
18.107/home/mo/mark3-source/embedded/kernel/thread.cpp File Reference	271
18.107.1 Detailed Description	271
18.108thread.cpp	271

18.109	home/mo/mark3-source/embedded/kernel/threadlist.cpp File Reference	276
18.109	Detailed Description	277
18.110	hreadlist.cpp	277
18.111	home/mo/mark3-source/embedded/kernel/timer.cpp File Reference	278
18.111	Detailed Description	278
18.112	timer.cpp	279
18.113	home/mo/mark3-source/embedded/kernel/timerlist.cpp File Reference	280
18.113	Detailed Description	280
18.114	timerlist.cpp	280
18.115	home/mo/mark3-source/embedded/kernel/tracebuffer.cpp File Reference	283
18.115	Detailed Description	284
18.116	tracebuffer.cpp	284
19	Example Documentation	287
19.1	buffalogger/main.cpp	287
19.2	lab10_notifications/main.cpp	289
19.3	lab11_mailboxes/main.cpp	290
19.4	lab1_kernel_setup/main.cpp	292
19.5	lab2_idle_function/main.cpp	293
19.6	lab3_round_robin/main.cpp	295
19.7	lab4_semaphores/main.cpp	296
19.8	lab5_mutexes/main.cpp	298
19.9	lab6_timers/main.cpp	300
19.10	lab7_events/main.cpp	301
19.11	lab8_messages/main.cpp	303
19.12	lab9_dynamic_threads/main.cpp	305
Index		308

Chapter 1

The Mark3 Realtime Kernel



--[Mark3 Realtime Platform]-----

Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
See license for more information

The Mark3 Realtime [Kernel](#) is a completely free, open-source, real-time operating system aimed at bringing powerful, easy-to-use multitasking to microcontroller systems without MMUs.

It uses modern programming languages and concepts to minimize code duplication, and its object-oriented design enhances readability. The API is simple – in six function calls, you can set up the kernel, initialize two threads, and start the scheduler.

The source is fully-documented with example code provided to illustrate concepts. The result is a performant RTOS, which is easy to read, easy to understand, and easy to extend to fit your needs.

But Mark3 is bigger than just a real-time kernel, it also contains a number of class-leading features:

- Device driver HAL which provides a meaningful abstraction around device-specific peripherals.
- Capable recursive-make driven build system which can be used to build all libraries, examples, tests, documentation, and user-projects for any number of targets from the command-line.
- Graphics and UI code designed to simplify the implementation of systems using displays, keypads, joysticks, and touchscreens
- Standards-based custom communications protocol used to simplify the creation of host tools
- A bulletproof, well-documented bootloader for AVR microcontrollers Support for kernel-aware simulators, including Funkenstein's own fIAVR.

Chapter 2

License

2.1 License

Copyright (c) 2012-2015, Funkenstein Software Consulting All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Funkenstein Software Consulting, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL FUNKENSTEIN SOFTWARE (MARK SLEVINSKY) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF use, DATA, OR PROFITS; OR BUSINESS INTERRUPTI↵ON) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE use OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 3

Configuring The Mark3 Kernel

3.1 Overview

The Mark3 [Kernel](#) features a large number of compile-time options that can be set by the user. In this way, the user can build a custom OS kernel that provides only the necessary feature set required by the application, and reduce the code and data requirements of the kernel.

Care has been taken to ensure that all valid combinations of features can be enabled or disabled, barring direct dependencies.

When Mark3 is built, the various compile-time definitions are used to alter how the kernel is compiled, and include or exclude various bits and pieces in order to satisfy the requirements of the selected features. As a result, the kernel must be rebuilt whenever changes are made to the configuration header.

Note that not all demos, libraries, and tests will build successfully if the prerequisite features are not included.

[Kernel](#) options are set by modifying [mark3cfg.h](#), located within the /kernel/public folder.

In the following sections, we will discuss the various configuration options, grouped by functionality.

3.2 Timer Options

KERNEL_USE_TIMERS

This option is related to all kernel time-tracking:

- Timers provide a way for events to be periodically triggered in a lightweight manner. These can be periodic, or one-shot.
- [Thread Quantum](#) (used for round-robin scheduling) is dependent on this module, as is [Thread Sleep](#) functionality.

Setting this option to 0 disables all timer-based functionality within the kernel.

KERNEL_TIMERS_TICKLESS

If you've opted to use the kernel timers module, you have an option as to which timer implementation to use: Tick-based or Tick-less.

Tick-based timers provide a "traditional" RTOS timer implementation based on a fixed-frequency timer interrupt. While this provides very accurate, reliable timing, it also means that the CPU is being interrupted far more often than may be necessary (as not all timer ticks result in "real work" being done).

Tick-less timers still rely on a hardware timer interrupt, but uses a dynamic expiry interval to ensure that the interrupt is only called when the next timer expires. This increases the complexity of the timer interrupt handler, but reduces the number and frequency.

Note that the CPU port ([kerneltimer.cpp](#)) must be implemented for the particular timer variant desired.

Set this option to 1 to use the tickless timer implementation, 0 to use the traditional tick-based approach. Tickless timers are a bit more heavy weight (larger code footprint), but can yield significant power savings as the CPU does not need to wake up at a fixed, high frequency.

KERNEL_USE_TIMEOUTS

By default, if you opt to enable kernel timers, you also get timeout- enabled versions of the blocking object APIs along with it. This support comes at a small cost to code size, but a slightly larger cost to realtime performance - as checking for the use of timers in the underlying internal code costs some cycles.

As a result, the option is given to the user here to manually disable these timeout-based APIs if desired by the user for performance and code-size reasons.

Set this option to 1 to enable timeout-based APIs for blocking calls.

KERNEL_USE_QUANTUM

Do you want to enable time quanta? This is useful when you want to have tasks in the same priority group share time in a controlled way. This allows equal tasks to use unequal amounts of the CPU, which is a great way to set up CPU budgets per thread in a round-robin scheduling system. If enabled, you can specify a number of ticks that serves as the default time period (quantum). Unless otherwise specified, every thread in a priority will get the default quantum.

Set this option to 1 to enable round-robin scheduling.

THREAD_QUANTUM_DEFAULT

This value defines the default thread quantum when `KERNEL_USE_QUANTUM` is enabled. The value defined is a time in milliseconds.

KERNEL_USE_SLEEP

This define enables the [Thread::Sleep\(\)](#) API, which allows a thread to suspend its operation for a defined length of time, specified in ms.

3.3 Blocking Objects

KERNEL_USE_NOTIFY

This is a simple blocking object, where a thread (or threads) are guaranteed to block until an asynchronous event signals the object.

KERNEL_USE_SEMAPHORE

Do you want the ability to use counting/binary semaphores for thread synchronization? Enabling this features provides fully-blocking semaphores and enables all API functions declared in [semaphore.h](#). If you have to pick one blocking mechanism, this is the one to choose.

Note that all IPC mechanisms (mailboxes, messages) rely on semaphores, so keep in mind that this is a prerequisite for many other features in the kernel.

KERNEL_USE_MUTEX

Do you want the ability to use mutual exclusion semaphores (mutex) for resource/block protection? Enabling this feature provides mutexes, with priority inheritance, as declared in [mutex.h](#).

KERNEL_USE_EVENTFLAG

Provides additional event-flag based blocking. This relies on an additional per-thread flag-mask to be allocated, which adds 2 bytes to the size of each thread object.

3.4 Inter-process/thread Communication

KERNEL_USE_MESSAGE

Enable inter-thread messaging using message queues. This is the preferred mechanism for IPC for serious multi-threaded communications; generally anywhere a semaphore or event-flag is insufficient.

GLOBAL_MESSAGE_POOL_SIZE

If Messages are enabled, define the size of the default kernel message pool. Messages can be manually added to the message pool, but this mechanism is more convenient and automatic. All message queues can share their message objects from this global pool to maximize efficiency and simplify data management.

KERNEL_USE_MAILBOX

Enable inter-thread messaging using mailboxes. A mailbox manages a blob of data provided by the user, that is partitioned into fixed-size blocks called envelopes. The size of an envelope is set by the user when the mailbox is initialized. Any number of threads can read-from and write-to the mailbox. Envelopes can be sent-to or received-from the mailbox at the head or tail. In this way, mailboxes essentially act as a circular buffer that can be used as a blocking FIFO or LIFO queue.

3.5 Debug Features

KERNEL_USE_THREADNAME

Provide [Thread](#) method to allow the user to set a name for each thread in the system. Adds a const char* pointer to the size of the thread object.

KERNEL_USE_DEBUG

Provides extra logic for kernel debugging, and instruments the kernel with extra asserts, and kernel trace functionality.

KERNEL_ENABLE_LOGGING

Set this to 1 to enable very chatty kernel logging. Since most important things in the kernel emit logs, a large log-buffer and fast output are required in order to keep up. This is a pretty advanced power-user type feature, so it's disabled by default.

KERNEL_ENABLE_USER_LOGGING

This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently. This allows for user-code to benefit from the built-in kernel logging macros without having to account for the super-high-volume of logs generated by kernel code.

3.6 Enhancements, Security, Miscellaneous

KERNEL_USE_DRIVER

Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.

KERNEL_USE_DYNAMIC_THREADS

Provide extra [Thread](#) methods to allow the application to create (and more importantly destroy) threads at runtime. useful for designs implementing worker threads, or threads that can be restarted after encountering error conditions.

KERNEL_USE_PROFILER

Provides extra classes for profiling the performance of code. useful for debugging and development, but uses an additional hardware timer.

KERNEL_USE_ATOMIC

Provides support for atomic operations, including addition, subtraction, set, and test-and-set. Add/Sub/Set contain 8, 16, and 32-bit variants.

SAFE_UNLINK

"Safe unlinking" performs extra checks on data to make sure that there are no consistencies when performing operations on linked lists. This goes beyond pointer checks, adding a layer of structural and metadata validation to help detect system corruption early.

KERNEL_AWARE_SIMULATION

Include support for kernel-aware simulation. Enabling this feature adds advanced profiling, trace, and environment-aware debugging and diagnostic functionality when Mark3-based applications are run on the fI AVR simulator.

KERNEL_USE_IDLE_FUNC

Enabling this feature removes the necessity for the user to dedicate a complete thread for idle functionality. This saves a full thread stack, but also requires a bit extra static data. This also adds a slight overhead to the context switch and scheduler, as a special case has to be taken into account.

KERNEL_USE_AUTO_ALLOC

This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory). Note that auto-alloc memory cannot be reclaimed.

AUTO_ALLOC_SIZE

Size (in bytes) of the static pool of memory reserved from RAM for use by the auto allocator (if enabled).

Chapter 4

Building Mark3

Mark3 is distributed with a recursive makefile build system, allowing the entire source tree to be built into a series of libraries with simple make commands.

The way the scripts work, every directory with a valid makefile is scanned, as well as all of its subdirectories. The build then generates binary components for all of the components it finds -libraries and executables. All libraries that are generated can then be imported into an application using the linker without having to copy-and-paste files on a module-by-module basis. Applications built during this process can then be loaded onto a device directly, without requiring a GUI-based IDE. As a result, Mark3 integrates well with 3rd party tools for continuous-integration and automated testing.

This modular framework allows for large volumes of libraries and binaries to be built at once - the default build script leverages this to build all of the examples and unit tests at once, linking against the pre-built kernel, services, and drivers. Whatever can be built as a library is built as a library, promoting reuse throughout the platform, and enabling Mark3 to be used as a platform, with an ecosystem of libraries, services, drivers and applications.

4.1 Source Layout

One key aspect of Mark3 is that system features are organized into their own separate modules. These modules are further grouped together into folders based on the type of features represented:

Root	Base folder, contains recursive makefiles for build system
arduino	Arduino-specific headers and API documentation files
bootloader	Mark3 Bootloader code for AVR microcontrollers
build	Makefiles and device-configuraton data for various platforms
docs	Documentation (including this)
drivers	Device driver code for various supported devices
example	Example applications
export	Platform specific output folder, used when running export.sh
fonts	Bitmap fonts converted from TTF, used by Mark3 graphics library
kernel	Basic Mark3 Components (the focus of this manual)
cpu	CPU-specific porting code
scripts	Scripts used to simplify build, documentation, and profiling
libs	Utility code and services, extended system features
stage	Staging directory, where the build system places artifacts
tests	Unit tests, written as C/C++ applications
util	.net-based font converter, terminal, programmer, config util

4.2 Building the kernel

The base.mak file determines how the kernel, drivers, and libraries are built, for what targets, and with what options. Most of these options can be copied directly from the options found in your IDE managed projects. Below is an overview of the main variables used to configure the build.

```

STAGE      - Location in the filesystem where the build output is stored
ROOT_DIR   - The location of the root source tree
ARCH       - The CPU architecture to build against
VARIANT    - The variant of the above CPU to target
TOOLCHAIN  - Which toolchain to build with (dependent on ARCH and VARIANT)

```

Build.mak contains the logic which is used to perform the recursive make in all directories. Unless you really know what you're doing, it's best to leave this as-is.

You must make sure that all required paths are set in your system environment variables so that they are accessible through from the command-line.

Once configured, you can build the source tree using the various make targets:

- make headers
 - copy all headers in each module's /public subdirectory to the location specified by STAGE environment variable's /inc subdirectory.
- make library
 - regenerate all objects copy marked as libraries (i.e. the kernel + drivers). Resulting binaries are copied into STAGE's /lib subdirectory.
- make binary
 - build all executable projects in the root directory structure. In the default distribution, this includes the basic set of demos.

These steps are chained together automatically as part of the build.sh script found under the /scripts subdirectory. Running ./scripts/build.sh from the root of the embedded source directory will result in all headers being exported, libraries built, and applications built. This script will also default to building for atmega328p using GCC if none of the required environment variables have previously been configured.

To add new components to the recursive build system, simply add your code into a new folder beneath the root install location.

Source files, the module makefile and private header files go directly in the new folder, while public headers are placed in a ./public subdirectory. Create a ./obj directory to hold the output from the builds.

The contents of the module makefile looks something like this:

```

# Include common prelude make file
include $(ROOT_DIR)base.mak

# If we're building a library, set IS_LIB and LIBNAME
# If we're building an app, set IS_APP and APPNAME
IS_LIB=1
LIBNAME=mylib

#this is the list of the source modules required to build the kernel
CPP_SOURCE = mylib.cpp \
             someotherfile.cpp

# Similarly, C-language source would be under the C_SOURCE variable.

# Include the rest of the script that is actually used for building the
# outputs
include $(ROOT_DIR)build.mak

```

Once you've placed your code files in the right place, and configured the makefile appropriately, call the following sequence to guarantee that your code will be built.

```

> make headers
> make library
> make binary

```

4.3 Building on Windows

Building Mark3 on Windows is the same as on Linux, but there are a few prerequisites that need to be taken into consideration before the build scripts and makefiles will work as expected.

Step 1 - Install Latest Atmel Studio IDE

Atmel Studio contains the AVR8 GCC toolchain, which contains the necessary compilers, assemblers, and platform support required to turn the source modules into libraries and executables.

To get Atmel Studio, go to the Atmel website (<http://www.atmel.com>) and register to download the latest version. This is a free download (and rather large). The included IDE (if you choose to use it) is very slick, as it's based on Visual Studio, and contains a wonderful cycle-accurate simulator for AVR devices. In fact, the simulator is so good that most of the kernel and its drivers were developed using this tool.

Once you have downloaded and installed Atmel Studio, you will need to add the location of the AVR toolchain to the PATH environment variable.

To do this, go to Control Panel -> System and Security -> System -> Advanced System Settings, and edit the PATH variable. Append the location of the toolchain bin folder to the end of the variable.

On Windows x64, it should look something like this:

```
C:\Program Files (x86)\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.2.1002\avr8-gnu-toolchain\bin
```

Step 2 - Install MinGW and MinSys

MinGW (and MinSys in particular) provide a unix-like environment that runs under windows. Some of the utilities provided include a version of the bash shell, and GNU standard make - both which are required by the Mark3 recursive build system.

The MinGW installer can be downloaded from its project page on SourceForge. When installing, be sure to select the "MinSys" component.

Once installed, add the MinSys binary path to the PATH environment variable, in a similar fashion as with Atmel Studio in Step 1.

Step 3 - Setup Include Paths in Platform Makefile

The AVR header file path must be added to the "platform.mak" makefile for each AVR Target you are attempting to build for. These files can be located under /embedded/build/avr/atmegaXXX/. The path to the includes directory should be added to the end of the CFLAGS and CPPFLAGS variables, as shown in the following:

```
TEST_INC="/c/Program Files (x86)/Atmel/Atmel Toolchain/AVR8
GCC/Native/3.4.2.1002/avr8-gnu-toolchain/include"
CFLAGS += -I$(TEST_INC)
CPPFLAGS += -I$(TEST_INC)
```

Step 4 - Build Mark3 using Bash

Launch a terminal to your Mark3 base directory, and cd into the "embedded" folder. You should now be able to build Mark3 by running "bash ./build.sh" from the command-line.

Alternately, you can run bash itself, building Mark3 by running ./build.sh or the various make targets using the same syntax as documented previously.

Note - building on Windows is *slow*. This has a lot to do with how "make" performs under windows. There are faster substitutes for make (such as cs-make) that are exponentially quicker, and approach the performance of make on Linux. Other mechanisms, such as running make with multiple concurrent jobs (i.e. "make -j4") also helps significantly, especially on systems with multicore CPUs.

4.4 Exporting the kernel source

While the build system is flexible enough to adapt to any toolchain, it may be desirable to integrate the Mark3 kernel and associated drivers/libraries into another build system.

Mark3 provides a script (the aptly-named `export.sh`) which allow for the source for any supported port to be exported for this purpose. This script will also generate appropriate doxygen documentation, and package the whole of it together in a zip file. The files in the archive are placed in a "flat" heirarchy, and do not require any specific path structure to be maintained when imported into another build system.

As a special feature, if the "arduino" AVR target is specified, additional pre-processing is done on the source to turn the standard Mark3 kernel into a library that can be imported directly into Arudino IDE. This is also how the official Mark3 arduino-compatible releases are generated (hosted on mark3os.com and sourceforge.net)

To exercise the build system, type the following from the main mark3 embedded source directory:

```
> ./scripts/export.sh <architecture> <variant> <toolchain>
```

Where:

```
<architecture> is the CPU archiectecure (i.e. AVR, MSP430, CM0)
<variant> is the CPU model within the architecture (i.e. atmega328p)
<toolchain> is the toolchain compatile with the port (i.e. GCC)
```

If successful, the generated artifacats will be placed in an output folder under the `./export` directory.

Chapter 5

Getting Started With The Mark3 API

5.1 Kernel Setup

This section details the process of defining threads, initializing the kernel, and adding threads to the scheduler.

If you're at all familiar with real-time operating systems, then these setup and initialization steps should be familiar. I've tried very hard to ensure that as much of the heavy lifting is hidden from the user, so that only the bare minimum of calls are required to get things started.

The examples presented in this chapter are real, working examples taken from the ATmega328p port.

First, you'll need to create the necessary data structures and functions for the threads:

1. Create a [Thread](#) object for all of the "root" or "initial" tasks.
2. Allocate stacks for each of the Threads
3. Define an entry-point function for each [Thread](#)

This is shown in the example code below:

```
//-----
#include "thread.h"
#include "kernel.h"

//1) Create a thread object for all of the "root" or "initial" tasks
static Thread AppThread;
static Thread IdleThread;

//2) Allocate stacks for each thread
#define STACK_SIZE_APP      (192)
#define STACK_SIZE_IDLE     (128)

static uint8_t aucAppStack[STACK_SIZE_APP];
static uint8_t aucIdleStack[STACK_SIZE_IDLE];

//3) Define entry point functions for each thread
void AppThread(void);
void IdleThread(void);
```

Next, we'll need to add the required kernel initialization code to main. This consists of running the [Kernel's](#) init routine, initializing all of the threads we defined, adding the threads to the scheduler, and finally calling [Kernel::Start\(\)](#), which transfers control of the system to the RTOS.

These steps are illustrated in the following example.

```
int main(void)
{
    //1) Initialize the kernel prior to use
    Kernel::Init();           // MUST be before other kernel ops

    //2) Initialize all of the threads we've defined
```

```

AppThread.Init( aucAppStack,      // Pointer to the stack
                STACK_SIZE_APP,   // Size of the stack
                1,                // Thread priority
                (void*)AppEntry,   // Entry function
                NULL );           // Entry function argument

IdleThread.Init( aucIdleStack,    // Pointer to the stack
                 STACK_SIZE_IDLE, // Size of the stack
                 0,               // Thread priority
                 (void*)IdleEntry, // Entry function
                 NULL );          // Entry function argument

//3) Add the threads to the scheduler
AppThread.Start();               // Actively schedule the threads
IdleThread.Start();

//4) Give control of the system to the kernel
Kernel::Start();                // Start the kernel!
}

```

Not much to it, is there? There are a few noteworthy points in this code, though.

In order for the kernel to work properly, a system must always contain an idle thread; that is, a thread at priority level 0 that never blocks. This thread is responsible for performing any of the low-level power management on the CPU in order to maximize battery life in an embedded device. The idle thread must also never block, and it must never exit. Either of these operations will cause undefined behavior in the system.

The App thread is at a priority level greater-than 0. This ensures that as long as the App thread has something useful to do, it will be given control of the CPU. In this case, if the app thread blocks, control will be given back to the Idle thread, which will put the CPU into a power-saving mode until an interrupt occurs.

Stack sizes must be large enough to accommodate not only the requirements of the threads, but also the requirements of interrupts - up to the maximum interrupt-nesting level used. Stack overflows are super-easy to run into in an embedded system; if you encounter strange and unexplained behavior in your code, chances are good that one of your threads is blowing its stack.

5.2 Threads

Mark3 Threads act as independent tasks in the system. While they share the same address-space, global data, device-drivers, and system peripherals, each thread has its own set of CPU registers and stack, collectively known as the thread's **context**. The context is what allows the RTOS kernel to rapidly switch between threads at a high rate, giving the illusion that multiple things are happening in a system, when really, only one thread is executing at a time.

5.2.1 Thread Setup

Each instance of the [Thread](#) class represents a thread, its stack, its CPU context, and all of the state and metadata maintained by the kernel. Before a [Thread](#) will be scheduled to run, it must first be initialized with the necessary configuration data.

The Init function gives the user the opportunity to set the stack, stack size, thread priority, entry-point function, entry-function argument, and round-robin time quantum:

[Thread](#) stacks are pointers to blobs of memory (usually char arrays) carved out of the system's address space. Each thread must have a stack defined that's large enough to handle not only the requirements of local variables in the thread's code path, but also the maximum depth of the ISR stack.

Priorities should be chosen carefully such that the shortest tasks with the most strict determinism requirements are executed first - and are thus located in the highest priorities. Tasks that take the longest to execute (and require the least degree of responsiveness) must occupy the lower thread priorities. The idle thread must be the only thread occupying the lowest priority level.

The thread quantum only applies when there are multiple threads in the ready queue at the same priority level. This interval is used to kick-off a timer that will cycle execution between the threads in the priority list so that they each get a fair chance to execute.

The entry function is the function that the kernel calls first when the thread instance is first started. Entry functions have at most one argument - a pointer to a data-object specified by the user during initialization.

An example thread initialization is shown below:

```
Thread clMyThread;
uint8_t aucStack[192];

void AppEntry(void)
{
    while(1)
    {
        // Do something!
    }
}

...
{
    clMyThread.Init(aucStack,    // Pointer to the stack to use by this thread
                    192,        // Size of the stack in bytes
                    1,          // Thread priority (0 = idle, 7 = max)
                    (void*)AppEntry, // Function where the thread starts executing
                    NULL );      // Argument passed into the entry function
}
```

Once a thread has been initialized, it can be added to the scheduler by calling:

```
clMyThread.Start();
```

The thread will be placed into the [Scheduler](#)'s queue at the designated priority, where it will wait its turn for execution.

5.2.2 Entry Functions

Mark3 Threads should not run-to-completion - they should execute as infinite loops that perform a series of tasks, appropriately partitioned to provide the responsiveness characteristics desired in the system.

The most basic [Thread](#) loop is shown below:

```
void Thread( void *param )
{
    while(1)
    {
        // Do Something
    }
}
```

Threads can interact with eachother in the system by means of synchronization objects ([Semaphore](#)), mutual-exclusion objects ([Mutex](#)), Inter-process messaging ([MessageQueue](#)), and timers ([Timer](#)).

Threads can suspend their own execution for a predetermined period of time by using the static [Thread::Sleep\(\)](#) method. Calling this will block the [Thread](#)'s executin until the amount of time specified has ellapsed. Upon expiry, the thread will be placed back into the ready queue for its priority level, where it awaits its next turn to run.

5.3 Timers

[Timer](#) objects are used to trigger callback events periodic or on a one-shot (alarm) basis.

While extremely simple to use, they provide one of the most powerful execution contexts in the system. The timer callbacks execute from within the timer callback ISR in an interrupt-enabled context. As such, timer callbacks are considered higher-priority than any thread in the system, but lower priority than other interrupts. Care must be taken to ensure that timer callbacks execute as quickly as possible to minimize the impact of processing on the throughput of tasks in the system. Wherever possible, heavy-lifting should be deferred to the threads by way of semaphores or messages.

Below is an example showing how to start a periodic system timer which will trigger every second:

```

{
    Timer clTimer;
    clTimer.Init();

    clTimer.Start( 1000,
                  1,
                  MyCallback,
                  (void*)&my_data );

    ... // Keep doing work in the thread
}

// Callback function, executed from the timer-expiry context.
void MyCallback( Thread *pclOwner_, void *pvData_ )
{
    LED.Flash(); // Flash an LED.
}

```

5.4 Semaphores

Semaphores are used to synchronized execution of threads based on the availability (and quantity) of application-specific resources in the system. They are extremely useful for solving producer-consumer problems, and are the method-of-choice for creating efficient, low latency systems, where ISRs post semaphores that are handled from within the context of individual threads. (Yes, Semaphores can be posted - but not pended - from the interrupt context).

The following is an example of the producer-consumer usage of a binary semaphore:

```

Semaphore clSemaphore; // Declare a semaphore shared between a producer and a consumer thread.

void Producer()
{
    clSemaphore.Init(0, 1);
    while(1)
    {
        // Do some work, create something to be consumed

        // Post a semaphore, allowing another thread to consume the data
        clSemaphore.Post();
    }
}

void Consumer()
{
    // Assumes semaphore initialized before use...
    While(1)
    {
        // Wait for new data from the producer thread
        clSemaphore.Pend();

        // Consume the data!
    }
}

```

And an example of using semaphores from the ISR context to perform event- driven processing.

```

Semaphore clSemaphore;

__interrupt__ MyISR()
{
    clSemaphore.Post(); // Post the interrupt. Lightweight when uncontested.
}

void MyThread()
{
    clSemaphore.Init(0, 1); // Ensure this is initialized before the MyISR interrupt is enabled.
    while(1)
    {
        // Wait until we get notification from the interrupt
        clSemaphore.Pend();

        // Interrupt has fired, do the necessary work in this thread's context
        HeavyLifting();
    }
}

```

5.5 Mutexes

Mutexes (Mutual exclusion objects) are provided as a means of creating "protected sections" around a particular resource, allowing for access of these objects to be serialized. Only one thread can hold the mutex at a time - other threads have to wait until the region is released by the owner thread before they can take their turn operating on the protected resource. Note that mutexes can only be owned by threads - they are not available to other contexts (i.e. interrupts). Calling the mutex APIs from an interrupt will cause catastrophic system failures.

Note that these objects are also not recursive- that is, the owner thread can not attempt to claim a mutex more than once.

Priority inheritance is provided with these objects as a means to avoid priority inversions. Whenever a thread at a priority than the mutex owner blocks on a mutex, the priority of the current thread is boosted to the highest-priority waiter to ensure that other tasks at intermediate priorities cannot artificially prevent progress from being made.

Mutex objects are very easy to use, as there are only three operations supported: Initialize, Claim and Release. An example is shown below.

```

Mutex clMutex; // Create a mutex globally.

void Init()
{
    // Initialize the mutex before use.
    clMutex.Init();
}

// Some function called from a thread
void Thread1Function()
{
    clMutex.Claim();

    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex

    my_protected_resource.do_something();
    my_protected_resource.do_something_else();

    clMutex.Release();
}

// Some function called from another thread
void Thread2Function()
{
    clMutex.Claim();

    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex

    my_protected_resource.do_something();
    my_protected_resource.do_different_things();

    clMutex.Release();
}

```

5.6 Event Flags

Event Flags are another synchronization object, conceptually similar to a semaphore.

Unlike a semaphore, however, the condition on which threads are unblocked is determined by a more complex set of rules. Each Event Flag object contains a 16-bit field, and threads block, waiting for combinations of bits within this field to become set.

A thread can wait on any pattern of bits from this field to be set, and any number of threads can wait on any number of different patterns. Threads can wait on a single bit, multiple bits, or bits from within a subset of bits within the field.

As a result, setting a single value in the flag can result in any number of threads becoming unblocked simultaneously. This mechanism is extremely powerful, allowing for all sorts of complex, yet efficient, thread synchronization schemes that can be created using a single shared object.

Note that Event Flags can be set from interrupts, but you cannot wait on an event flag from within an interrupt.

Examples demonstrating the use of event flags are shown below.

```
// Simple example showing a thread blocking on a multiple bits in the
// fields within an event flag.

EventFlag clEventFlag;

int main()
{
    ...
    clEventFlag.Init(); // Initialize event flag prior to use
    ...
}

void MyInterrupt()
{
    // Some interrupt corresponds to event 0x0020
    clEventFlag.Set(0x0020);
}

void MyThreadFunc()
{
    ...
    while(1)
    {
        ...
        uint16_t ul6WakeCondition;

        // Allow this thread to block on multiple flags
        ul6WakeCondition = clEventFlag.Wait(0x00FF, EVENT_FLAG_ANY);

        // Clear the event condition that caused the thread to wake (in this case,
        // ul6WakeCondition will equal 0x20 when triggered from the interrupt above)
        clEventFlag.Clear(ul6WakeCondition);

        // <do something>
    }
}
```

5.7 Messages

Sending messages between threads is the key means of synchronizing access to data, and the primary mechanism to perform asynchronous data processing operations.

Sending a message consists of the following operations:

- Obtain a [Message](#) object from the global message pool
- Set the message data and event fields
- Send the message to the destination message queue

While receiving a message consists of the following steps:

- Wait for a messages in the destination message queue
- Process the message data
- Return the message back to the global message pool

These operations, and the various data objects involved are discussed in more detail in the following section.

5.7.1 Message Objects

[Message](#) objects are used to communicate arbitrary data between threads in a safe and synchronous way.

The message object consists of an event code field and a data field. The event code is used to provide context to the message object, while the data field (essentially a void * data pointer) is used to provide a payload of data corresponding to the particular event.

Access to these fields is marshalled by accessors - the transmitting thread uses the `SetData()` and `SetCode()` methods to seed the data, while the receiving thread uses the `GetData()` and `GetCode()` methods to retrieve it.

By providing the data as a void data pointer instead of a fixed-size message, we achieve an unprecedented measure of simplicity and flexibility. Data can be either statically or dynamically allocated, and sized appropriately for the event without having to format and reformat data by both sending and receiving threads. The choices here are left to the user - and the kernel doesn't get in the way of efficiency.

It is worth noting that you can send messages to message queues from within ISR context. This helps maintain consistency, since the same APIs can be used to provide event-driven programming facilities throughout the whole of the OS.

5.7.2 Global Message Pool

To maintain efficiency in the messaging system (and to prevent over-allocation of data), a global pool of message objects is provided. The size of this message pool is specified in the implementation, and can be adjusted depending on the requirements of the target application as a compile-time option.

Allocating a message from the message pool is as simple as calling the `GlobalMessagePool::Pop()` Method.

Messages are returned back to the `GlobalMessagePool::Push()` method once the message contents are no longer required.

One must be careful to ensure that discarded messages always are returned to the pool, otherwise a resource leak can occur, which may cripple the operating system's ability to pass data between threads.

5.7.3 Message Queues

`Message` objects specify data with context, but do not specify where the messages will be sent. For this purpose we have a `MessageQueue` object. Sending an object to a message queue involves calling the `MessageQueue::Send()` method, passing in a pointer to the `Message` object as an argument.

When a message is sent to the queue, the first thread blocked on the queue (as a result of calling the `MessageQueue::Receive()` method) will wake up, with a pointer to the `Message` object returned.

It's worth noting that multiple threads can block on the same message queue, providing a means for multiple threads to share work in parallel.

5.7.4 Messaging Example

```
// Message queue object shared between threads
MessageQueue cMsgQ;

// Function that initializes the shared message queue
void MsgQInit()
{
    cMsgQ.Init();
}

// Function called by one thread to send message data to
// another
void TxMessage()
{
    // Get a message, initialize its data
    Message *pClMsg = GlobalMessagePool::Pop();

    pClMsg->SetCode(0xAB);
    pClMsg->SetData((void*)some_data);

    // Send the data to the message queue
    cMsgQ.Send(pClMsg);
}

// Function called in the other thread to block until
// a message is received in the message queue.
void RxMessage()
{
    Message *pClMsg;
```

```

// Block until we have a message in the queue
pclMsg = clMsgQ.Receive();

// Do something with the data once the message is received
pclMsg->GetCode();

// Free the message once we're done with it.
GlobalMessagePool::Push(pclMsg);
}

```

5.8 Mailboxes

Another form of IPC is provided by Mark3, in the form of Mailboxes and Envelopes.

Mailboxes are similar to message queues in that they provide a synchronized interface by which data can be transmitted between threads.

Where [Message](#) Queues rely on linked lists of lightweight message objects (containing only message code and a void* data-pointer), which are inherently abstract, Mailboxes use a dedicated blob of memory, which is carved up into fixed-size chunks called Envelopes (defined by the user), which are sent and received. Unlike message queues, mailbox data is copied to and from the mailboxes dedicated pool.

Mailboxes also differ in that they provide not only a blocking "receive" call, but also a blocking "send" call, providing the opportunity for threads to block on "mailbox full" as well as "mailbox empty" conditions.

All send/receive APIs support an optional timeout parameter if the `KERNEL_USE_TIMEOUTS` option has been configured in [mark3cfg.h](#)

5.8.1 Mailbox Example

```

// Create a mailbox object, and define a buffer that will be used to store the
// mailbox' envelopes.
static Mailbox clMbox;
static uint8_t aucMBoxBuffer[128];

...
void InitMailbox(void)
{
    // Initialize our mailbox, telling it to use our defined buffer for envelope
    // storage. Pass in the size of the buffer, and set the size of each
    // envelope to 16 bytes. This gives u16 a mailbox capacity of (128 / 16) = 8
    // envelopes.
    clMbox.Init((void*)aucMBoxBuffer, 128, 16);
}

...
void SendThread(void)
{
    // Define a buffer that we'll eventually send to the
    // mailbox. Note the size is the same as that of an
    // envelope.
    uint8_t aucTxBuf[16];

    while(1)
    {
        // Copy some data into aucTxBuf, a 16-byte buffer, the
        // same size as a mailbox envelope.
        ...

        // Deliver the envelope (our buffer) into the mailbox
        clMbox.Send((void*)aucTxBuf);
    }
}

...
void RecvThred(void)
{
    uint8_t aucRxBuf[16];

    while(1)
    {
        // Wait until there's a message in our mailbox. Once
        // there is a message, read it into our local buffer.
        cmMbox.Receive((void*)aucRxBuf);
    }
}

```



```

        // Do something with the contents of aucRxBuf, which now
        // contains an envelope of data read from the mailbox.
        ...
    }
}

```

5.9 Notification Objects

Notification objects are the most lightweight of all blocking objects supplied by Mark3.

using this blocking primitive, one or more threads wait for the notification object to be signalled by code elsewhere in the system (i.e. another thread or interrupt). Once the the notification has been signalled, all threads currently blocked on the object become unblocked.

5.9.1 Notification Example

```

static Notify clNotifier;

...
void MyThread(void *unused_)
{
    // Initialize our notification object before use
    clNotifier.Init();

    while (1)
    {
        // Wait until our thread has been notified that it
        // can wake up.
        clNotifier.Wait();

        ...
        // Thread has woken up now -- do something!
    }
}

...
void SignalCallback(void)
{
    // Something in the system (interrupt, thread event, IPC,
    // etc.,) has called this function. As a result, we need
    // our other thread to wake up. Call the Notify object's
    // Signal() method to wake the thread up. Note that this
    // will have no effect if the thread is not presently
    // blocked.

    clNotifier.Signal();
}

```

5.10 Sleep

There are instances where it may be necessary for a thread to poll a resource, or wait a specific amount of time before proceeding to operate on a peripheral or volatile piece of data.

While the [Timer](#) object is generally a better choice for performing time-sensitive operations (and certainly a better choice for periodic operations), the [Thread::Sleep\(\)](#) method provides a convenient (and efficient) mechanism that allows for a thread to suspend its execution for a specified interval.

Note that when a thread is sleeping it is blocked, during which other threads can operate, or the system can enter its idle state.

```

int GetPeripheralData()
{
    int value;
    // The hardware manual for a peripheral specifies that
    // the "foo()" method will result in data being generated
    // that can be captured using the "bar()" method.
    // However, the value only becomes valid after 10ms

    peripheral.foo();
    Thread::Sleep(10); // Wait 10ms for data to become valid
    value = peripheral.bar();
}

```

```
    return value;  
}
```

5.11 Round-Robin Quantum

Threads at the same thread priority are scheduled using a round-robin scheme. Each thread is given a timeslice (which can be configured) of which it shares time amongst ready threads in the group. Once a thread's timeslice has expired, the next thread in the priority group is chosen to run until its quantum has expired - the cycle continues over and over so long as each thread has work to be done.

By default, the round-robin interval is set at 4ms.

This value can be overridden by calling the thread's `SetQuantum()` with a new interval specified in milliseconds.

Chapter 6

Why Mark3?

My first job after graduating from university in 2005 was with a small company that had a very old-school, low-budget philosophy when it came to software development.

Every make-or-buy decision ended with "make" when it came to tools. It was the kind of environment where vendors cost us money, but manpower was free. In retrospect, we didn't have a ton of business during the time that I worked there, and that may have had something to do with the fact that we were constantly short on ready cash for things we could code ourselves.

Early on, I asked why we didn't use industry-standard tools - like JTAG debuggers or IDEs. One senior engineer scoffed that debuggers were tools for wimps - and something that a good programmer should be able to do without. After all - we had serial ports, GPIOs, and a bi-color LED on our boards. Since these were built into the hardware, they didn't cost us a thing. We also had a single software "build" server that took 5 minutes to build a 32k binary on its best days, so when we had to debug code, it was a painful process of trial and error, with lots of Youtube between iterations. We complained that tens of thousands of dollars of productivity was being flushed away that could have been solved by implementing a proper build server - and while we eventually got our wish, it took far more time than it should have.

Needless to say, software development was painful at that company. We made life hard on ourselves purely out of pride, and for the right to say that we walked "up-hills both ways through 3 feet of snow, everyday". Our code was tied ever-so-tightly to our hardware platform, and the system code was indistinguishable from the application. While we didn't use an RTOS, we had effectively implemented a 3-priority threading scheme using a carefully designed interrupt nesting scheme with event flags and a while(1) superloop running as a background thread. Nothing was abstracted, and the code was always optimized for the platform, presumably in an effort to save on code size and wasted cycles. I asked why we didn't use an RTOS in any of our systems and received dismissive scoffs - the overhead from thread switching and maintaining multiple threads could not be tolerated in our systems according to our chief engineers. In retrospect, our ad-hoc system was likely as large as my smallest kernel, and had just as much context switching (although it was hidden by the compiler).

And every time a new iteration of our product was developed, the firmware took far too long to bring up, because the algorithms and data structures had to be re-tooled to work with the peripherals and sensors attached to the new boards. We worked very hard in an attempt to reinvent the wheel, all in the name of producing "efficient" code.

Regardless, I learned a lot about embedded software development.

Most important, I learned that good design is the key to good software; and good design doesn't have to come at a price. In all but the smallest of projects, the well-designed, well-abstracted code is not only more portable, but it's usually smaller, easier to read, and easier to reuse.

Also, since we had all the time in the world to invest in developing our own tools, I gained a lot of experience building them, and making use of good, free PC tools that could be used to develop and debug a large portion of our code. I ended up writing PC-based device and peripheral simulators, state-machine frameworks, and abstractions for our horrible ad-hoc system code. At the end of the day, I had developed enough tools that I could solve a lot of our development problems without having to re-inventing the wheel at each turn. Gaining a background in how these tools worked gave me a better understanding of how to use them - making me more productive at the jobs that I've had since.

I am convinced that designing good software takes honest effort up-front, and that good application code cannot be written unless it is based on a solid framework. Just as the wise man builds his house on rocks, and not on sand, wise developers write applications based on a well-defined platforms. And while you can probably build a house using nothing but a hammer and sheer will, you can certainly build one a lot faster with all the right tools.

This conviction lead me to development my first RTOS kernel in 2009 - FunkOS. It is a small, yet surprisingly full-featured kernel. It has all the basics (semaphores, mutexes, round-robin and preemptive scheduling), and some pretty advanced features as well (device drivers and other middleware). However, it had two major problems - it doesn't scale well, and it doesn't support many devices.

While I had modest success with this kernel (it has been featured on some blogs, and still gets around 125 downloads a month), it was nothing like the success of other RTOS kernels like uC/OS-II and FreeRTOS. To be honest, as a one-man show, I just don't have the resources to support all of the devices, toolchains, and evaluation boards that a real vendor can. I had never expected my kernel to compete with the likes of them, and I don't expect Mark3 to change the embedded landscape either.

My main goal with Mark3 was to solve the technical shortfalls in the FunkOS kernel by applying my experience in kernel development. As a result, Mark3 is better than FunkOS in almost every way; it scales better, has lower interrupt latency, and is generally more thoughtfully designed (all at a small cost to code size).

Another goal I had was to create something easy to understand, that could be documented and serve as a good introduction to RTOS kernel design. The end result of these goals is the kernel as presented in this book - a full source listing of a working OS kernel, with each module completely documented and explained in detail.

Finally, I wanted to prove that a kernel written entirely in C++ could perform just as well as one written in C. Mark3 is fully benchmarked and profiled – you can see exactly how much it costs to call certain APIs or include various features in the kernel.

And in addition, the code is more readable and easier to understand as a result of making use of object-oriented concepts provided by C++. Applications are easier to write because common concepts are encapsulated into objects (Threads, Semaphores, Mutexes, etc.) with their own methods and data, as opposed to APIs which rely on lots of explicit pointer or handle-passing, type casting, and other operations that are typically considered "unsafe" or "advanced" topics in C.

Chapter 7

When should you use an RTOS?

7.1 The reality of system code

System code can be defined as the program logic required to manage, synchronize, and schedule all of the resources (CPU time, memory, peripherals, etc.) used by the application running on the CPU. And it's true that a significant portion of the code running on an embedded system will be system code. No matter how simple a system is, whether or not this logic is embedded directly into the application (bare-metal system), or included as part of a well-defined stack on which an application is written (RTOS-based); system code is still present, and it comes with a cost.

As an embedded systems is being designed, engineers have to decide which approach to take: Bare-metal, or RTOS. There are advantages and disadvantages to each – and a reasonable engineer should always perform a thorough analysis of the pros and cons of each - in the context of the given application - before choosing a path.

The following figure demonstrates the differences between the architecture of a bare-metal system and RTOS based system at a high level:

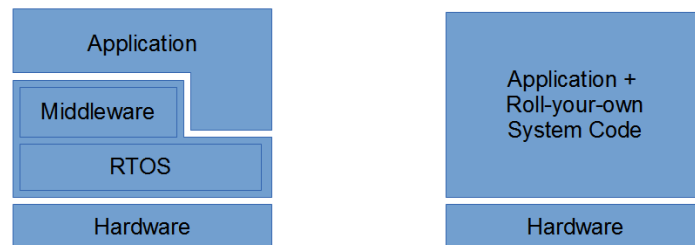


Figure 7.1: Arch

As can be seen, the RTOS (And associated middleware + libraries) captures a certain fixed size.

As a generalization, bare-metal systems typically have the advantage in that the system code overhead is small to start – but grows significantly as the application grows in complexity. At a certain point, it becomes extremely difficult and error-prone to add more functionality to an application running on such a system. There's a tipping point, where the cost of the code used to work-around the limitations of a bare-metal system outweigh the cost of a capable RTOS. Bare-metal systems also generally take longer to implement, because the system code has to be written from scratch (or derived from existing code) for the application. The resulting code also tend to be less portable, as it takes serious discipline to keep the system-specific elements of the code separated – in an RTOS-based system, once the kernel and drivers are ported, the application code is generally platform agnostic.

Conversely, an RTOS-based system incurs a slightly higher fixed cost up-front, but scales infinitely better than a bare-metal system as application's complexity increases. Using an RTOS for simple systems reduces application development time, but may cause an application not to fit into some extremely size-constrained microcontroller. An

RTOS can also cause the size of an application to grow more slowly relative to a bare-metal system – especially as a result of applying synchronization mechanisms and judicious IPC. As a result, an RTOS makes it significantly easier to “go agile” with an application – iteratively adding features and functionality, without having to consider refactoring the underlying system at each turn.

Some of these factors may be more important than others. Requirements, specifications, schedules, chip-selection, and volume projections for a project should all be used to feed into the discussions to decide whether or to go bare-metal or RTOS as a result.

Consider the following questions when making that decision:

- What is the application?
- How efficient is efficient enough?
- How fast is fast enough?
- How small is small enough?
- How responsive is responsive enough?
- How much code space/RAM/etc is available on the target system?
- How much code space/RAM do I need for an RTOS?
- How much code space/RAM do I think I'll need for my application?
- How much time do I have to deliver my system?
- How many units do we plan to sell?

7.2 Superloops, and their limitations

7.2.1 Intro to Superloops

Before we start taking a look at designing a real-time operating system, it's worthwhile taking a look through one of the most-common design patterns that developers use to manage task execution in bare-metal embedded systems - Superloops.

Systems based on superloops favor the system control logic baked directly into the application code, usually under the guise of simplicity, or memory (code and RAM) efficiency. For simple systems, superloops can definitely get the job done. However, they have some serious limitations, and are not suitable for every kind of project. In a lot of cases you can squeak by using superloops - especially in extremely constrained systems, but in general they are not a solid basis for reusable, portable code.

Nonetheless, a variety of examples are presented here- from the extremely simple, to cooperative and limited-preemptive multitasking systems, all of which are examples are representative of real-world systems that I've either written the firmware for, or have seen in my experience.

7.2.2 The simplest loop

Let's start with the simplest embedded system design possible - an infinite loop that performs a single task repeatedly:

```
int main()
{
    while(1)
    {
        Do_Something();
    }
}
```

Here, the code inside the loop will run a single function forever and ever. Not much to it, is there? But you might be surprised at just how much embedded system firmware is implemented using essentially the same mechanism - there isn't anything wrong with that, but it's just not that interesting.

Despite its simplicity we can see the beginnings of some core OS concepts. Here, the `while(1)` statement can be logically seen as the operating system kernel - this one control statement determines what tasks can run in the system, and defines the constraints that could modify their execution. But at the end of the day, that's a big part of what a kernel is - a mechanism that controls the execution of application code.

The second concept here is the task. This is application code provided by the user to perform some useful purpose in a system. In this case `Do_something()` represents that task - it could be monitoring blood pressure, reading a sensor and writing its data to a terminal, or playing an MP3; anything you can think of for an embedded system to do. A simple round-robin multi-tasking system can be built off of this example by simply adding additional tasks in sequence in the main `while`-loop. Note that in this example the CPU is always busy running tasks - at no time is the CPU idle, meaning that it is likely burning a lot of power.

While we conceptually have two separate pieces of code involved here (an operating system kernel and a set of running tasks), they are not logically separate. The OS code is indistinguishable from the application. It's like a single-celled organism - everything is crammed together within the walls of an indivisible unit; and specialized to perform its given function relying solely on instinct.

7.2.3 Interrupt-Driven Super-loop

In the previous example, we had a system without any way to control the execution of the task- it just runs forever. There's no way to control when the task can (or more importantly can't) run, which greatly limits the usefulness of the system. Say you only want your task to run every 100 milliseconds - in the previous code, you have to add a hard-coded delay at the end of your task's execution to ensure your code runs only when it should.

Fortunately, there is a much more elegant way to do this. In this example, we introduce the concept of the synchronization object. A Synchronization object is some data structure which works within the bounds of the operating system to tell tasks when they can run, and in many cases includes special data unique to the synchronization event.

There are a whole family of synchronization objects, which we'll get into later. In this example, we make use of the simplest synchronization primitive

- the global flag.

With the addition of synchronization brings the addition of event-driven systems. If you're programming a microcontroller system, you generally have scores of peripherals available to you - timers, GPIOs, ADCs, UARTs, ethernet, USB, etc. All of which can be configured to provide a stimulus to your system by means of interrupts. This stimulus gives us the ability not only to program our micros to `do_something()`, but to `do_something()` if-and-only-if a corresponding trigger has occurred.

The following concepts are shown in the example below:

```
volatile K_BOOL something_to_do = false;

__interrupt__ My_Interrupt_Source(void)
{
    something_to_do = true;
}

int main()
{
    while (1)
    {
        if (something_to_do)
        {
            Do_something();
            something_to_do = false;
        }
        else
        {
            Idle();
        }
    }
}
```

So there you have it - an event driven system which uses a global variable to synchronize the execution of our task based on the occurrence of an interrupt. It's still just a bare-metal, OS-baked-into-the-application system, but it's introduced a whole bunch of added complexity (and control!) into the system.

The first thing to notice in the source is that the global variable, `something_to_do`, is used as a synchronization object. When an interrupt occurs from some external event, triggering the `My_Interrupt_Source()` ISR, program flow in `main()` is interrupted, the interrupt handler is run, and `something_to_do` is set to true, letting us know that when we get back to `main()`, that we should run our `Do_something()` task.

Another new concept at play here is that of the idle function. In general, when running an event driven system, there are times when the CPU has no application tasks to run. In order to minimize power consumption, CPUs usually contain instructions or registers that can be set up to disable non-essential subsets of the system when there's nothing to do. In general, the sleeping system can be re-activated quickly as a result of an interrupt or other external stimulus, allowing normal processing to resume.

Now, we could just call `Do_something()` from the interrupt itself - but that's generally not a great solution. In general, the more time we spend inside an interrupt, the more time we spend with at least some interrupts disabled. As a result, we end up with interrupt latency. Now, in this system, with only one interrupt source and only one task this might not be a big deal, but say that `Do_something()` takes several seconds to complete, and in that time several other interrupts occur from other sources. While executing in our long-running interrupt, no other interrupts can be processed - in many cases, if two interrupts of the same type occur before the first is processed, one of these interrupt events will be lost. This can be utterly disastrous in a real-time system and should be avoided at all costs. As a result, it's generally preferable to use synchronization objects whenever possible to defer processing outside of the ISR.

Another OS concept that is implicitly introduced in this example is that of task priority. When an interrupt occurs, the normal execution of code in `main()` is preempted: control is swapped over to the ISR (which runs to completion), and then control is given back to `main()` where it left off. The very fact that interrupts take precedence over what's running shows that `main` is conceptually a "low-priority" task, and that all ISRs are "high-priority" tasks. In this example, our "high-priority" task is setting a variable to tell our "low-priority" task that it can do something useful. We will investigate the concept of task priority further in the next example.

Preemption is another key principle in embedded systems. This is the notion that whatever the CPU is doing when an interrupt occurs, it should stop, cache its current state (referred to as its context), and allow the high-priority event to be processed. The context of the previous task is then restored its state before the interrupt, and resumes processing. We'll come back to preemption frequently, since the concept comes up frequently in RTOS-based systems.

7.2.4 Cooperative multi-tasking

Our next example takes the previous example one step further by introducing cooperative multi-tasking:

```
// Bitfield values used to represent three distinct tasks
#define TASK_1_EVENT (0x01)
#define TASK_2_EVENT (0x02)
#define TASK_3_EVENT (0x04)

volatile K_UCHAR event_flags = 0;

// Interrupt sources used to trigger event execution

__interrupt__ My_Interrupt_1(void)
{
    event_flags |= TASK_1_EVENT;
}

__interrupt__ My_Interrupt_2(void)
{
    event_flags |= TASK_2_EVENT;
}

__interrupt__ My_Interrupt_3(void)
{
    event_flags |= TASK_3_EVENT;
}

// Main tasks
int main(void)
{

```



```

while(1)
{
    while(event_flags)
    {
        if( event_flags & TASK_1_EVENT)
        {
            Do_Task_1();
            event_flags &= ~TASK_1_EVENT;
        } else if( event_flags & TASK_2_EVENT) {
            Do_Task_2();
            event_flags &= ~TASK_2_EVENT;
        } else if( event_flags & TASK_3_EVENT) {
            Do_Task_3();
            event_flags &= ~TASK_3_EVENT;
        }
    }
    Idle();
}
}

```

This system is very similar to what we had before - however the differences are worth discussing. First, we have stimulus from multiple interrupt sources: each ISR is responsible for setting a single bit in our global event flag, which is then used to control execution of individual tasks from within main().

Next, we can see that tasks are explicitly given priorities inside the main loop based on the logic of the if/else if structure. As long as there is something set in the event flag, we will always try to execute Task1 first, and only when Task1 isn't set will we attempt to execute Task2, and then Task3. This added logic provides the notion of priority. However, because each of these tasks exist within the same context (they're just different functions called from our main control loop), we don't have the same notion of preemption that we have when dealing with interrupts.

That means that even through we may be running Task2 and an event flag for Task1 is set by an interrupt, the CPU still has to finish processing Task2 to completion before Task1 can be run. And that's why this kind of scheduling is referred to as cooperative multitasking: we can have as many tasks as we want, but unless they cooperate by means of returning back to main, the system can end up with high-priority tasks getting starved for CPU time by lower-priority, long-running tasks.

This is one of the more popular Os-baked-into-the-application approaches, and is widely used in a variety of real-time embedded systems.

7.2.5 Hybrid cooperative/preemptive multi-tasking

The final variation on the superloop design utilizes software-triggered interrupts to simulate a hybrid cooperative/preemptive multitasking system. Consider the example code below.

```

// Bitfields used to represent high-priority tasks. Tasks in this group
// can preempt tasks in the group below - but not eachother.
#define HP_TASK_1(0x01)
#define HP_TASK_2(0x02)

volatile K_UCHAR hp_tasks = 0;

// Bitfields used to represent low-priority tasks.
#define LP_TASK_1(0x01)
#define LP_TASK_2(0x02)

volatile K_UCHAR lp_tasks = 0;

// Interrupt sources, used to trigger both high and low priority tasks.
__interrupt__ System_Interrupt_1(void)
{
    // Set any of the other tasks from here...
    hp_tasks |= HP_TASK_1;
    // Trigger the SWI that calls the High_Priority_Tasks interrupt handler
    SWI();
}

__interrupt__ System_Interrupt_n...(void)
{
    // Set any of the other tasks from here...
}

// Interrupt handler that is used to implement the high-priority event context
__interrupt__ High_Priority_Tasks(void)
{
}

```

```

// Enabled every interrupt except this one
Disable_My_Interrupt();
Enable_Interrupts();
while( hp_tasks)
{
    if( hp_tasks & HP_TASK_1)
    {
        HP_Task1();
        hp_tasks &= ~HP_TASK_1;
    }
    else if (hp_tasks & HP_TASK_2)
    {
        HP_Task2();
        hp_tasks &= ~HP_TASK_2;
    }
}
Restore_Interrupts();
Enable_My_Interrupt();
}

// Main loop, used to implement the low-priority events
int main(void)
{
    // Set the function to run when a SWI is triggered
    Set_SWI(High_Priority_Tasks);

    // Run our super-loop
    while(1)
    {
        while (lp_tasks)
        {
            if (lp_tasks & LP_TASK_1)
            {
                LP_Task1();
                lp_tasks &= ~LP_TASK_1;
            }
            else if (lp_tasks & LP_TASK_2)
            {
                LP_Task2();
                lp_tasks &= ~LP_TASK_2;
            }
        }
        Idle();
    }
}

```

In this example, `High_Priority_Tasks()` can be triggered at any time as a result of a software interrupt (SWI),. When a high-priority event is set, the code that sets the event calls the SWI as well, which instantly preempts whatever is happening in main, switching to the high-priority interrupt handler. If the CPU is executing in an interrupt handler already, the current ISR completes, at which point control is given to the high priority interrupt handler.

Once inside the HP ISR, all interrupts (except the software interrupt) are re-enabled, which allows this interrupt to be preempted by other interrupt sources, which is called interrupt nesting. As a result, we end up with two distinct execution contexts (main and `HighPriorityTasks()`), in which all tasks in the high-priority group are guaranteed to preempt main() tasks, and will run to completion before returning control back to tasks in main(). This is a very basic preemptive multitasking scenario, approximating a "real" RTOS system with two threads of different priorities.

7.3 Problems with superloops

As mentioned earlier, a lot of real-world systems are implemented using a superloop design; and while they are simple to understand due to the limited and obvious control logic involved, they are not without their problems.

7.3.1 Hidden Costs

It's difficult to calculate the overhead of the superloop and the code required to implement workarounds for blocking calls, scheduling, and preemption. There's a cost in both the logic used to implement workarounds (usually involving state machines), as well as a cost to maintainability that comes with breaking up into chunks based on execution time instead of logical operations. In moderate firmware systems, this size cost can exceed the overhead of a reasonably well-featured RTOS, and the deficit in maintainability is something that is measurable in terms of lost productivity through debugging and profiling.

7.3.2 Tightly-coupled code

Because the control logic is integrated so closely with the application logic, a lot of care must be taken not to compromise the separation between application and system code. The timing loops, state machines, and architecture-specific control mechanisms used to avoid (or simulate) preemption can all contribute to the problem. As a result, a lot of superloop code ends up being difficult to port without effectively simulating or replicating the underlying system for which the application was written. Abstraction layers can mitigate the risks, but a lot of care should be taken to fully decouple the application code from the system code.

7.3.3 No blocking Calls

In a super-loop environment, there's no such thing as a blocking call or blocking objects. Tasks cannot stop mid-execution for event-driven I/O from other contexts - they must always run to completion. If busy-waiting and polling are used as a substitute, it increases latency and wastes cycles. As a result, extra code complexity is often times necessary to work-around this lack of blocking objects, often times through implementing additional state machines. In a large enough system, the added overhead in code size and cycles can add up.

7.3.4 Difficult to guarantee responsiveness

Without multiple levels of priority, it may be difficult to guarantee a certain degree of real-time responsiveness without added profiling and tweaking. The latency of a given task in a priority-based cooperative multitasking system is the length of the longest task. Care must be taken to break tasks up into appropriate sized chunks in order to ensure that higher- priority tasks can run in a timely fashion - a manual process that must be repeated as new tasks are added in the system. Once again, this adds extra complexity that makes code larger, more difficult to understand and maintain due to the artificial subdivision of tasks into time-based components.

7.3.5 Limited preemption capability

As shown in the example code, the way to gain preemption in a superloop is through the use of nested interrupts. While this isn't unwieldy for two levels of priority, adding more levels beyond this becomes complicated. In this case, it becomes necessary to track interrupt nesting manually, and separate sets of tasks that can run within given priority loops - and deadlock becomes more difficult to avoid.

Chapter 8

Can you afford an RTOS?

8.1 Intro

Of course, since you're reading the manual for an RTOS that I've been developing over the course of the several years, you can guess that the conclusion that I draw.

If your code is of any sort of non-trivial complexity (say, at least a few- thousand lines), then a more appropriate question would be "can you afford not* to use an RTOS in your system?".

In short, there are simply too many benefits of an RTOS to ignore, the most important being:

Threading, along with priority and time-based scheduling Sophisticated synchronization objects and IPC Flexible, powerful Software Timers Ability to write more portable, decoupled code

Sure, these features have a cost in code space and RAM, but from my experience the cost of trying to code around a lack of these features will cost you as much - if not more. The results are often far less maintainable, error prone, and complex. And that simply adds time and cost. Real developers ship, and the RTOS is quickly becoming one of the standard tools that help keep developers shipping.

One of the main arguments against using an RTOS in an embedded project is that the overhead incurred is too great to be justified. Concerns over "wasted" RAM caused by using multiple stacks, added CPU utilization, and the "large" code footprint from the kernel cause a large number of developers to shun using a preemptive RTOS, instead favoring a non-preemptive, application-specific solution.

I believe that not only is the impact negligible in most cases, but that the benefits of writing an application with an RTOS can lead to savings around the board (code size, quality, reliability, and development time). While these other benefits provide the most compelling case for using an RTOS, they are far more challenging to demonstrate in a quantitative way, and are clearly documented in numerous industry-based case studies.

While there is some overhead associated with an RTOS, the typical arguments are largely unfounded when an RTOS is correctly implemented in a system. By measuring the true overhead of a preemptive RTOS in a typical application, we will demonstrate that the impact to code space, RAM, and CPU usage is minimal, and indeed acceptable for a wide range of CPU targets.

To illustrate just how little an RTOS impacts the size of an embedded software design we will look at a typical microcontroller project and analyze the various types of overhead associated with using a pre-emptive realtime kernel versus a similar non-preemptive event-based framework.

RTOS overhead can be broken into three distinct areas:

- Code space: The amount of code space eaten up by the kernel (static)
- Memory overhead: The RAM associated with running the kernel and application threads.
- Runtime overhead: The CPU cycles required for the kernel's functionality (primarily scheduling and thread switching)

While there are other notable reasons to include or avoid the use of an RTOS in certain applications (determinism,

responsiveness, and interrupt latency among others), these are not considered in this discussion - as they are difficult to consider for the scope of our "canned" application.

8.2 Application description

For the purpose of this comparison, we first create an application using the standard preemptive Mark3 kernel with 2 system threads running: A foreground thread and a background thread. This gives three total priority levels in the system - the interrupt level (high), and two application priority threads (medium and low), which is quite a common paradigm for microcontroller firmware designs. The foreground thread processes a variety of time-critical events at a fixed frequency, while the background thread processes lower priority, aperiodic events. When there are no background thread events to process, the processor enters its low-power mode until the next interrupt is acknowledged.

The contents of the threads themselves are unimportant for this comparison, but we can assume they perform a variety of realtime I/O functions. As a result, a number of device drivers are also implemented.

Code Space and Memory Overhead:

The application is compiled for an ATmega328p processor which contains 32kB of code space in flash, and 2kB of RAM, which is a lower-mid-range microcontroller in Atmel's 8-bit AVR line of microcontrollers. Using the AVR GCC compiler with -Os level optimizations, an executable is produced with the following code/RAM utilization:

```
Program: 27914 bytes
Data:    1313 bytes
```

An alternate version of this project is created using a custom "super-loop" kernel, which uses a single application thread and provides 2 levels of priority (interrupt and application). In this case, the event handler processes the different priority application events to completion from highest to lowest priority.

This approach leaves the application itself largely unchanged. Using the same optimization levels as the preemptive kernel, the code compiles as follows:

```
Program: 24886 bytes
Data:    750 bytes
```

At first glance, the difference in RAM utilization seems quite a lot higher for the preemptive mode version of the application, but the raw numbers don't tell the whole story.

The first issue is that the cooperative-mode total does not take into account the system stack - whereas these values are included in the totals for RTOS version of the project. As a result, some further analysis is required to determine how the stack sizes truly compare.

In cooperative mode, there is only one thread of execution - so considering that multiple event handlers are executed in turn, the stack requirements for cooperative mode is simply determined by those of the most stack-intensive event handler (ignoring stack use contributions due to interrupts).

In contrast, the preemptive kernel requires a separate stack for each active thread, and as a result the stack usage of the system is the sum of the stacks for all threads.

Since the application and idle events are the same for both preemptive and cooperative mode, we know that their (independent) stack requirements will be the same in both cases.

For cooperative mode, we see that the idle thread stack utilization is lower than that of the application thread, and so the application thread's determines the stack size requirement. Again, with the preemptive kernel the stack utilization is the sum of the stacks defined for both threads.

As a result, the difference in overhead between the two cases becomes the extra stack required for the idle thread - which in our case is (a somewhat generous) 128 bytes.

The numbers still don't add up completely, but looking into the linker output we see that the rest of the difference comes from the extra data structures used to manage the kernel in preemptive mode, and the kernel data itself.

Fixed kernel data costs:

```
--- 134 Bytes Kernel data
--- 26 Bytes Kernel Vtables
```

Application (Variable) data costs:

```
--- 24 Bytes Driver Vtables
--- 123 Bytes - statically-allocated kernel objects (semaphores, timers, etc.)
```

With this taken into account, the true memory cost of a 2-thread system ends up being around 428 bytes of RAM - which is about 20% of the total memory available on this particular microcontroller. Whether or not this is reasonable certainly depends on the application, but more importantly, it is not so unreasonable as to eliminate an RTOS-based solution from being considered. Also note that by using the “simulated idle” feature provided in Mark3 R3 and onward, the idle thread (and its associated stack) can be eliminated altogether to reduce the cost in constrained devices.

The difference in code space overhead between the preemptive and cooperative mode solutions is less of an issue. Part of this reason is that both the preemptive and cooperative kernels are relatively small, and even an average target device (like the Atmega328 we’ve chosen) has plenty of room.

Mark3 can be configured so that only features necessary for the application are included in the RTOS - you only pay for the parts of the system that you use. In this way, we can measure the overhead on a feature-by-feature basis, which is shown below for the kernel as configured for this application:

```
Kernel ..... 2563 Bytes
Synchronization Objects. 644 Bytes
Port ..... 974 Bytes
Features ..... 871 Bytes
```

The configuration tested in this comparison uses the thread/port module with timers, drivers, and semaphores, and mutexes, for a total kernel size of 5052 Bytes, with the rest of the code space occupied by the application.

As can be seen from the compiler’s output, the difference in code space between the two versions of the application is 3028 bytes - or about 9% of the available code space on the selected processor. While nearly all of this comes from the added overhead of the kernel, the rest of the difference comes the changes to the application necessary to facilitate the different frameworks. This also demonstrates that the system-software code size in the cooperative case is about 2024 bytes.

8.3 Runtime Overhead

On the cooperative kernel, the overhead associated with running the thread is the time it takes the kernel to notice a pending event flag and launch the appropriate event handler, plus the timer interrupt execution time.

Similarly, on the preemptive kernel, the overhead is the time it takes to switch contexts to the application thread, plus the timer interrupt execution time.

The timer interrupt overhead is similar for both cases, so the overhead then becomes the difference between the following:

Preemptive mode:

- Posting the semaphore that wakes the high-priority thread
- Performing a context switch to the high-priority thread

Cooperative mode:

- Setting the event flag from the timer interrupt
- Acknowledging the event from the event loop

coop – 438 cycles preempt – 764 cycles

Using a cycle-accurate AVR simulator (fIAVR) running with a simulated speed of 16MHz, we find the end-to-end event sequence time to be 27us for the cooperative mode scheduler and 48us for the preemptive, and a raw difference of 20us.

With a fixed high-priority event frequency of 30Hz, we achieve a runtime overhead of 611us per second, or 0.06% of the total available CPU time. Now, obviously this value would expand at higher event frequencies and/or slower CPU frequencies, but for this typical application we find the difference in runtime overhead to be negligible for a preemptive system.

8.4 Analysis

For the selected test application and platform, including a preemptive RTOS is entirely reasonable, as the costs are low relative to a non-preemptive kernel solution. But these costs scale relative to the speed, memory and code space of the target processor. Because of these variables, there is no "magic bullet" environment suitable for every application, but Mark3 attempts to provide a framework suitable for a wide range of targets.

On the one hand, if these tests had been performed on a higher-end microcontroller such as the ATmega1284p (containing 128kB of code space and 16kB of RAM), the overhead would be in the noise. For this type of resource-rich microcontroller, there would be no reason to avoid using the Mark3 preemptive kernel.

Conversely, using a lower-end microcontroller like an ATmega88pa (which has only 8kB of code space and 1kB of RAM), the added overhead would likely be prohibitive for including a preemptive kernel. In this case, the cooperative-mode kernel would be a better choice.

As a rule of thumb, if one budgets 25% of a microcontroller's code space/RAM for system code, you should only require at minimum a microcontroller with 16kB of code space and 2kB of RAM as a base platform for an RTOS. Unless there are serious constraints on the system that require much better latency or responsiveness than can be achieved with RTOS overhead, almost any modern platform is sufficient for hosting a kernel. In the event you find yourself with a microprocessor with external memory, there should be no reason to avoid using an RTOS at all.

Chapter 9

Mark3 Design Goals

9.1 Overview

9.1.1 Services Provided by an RTOS Kernel

At its lowest-levels, an operating system kernel is responsible for managing and scheduling resources within a system according to the application. In a typical thread-based RTOS, the resources involved is CPU time, and the kernel manages this by scheduling threads and timers. But capable RTOS kernels provide much more than just threading and timers.

In the following section, we discuss the Mark3 kernel architecture, all of its features, and a thorough discussion of how the pieces all work together to make an awesome RTOS kernel.

9.1.2 Guiding Principles of Mark3

Mark3 was designed with a number of over-arching principles, coming from years of experience designing, implementing, refining, and experimenting with RTOS kernels. Through that process I not only discovered what features I wanted in an RTOS, but how I wanted to build those features to look, work, and “feel”. With that understanding, I started with a clean slate and began designing a new RTOS. Mark3 is the result of that process, and its design goals can be summarized in the following guiding principles.

9.1.3 Be feature competitive

To truly be taken seriously as more than just a toy or educational tool, an RTOS needs to have a certain feature suite. While Mark3 isn't a clone of any existing RTOS, it should at least attempt parity with the most common software in its class.

Looking at its competitors, Mark3 as a kernel supports most, if not all of the compelling features found in modern RTOS kernels, including dynamic threads, tickless timers, efficient message passing, and multiple types of synchronization primitives.

9.1.4 Be highly configuration

Mark3 isn't a one-size-fits-all kernel – and as a result, it provides the means to build a custom kernel to suit your needs. By configuring the kernel at compile-time, Mark3 can be built to contain the optimal feature set for a given application. And since features can be configured individually, you only pay the code/RAM footprint for the features you actually use.

9.1.5 No external dependencies, no new language features

To maximize portability and promote adoption to new platforms, Mark3 is written in a widely supported subset of C++ that lends itself to embedded applications. It avoids RTTI, exceptions, templates, and libraries (C standard, STL, etc.), with all fundamental data structures and types implemented completely for use by the kernel. As a result, the portable parts of Mark3 should compile for any capable C++ toolchain.

9.1.6 Target the most popular hobbyist platforms available

Realistically, this means supporting the various Arduino-compatible target CPUs, including AVR and ARM Cortex-M series microcontrollers. As a result, the current default target for Mark3 is the atmega328p, which has 32KB of flash and 2KB of RAM. All decisions regarding default features, code size, and performance need to take that target system into account.

Mark3 integrates cleanly as a library into the Arduino IDE to support atmega328-based targets. Other AVR and Cortex-M targets can be supported using the port code provided in the source package.

9.1.7 Maximize determinism – but be pragmatic

Guaranteeing deterministic and predictable behavior is tough to do in an embedded system, and often comes with a heavy price tag in either RAM or code-space. With Mark3, we strive to keep the core kernel APIs and features as lightweight as possible, while avoiding algorithms that don't scale to large numbers of threads. We also achieve minimal latency by keeping interrupts enabled (operating out of the critical section) wherever possible.

In Mark3, the most important parts of the kernel are fixed-time, including thread scheduling and context switching. Operations that are not fixed time can be characterized as a function of their dependent data. For instances, the [Mutex](#) and [Semaphore](#) APIs operate in fixed time in the uncontested case, and execute in linear time for the contested case – where the speed of execution is dependent on the number of threads currently waiting on that object.

The caveat here is that while we want to minimize latency and time spent in critical sections, that has to be balanced against increases in code size, and uncontested-case performance.

9.1.8 Apply engineering principles – and that means discipline, measurement and verification

My previous RTOS, FunkOS, was designed to be very ad-hoc. The usage instructions were along the lines of “drag and drop the source files into your IDE and compile”. There was no regression/unit testing, no code size/speed profiling, and all documentation was done manually. It worked, but the process was a bit of a mess, and resulted in a lot of re-spins of the software, and a lot of time spent stepping through emulators to measure parameters.

We take a different approach in Mark3. Here, we've designed not only the kernel-code, but the build system, unit tests, profiling code, documentation and reporting that supports the kernel. Each release is built and tested using automation in order to ensure quality and correctness, with supporting documentation containing all critical metrics. Only code that passes testing is submitted to the repos and public forums for distribution. These metrics can be traced from build-to-build to ensure that performance remains consistent from one drop to the next, and that no regressions are introduced by new/refactored code.

And while the kernel code can still be exported into an IDE directly, that takes place with the knowledge that the kernel code has already been rigorously tested and profiled. Exporting source in Mark3 is also supported by scripting to ensure reliable, reproducible results without the possibility for human-error.

9.1.9 Use Virtualization For Verification

Mark3 was designed to work with automated simulation tools as the primary means to validate changes to the kernel, due to the power and flexibility of automatic tests on virtual hardware. I was also intrigued by the thought of extending the virtual target to support functionality useful for a kernel, but not found on real hardware.

When the project was started, simavr was the tool of choice- however, its simulation was found to be incorrect compared to execution on a real MCU, and it did not provide the degree of extension that I desired for use with kernel development.

The flAVR AVR simulator was written to replace the dependency on that tool, and overcome those limitations. It also provides a GDB interface, as well as its own built-in debugger, profilers, and trace tools.

The example and test code relies heavily on flAVR kernel aware messaging, so it is recommended that you familiarize yourself with that tool if you intend to do any sort of customizations or extensions to the kernel.

flAVR is hosted on sourceforge at <http://www.sourceforge.net/projects/flavr/> . In its basic configuration, it builds with minimal external dependencies.

- On linux, it requires only pthreads.
- On Windows, it requires pthreads and ws2_32, both satisfied via MinGW.
- Optional SDL builds for both targets (featuring graphics and simulated joystick input) can be built, and rely on libSDL.

Chapter 10

Mark3 Kernel Architecture

10.1 Overview

At a high level, the Mark3 RTOS is organized into the following features, and layered as shown below:

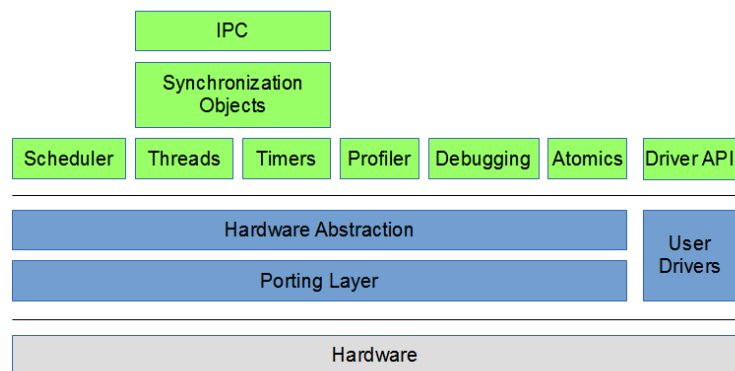


Figure 10.1: Overview

Everything in the “green” layer represents the Mark3 public API and classes, beneath which lives all hardware abstraction and CPU-specific porting and driver code, which runs on a given target CPU.

The features and concepts introduced in this diagram can be described as follows:

Threads: The ability to multiplex the CPU between multiple tasks to give the perception that multiple programs are running simultaneously. Each thread runs in its own context with its own stack.

Scheduler: Algorithm which determines the thread that gets to run on the CPU at any given time. This algorithm takes into account the priorities (and other execution parameters) associated with the threads in the system.

IPC: Inter-process-communications. Message-passing and [Mailbox](#) interfaces used to communicate between threads synchronously or asynchronously.

Synchronization Objects: Ability to schedule thread execution relative to system conditions and events, allowing for sharing global data and resources safely and effectively.

Timers: High-resolution software timers that allow for actions to be triggered on a periodic or one-shot basis.

Profiler: Special timer used to measure the performance of arbitrary blocks of code.

Debugging: Realtime logging and trace functionality, facilitating simplified debugging of systems using the OS.

Atomsics: Support for UN-interruptible arithmetic operations.

Driver API: Hardware abstraction interface allowing for device drivers to be written in a consistent, portable manner.

Hardware Abstraction Layer: Class interface definitions to represent threading, context-switching, and timers in a generic, abstracted manner.

Porting Layer: Class interface implementation to support threading, context-switching, and timers for a given CPU.

User Drivers: Code written by the user to implement device-specific peripheral drivers, built to make use of the Mark3 driver API.

Each of these features will be described in more detail in the following sections of this chapter.

The concepts introduced in the above architecture are implemented in a variety of source modules, which are logically broken down into classes (or in some cases, groups of functions/macros). The relationship between objects in the Mark3 kernel is shown below:

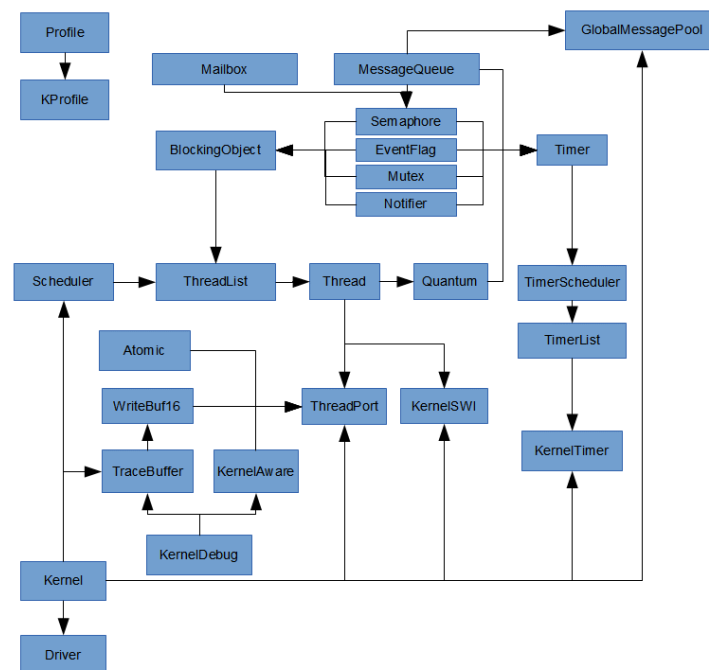


Figure 10.2: Overview

The objects shown in the preceding table can be grouped together by feature. In the table below, we group each feature by object, referencing the source module in which they can be found in the Mark3 source tree.

Feature	Kernel Object	Source Files
Profiling	ProfileTimer	profile.cpp/.h
Threads + Scheduling	Thread	thread.cpp/.h
	Scheduler	scheduler.cpp/.h/cpp
	Quantum	quantum.cpp/.h
	ThreadPort	threadport.cpp/.h **
	KernelSWI	kernelswi.cpp/.h **
Timers	Timer	timer.h/timer.cpp
	TimerScheduler	timerscheduler.h
	TimerList	timerlist.h/cpp
	KernelTimer	kernelswimer.cpp/.h **

Synchronization	BlockingObject	blocking.cpp/.h
	Semaphore	ksemaphore.cpp/.h
	EventFlag	eventflag.cpp/.h
	Mutex	mutex.cpp/.h
	Notify	notify.cpp/.h
IPC/Message-passing	Mailbox	mailbox.cpp/.h
	MessageQueue	message.cpp/.h
	GlobalMessagePool	message.cpp/.h
Debugging	Miscellaneous Macros	kerneldebug.h
	KernelAware	kernelaware.cpp/.h
	TraceBuffer	tracebuffer.cpp/.h
	Buffalogger	buffalogger.h
Device Drivers	Driver	driver.cpp/.h
Atomic Operations	Atomic	atomic.cpp/.h
Kernel	Kernel	kernel.cpp/.h

** implementation is platform-dependent, and located under the kernel's
 ** /cpu/<arch>/<variant>/<toolchain> folder in the source tree

10.2 Threads and Scheduling

The classes involved in threading and scheduling in Mark3 are highlighted in the following diagram, and are discussed in detail in this chapter:

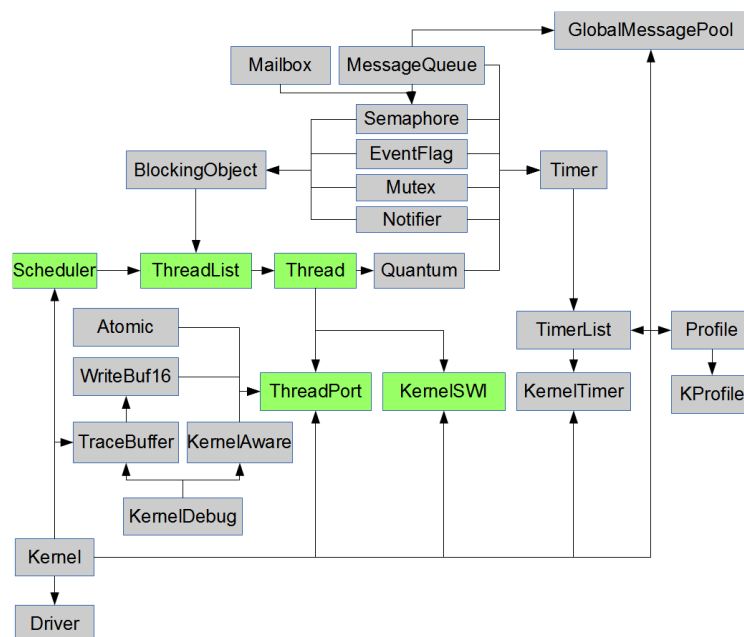


Figure 10.3: Threads and Scheduling

10.2.1 A Bit About Threads

Before we get started talking about the internals of the Mark3 scheduler, it's necessary to go over some background material - starting with: what is a thread, anyway?

Let's look at a very basic CPU without any sort of special multi-threading hardware, and without interrupts. When the CPU is powered up, the program counter is loaded with some default location, at which point the processor core will start executing instructions sequentially - running forever and ever according to whatever has been loaded into program memory. This single instance of a simple program sequence is the only thing that runs on the processor, and the execution of the program can be predicted entirely by looking at the CPU's current register state, its program, and any affected system memory (the CPU's "context").

It's simple enough, and that's exactly the definition we have for a thread in an RTOS.

Each thread contains an instance of a CPU's register context, its own stack, and any other bookkeeping information necessary to define the minimum unique execution state of a system at runtime. It is the job of a RTOS to multiplex the execution of multiple threads on a single physical CPU, thereby creating the illusion that many programs are being executed simultaneously. In reality there can only ever be one thread truly executing at any given moment on a CPU core, so it's up to the scheduler to set and enforce rules about what thread gets to run when, for how long, and under what conditions. As mentioned earlier, any system without an RTOS executes as a single thread, so at least two threads are required for an RTOS to serve any useful purpose.

Note that all of this information is common to pretty well every RTOS in existence - the implementation details, including the scheduler rules, are all part of what differentiates one RTOS from another.

10.2.2 Thread States and ThreadLists

Since only one thread can run on a CPU at a time, the scheduler relies on thread information to make its decisions. Mark3's scheduler relies on a variety of such information, including:

- The thread's current priority
- Round-Robin execution quanta
- Whether or not the thread is blocked on a synchronization object, such as a mutex or semaphore
- Whether or not the thread is currently suspended

The scheduler further uses this information to logically place each thread into 1 of 4 possible states:

- Ready - The thread is currently running
- Running - The thread is able to run
- Blocked - The thread cannot run until a system condition is met
- Stopped - The thread cannot run because its execution has been suspended

In order to determine a thread's state, threads are placed in "buckets" corresponding to these states. Ready and running threads exist in the scheduler's buckets, blocked threads exist in a bucket belonging to the object they're blocked on, and stopped threads exist in a separate bucket containing all stopped threads.

In reality, the various buckets are just doubly-linked lists of [Thread](#) objects - implemented in something called the [ThreadList](#) class. To facilitate this, the [Thread](#) class directly inherits from a [LinkListNode](#) class, which contains the node pointers required to implement a doubly-linked list. As a result, Threads may be effortlessly moved from one state to another using efficient linked-list operations built into the [ThreadList](#) class.

10.2.3 Blocking and Unblocking

While many developers new to the concept of an RTOS assume that all threads in a system are entirely separate from each other, the reality is that practical systems typically involve multiple threads working together, or at the very least sharing resources. In order to synchronize the execution of threads for that purpose, a number of synchronization primitives (blocking objects) are implemented to create specific sets of conditions under which threads can continue execution. The concept of "blocking" a thread until a specific condition is met is fundamental to understanding RTOS applications design, as well as any highly-multithreaded applications.

10.2.4 Blocking Objects

Blocking objects and primitives provided by Mark3 include:

- Semaphores (binary and counting)
- Mutexes
- Event Flags
- [Thread](#) Notification Objects
- [Thread](#) Sleep
- [Message](#) Queues
- Mailboxes

The relationship between these objects in the system are shown below:

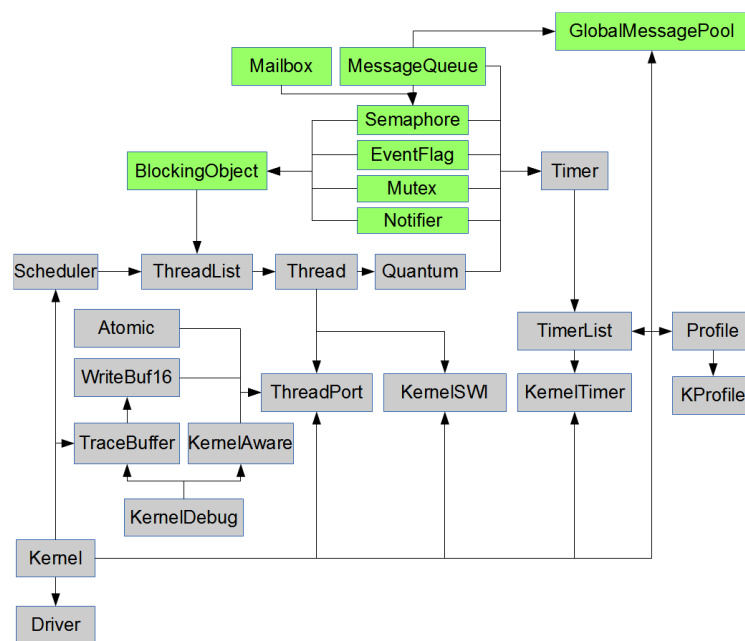


Figure 10.4: Blocking Objects

Each of these objects inherit from the [BlockingObject](#) class, which itself contains a [ThreadList](#) object. This class contains methods to `Block()` a thread (remove it from the [Scheduler](#)'s "Ready" or "Running" ThreadLists), as well as `UnBlock()` a thread (move a thread back to the "Ready" lists). This object handles transitioning threads from list-to-list (and state-to-state), as well as taking care of any other [Scheduler](#) bookkeeping required in the process. While each of the Blocking types implement a different condition, they are effectively variations on the same theme. Many simple Blocking objects are also used to build complex blocking objects - for instance, the [Thread](#) Sleep mechanism is essentially a binary semaphore and a timer object, while a message queue is a linked-list of message objects combined with a semaphore.

10.3 Inside the Mark3 Scheduler

At this point we've covered the following concepts:

- Threads
- [Thread](#) States and [Thread](#) Lists
- Blocking and Un-Blocking Threads

Thankfully, this is all the background required to understand how the Mark3 [Scheduler](#) works. In technical terms, Mark3 implements "strict priority scheduling, with round-robin scheduling among threads in each priority group". In plain English, this boils down to a scheduler which follows a few simple rules:

```
Find the highest-priority "Ready" list that has at least one Threads.
If the first thread in that bucket is not the current thread, select it
to run next. Otherwise, rotate the linked list, and choose the next
thread in the list to run
```

Since context switching is one of the most common and frequent operation performed by an RTOS, this needs to be as fast and deterministic as possible. While the logic is simple, a lot of care must be put into optimizing the scheduler to achieve those goals. In the section below we discuss the optimization approaches taken in Mark3.

There are a number of ways to find the highest-priority thread. The naive approach would be to simply iterate through the scheduler's array of ThreadLists from highest to lowest, stopping when the first non-empty list is found, such as in the following block of code:

```
for (prio = num_prio - 1; prio >= 0; prio--)
{
    if (thread_list[prio].get_head() != NULL)
    {
        break;
    }
}
```

While that would certainly work and be sufficient for a variety of systems, it's a non-deterministic approach (complexity $O(n)$) whose cost varies substantially based on how many priorities have to be evaluated. It's simple to read and understand, but it's non-optimal.

Fortunately, a functionally-equivalent and more deterministic approach can be implemented with a few tricks.

In addition to maintaining an array of ThreadLists, Mark3 also maintains a bitmap (one bit per priority level) that indicates which thread lists have ready threads. This bitmap is maintained automatically by the [ThreadList](#) class, and is updated every time a thread is moved to/from the [Scheduler's](#) ready lists.

By inspecting this bitmap using a technique to count the leading zero bits in the bitmap, we determine which threadlist to choose in fixed time.

Now, to implement the leading-zeros check, this can once again be performed iteratively using bitshifts and compares (which isn't any more efficient than the raw list traversal), but it can also be evaluated using either a lookup table, or via a special CPU instruction to count the leading zeros in a value. In Mark3, we opt for the lookup-table approach since we have a limited number of priorities and not all supported CPU architectures support a count leading zero instruction. To achieve a balance between performance and memory use, we use a 4-bit lookup table (costing 16 bytes) to perform the lookup.

(As a sidenote - this is actually a very common approach in OS schedulers. It's actually part of the reason why modern ARM cores implement a dedicated count-leading-zeros [CLZ] instruction!)

With a 4-bit lookup table and an 8-bit priority-level bitmap, the priority check algorithm looks something like this:

```
// Check the highest 4 priority levels, represented in the
// upper 4 bits in the bitmap
priority = priority_lookup_table[(priority_bitmap >> 4)];

// priority is non-zero if we found something there
if( priority )
```

```

{
    // Add 4 because we were looking at the higher levels
    priority += 4;
}
else
{
    // Nothing in the upper 4, look at the lowest 4 priority levels
    // represented by the lowest 4 bits in the bitmap
    priority = priority_lookup_table[priority_bitmap & 0x0F];
}

```

Deconstructing this algorithm, you can see that the priority lookup will have an $O(1)$ complexity - and is extremely low-cost.

This operation is thus fully deterministic and time bound - no matter how many threads are scheduled, the operation will always be time-bound to the most expensive of these two code paths. Even with only 8 priority levels, this is still much faster than iteratively checking the thread lists manually, compared with the previous example implementation.

Once the priority level has been found, selecting the next thread to run is trivial, consisting of something like this:

```
next_thread = thread_list[prio].get_head();
```

In the case of the `get_head()` calls, this evaluates to an inline-load of the "head" pointer in the particular thread list.

One important thing to take away from this analysis is that the scheduler is only responsible for selecting the next-to-run thread. In fact, these two operations are totally decoupled - no context switching is performed by the scheduler, and the scheduler isn't called from the context switch. The scheduler simply produces new "next thread" values that are consumed from within the context switch code.

10.3.1 Considerations for Round-Robin Scheduling

One thing that isn't considered directly from the scheduler algorithm is the problem of dealing with multiple threads within a single priority group; all of the algorithms that have been explored above simply look at the first [Thread](#) in each group.

Mark3 addresses this issue indirectly, using a software timer to manage round-robin scheduling, as follows.

In some instances where the scheduler is run by the kernel directly (typically as a result of calling [Thread::Yield\(\)](#)), the kernel will perform an additional check after running the [Scheduler](#) to determine whether or there are multiple ready [Threads](#) in the priority of the next ready thread.

If there are multiple threads within that priority, the kernel adds a one-shot software timer which is programmed to expire at the next [Thread](#)'s configured quantum. When this timer expires, the timer's callback function executes to perform two simple operations:

"Pivot" the current [Thread](#)'s priority list. Set a flag telling the kernel to trigger a Yield after exiting the main [Timer](#)↔[Scheduler](#) processing loop

Pivoting the thread list basically moves the head of a circular-linked-list to its next value, which in our case ensures that a new thread will be chosen the next time the scheduler is run (the scheduler only looks at the head node of the priority lists). And by calling Yield, the system forces the scheduler to run, a new round-robin software timer to be installed (if necessary), and triggers a context switch SWI to load the newly-chosen thread. Note that if the thread attached to the round-robin timer is pre-empted, the kernel will take steps to abort and invalidate that round-robin software timer, installing a new one tied to the next thread to run if necessary.

Because the round-robin software timer is dynamically installed when there are multiple ready threads at the highest ready priority level, there is no CPU overhead with this feature unless that condition is met. The cost of round-robin scheduling is also fixed - no matter how many threads there are, and the cost is identical to any other one-shot software timer in the system.

10.3.2 Context Switching

There's really not much to say about the actual context switch operation at a high level. Context switches are triggered whenever it has been determined that a new thread needs to be swapped into the CPU core when the scheduler is run. Mark3 implements also context switches as a call to a software interrupt - on AVR platforms, we

typically use INT0 or INT2 for this (although any pin-change GPIO interrupt can be used), and on ARM we achieve this by triggering a PendSV exception.

However, regardless of the architecture, the contex-switch ISR will perform the following three operations:

Save the current [Thread](#)'s context to the current [Thread](#) stack
Make the "next to run" thread the "currently running" thread
Restore the context of the next [Thread](#) from the [Thread](#) stack

The code to implement the context switch is entirely architecture-specific, so it won't be discussed in detail here. It's almost always gory inline-assembly which is used to load and store various CPU registers, and is highly-optimized for speed. We dive into an example implementation for the ARM Cortex-M0 microcontroller in a later section of this book.

10.3.3 Putting It All Together

In short, we can say that the Mark3 scheduler works as follows:

- The scheduler is run whenever a [Thread::Yield\(\)](#) is called by a user, as part of blocking calls, or whenever a new thread is started
- The Mark3 scheduler is deterministic, selecting the next thread to run in fixed-time
- The scheduler only chooses the next thread to run, the context switch SWI consumes that information to get that thread running
- Where there are multiple ready threads in the highest populated priority level, a software timer is used to manage round-robin scheduling

While we've covered a lot of ground in this section, there's not a whole lot of code involved. However, the code that performs these operations is nuanced and subtle. If you're interested in seeing how this all works in practice, I suggest reading through the Mark3 source code (which is heavily annotated), and stepping through the code with a simulator/emulator.

10.4 Timers

Mark3 implements one-shot and periodic software-timers via the [Timer](#) class. The user configures the timer for duration, repetition, and action, at which point the timer can be activated. When an active timer expires, the kernel calls a user-specified callback function, and then reloads the timer in the case of periodic timers. The same timer objects exposed to the user are also used within the kernel to implement round-robin scheduling, and timeout-based APIs for semaphores, mutexes, events, and messages.

Timers are implemented using the following components in the Mark3 [Kernel](#):

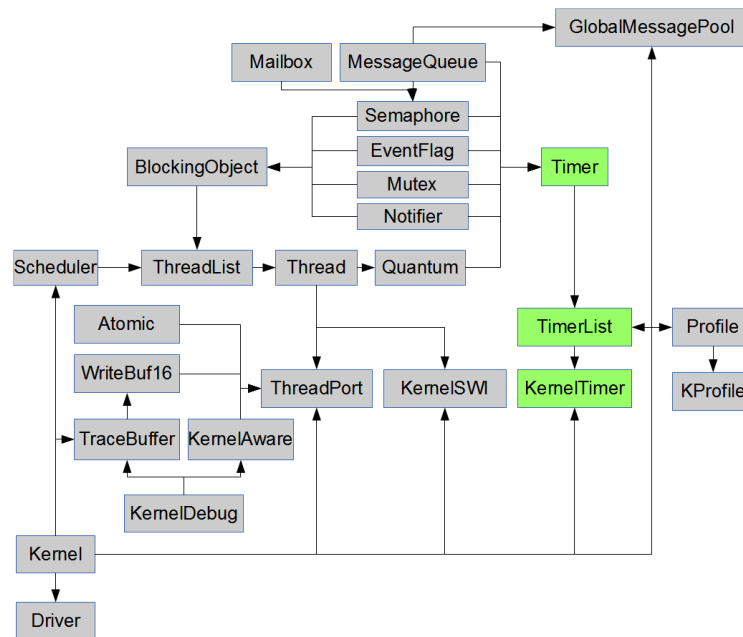


Figure 10.5: Timers

The [Timer](#) class provides the basic periodic and one-shot timer functionality used by application code, blocking objects, and IPC.

The [TimerList](#) class implements a doubly-linked list of [Timer](#) objects, and the logic required to implement a timer tick (tick-based kernel) or timer expiry (tickless kernel) event.

The [TimerScheduler](#) class contains a single [TimerList](#) object, implementing a single, system-wide list of [Timer](#) objects within the kernel. It also provides hooks for the hardware timer, such that when a timer tick or expiry event occurs, the [TimerList](#) expiry handler is run.

The [KernelTimer](#) class ([kerneltimer.cpp/.h](#)) implements the CPU specific hardware timer driver that is used by the kernel and the [TimerScheduler](#) to implement software timers.

While extremely simple to use, they provide one of the most powerful execution contexts in the system.

The software timers implemented in Mark3 use interrupt-nesting within the kernel timer's interrupt handler. This context is considered higher-priority than the highest priority user thread, but lower-priority than other interrupts in the system. As a result, this minimizes critical interrupt latency in the system, albeit at the expense of responsiveness of the user-threads.

For this reason, it's critical to ensure that all timer callback events are kept as short as possible to prevent adding thread-level latency. All heavy-lifting should be left to the threads, so the callback should only implement signalling via IPC or synchronization object.

The time spent in this interrupt context is also dependent on the number of active timers at any given time. However, Mark3 also can be used to minimize the frequency of these interrupts wakeups, by using an optional "tolerance" parameter in the timer API calls. In this way, periodic tasks that have less rigorous real-time constraints can all be grouped together – executing as a group instead of one-after-another.

Mark3 also contains two different timer implementations that can be configured at build-time, each with their own advantages.

10.4.1 Tick-based Timers

In a tick-based timing scheme, the kernel relies on a system-timer interrupt to fire at a relatively-high frequency, on which all kernel timer events are derived. On modern CPUs and microcontrollers, a 1kHz system tick is common, although quite often lower frequencies such as 60Hz, 100Hz, or 120Hz are used. The resolution of this timer also defines the maximum resolution of timer objects as a result. That is, if the timer frequency is 1kHz, a user cannot specify a timer resolution lower than 1ms.

The advantage of a tick-based timer is its sheer simplicity. It typically doesn't take much to set up a timer to trigger an interrupt at a fixed-interval, at which point, all system timer intervals are decremented by 1 count. When each system timer interval reaches zero, a callback is called for the event, and the events are either reset and restarted (repeated timers) or cleared (1-shot).

Unfortunately, that simplicity comes at a cost of increased interrupt count, which cause frequent CPU wakeups and utilization, and power consumption.

10.4.2 Tickless Timers

In a tickless system, the kernel timer only runs when there are active timers pending expiry, and even then, the timer module only generates interrupts when a timer expires, or a timer reaches its maximum count value. Additionally, when there are no active timer objects, the timer can be completely disabled – saving even more cycles, power, and CPU wakeups. These factors make the tickless timer approach a highly-optimal solution, suitable for a wide array of low-power applications.

Also, since tickless timers do not rely on a fixed, periodic clock, they can potentially be higher resolution. The only limitation in timer resolution is the precision of the underlying hardware timer as configured. For example, if a 32kHz hardware timer is being used to drive the timer scheduler, the resolution of timer objects would be in the $\sim 33\mu\text{s}$ range.

The only downside of the tickless timer system is an added complexity to the timer code, requiring more code space, and slightly longer execution of the timer routines when the timer interrupt is executed.

10.4.3 Timer Processing Algorithm

Timer interrupts occur at either a fixed-frequency (tick-based), or at the next timer expiry interval (tickless), at which point the timer processing algorithm runs. While the timer count is reset by the timer-interrupt, it is still allowed to accumulate ticks while this algorithm is executed in order to ensure that timer-accuracy is kept in real-time. It is also important to note that round-robin scheduling changes are disabled during the execution of this algorithm to prevent race conditions, as the round-robin code also relies on timer objects.

All active timer objects are stored in a doubly-linked list within the timer-scheduler, and this list is processed in two passes by the algorithm which runs from the timer-interrupt (with interrupt nesting enabled). The first pass determines which timers have expired and the next timer interval, while the second pass deals with executing the timer callbacks themselves. Both phases are discussed in more detail below.

In the first pass, the active timers are decremented by either 1 tick (tick-based), or by the duration of the last elapsed timer interval (tickless). Timers that have zero (or less-than-zero) time remaining have a "callback" flag set, telling the algorithm to call the timer's callback function in the second pass of the loop. In the event of a periodic timer, the timer's interval is reset to its starting value.

For the tickless case, the next timer interval is also computed in the first-pass by looking for the active timer with the least amount of time remaining in its interval. Note that this calculation is irrelevant in the tick-based timer code, as the timer interrupt fires at a fixed-frequency.

In the second pass, the algorithm loops through the list of active timers, looking for those with their "callback" flag set in the first pass. The callback function is then executed for each expired timer, and the "callback" flag cleared. In the event that a non-periodic (one-shot) timer expires, the timer is also removed from the timer scheduler at this time.

In a tickless system, once the second pass of the loop has been completed, the hardware timer is checked to see if the next timer interval has expired while processing the expired timer callbacks. In that event, the complete

algorithm is re-run to ensure that no expired timers are missed. Once the algorithm has completed without the next timer expiring during processing, the expiry time is programmed into the hardware timer. Round-robin scheduling is re-enabled, and if a new thread has been scheduled as a result of action taken during a timer callback, a context switch takes place on return from the timer interrupt.

10.5 Synchronization and IPC

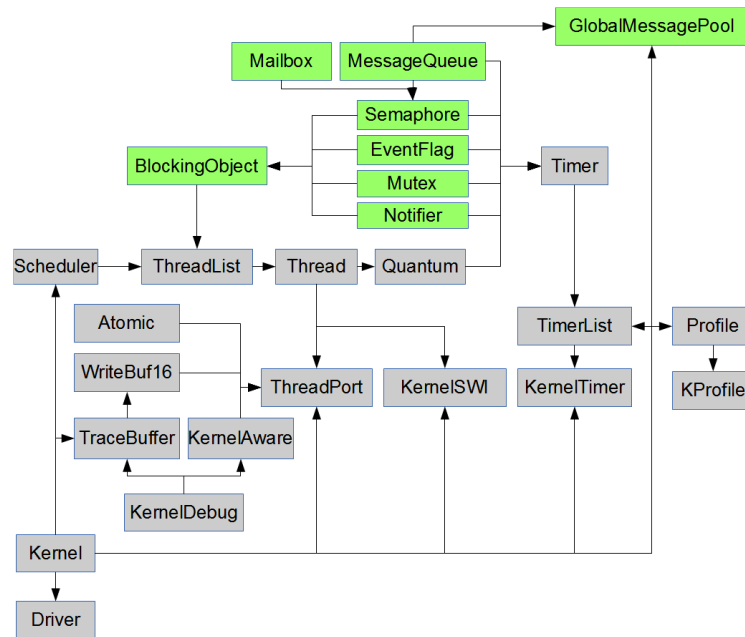


Figure 10.6: Synchronization and IPC

10.6 Blocking Objects

A Blocking object in Mark3 is essentially a thread list. Any blocking object implementation (being a semaphore, mutex, event flag, etc.) can be built on top of this class, utilizing the provided functions to manipulate thread location within the [Kernel](#).

Blocking a thread results in that thread becoming de-scheduled, placed in the blocking object's own private list of threads which are waiting on the object.

Unblocking a thread results in the reverse: The thread is moved back to its original location from the blocking list.

The only difference between a blocking object based on this class is the logic used to determine what constitutes a Block or Unblock condition.

For instance, a semaphore Pend operation may result in a call to the Block() method with the currently-executing thread in order to make that thread wait for a semaphore Post. That operation would then invoke the Unblock() method, removing the blocking thread from the semaphore's list, and back into the appropriate thread inside the scheduler.

Care must be taken when implementing blocking objects to ensure that critical sections are used judiciously, otherwise asynchronous events like timers and interrupts could result in non-deterministic and often catastrophic behavior.

Mark3 implements a variety of blocking objects including semaphores, mutexes, event flags, and IPC mechanisms that all inherit from the basic Blocking-object class found in [blocking.h/cpp](#), ensuring consistency and a high degree of code-reuse between components.

10.6.1 Semaphores

Semaphores are used to synchronized execution of threads based on the availability (and quantity) of application-specific resources in the system. They are extremely useful for solving producer-consumer problems, and are the method-of-choice for creating efficient, low latency systems, where ISRs post semaphores that are handled from within the context of individual threads. Semaphores can also be posted (but not pended) from within the interrupt context.

10.6.2 Mutex

Mutexes (Mutual exclusion objects) are provided as a means of creating "protected sections" around a particular resource, allowing for access of these objects to be serialized. Only one thread can hold the mutex at a time

- other threads have to wait until the region is released by the owner thread before they can take their turn operating on the protected resource. Note that mutexes can only be owned by threads - they are not available to other contexts (i.e. interrupts). Calling the mutex APIs from an interrupt will cause catastrophic system failures.

Note that these objects are recursive in Mark3 - that is, the owner thread can claim a mutex more than once. The caveat here is that a recursively-held mutex will not be released until a matching "release" call is made for each "claim" call.

Priority inheritance is provided with these objects as a means to avoid priority inversions. Whenever a thread at a priority than the mutex owner blocks on a mutex, the priority of the current thread is boosted to the highest-priority waiter to ensure that other tasks at intermediate priorities cannot artificially prevent progress from being made.

10.6.3 Event Flags

Event Flags are another synchronization object, conceptually similar to a semaphore.

Unlike a semaphore, however, the condition on which threads are unblocked is determined by a more complex set of rules. Each Event Flag object contains a 16-bit field, and threads block, waiting for combinations of bits within this field to become set.

A thread can wait on any pattern of bits from this field to be set, and any number of threads can wait on any number of different patterns. Threads can wait on a single bit, multiple bits, or bits from within a subset of bits within the field.

As a result, setting a single value in the flag can result in any number of threads becoming unblocked simultaneously. This mechanism is extremely powerful, allowing for all sorts of complex, yet efficient, thread synchronization schemes that can be created using a single shared object.

Note that Event Flags can be set from interrupts, but you cannot wait on an event flag from within an interrupt.

10.6.4 Notification Objects

Notification objects are the most lightweight of all blocking objects supplied by Mark3.

using this blocking primitive, one or more threads wait for the notification object to be signalled by code elsewhere in the system (i.e. another thread or interrupt). Once the notification has been signalled, all threads currently blocked on the object become unblocked and moved into the ready list.

Signalling a notification object that has no actively-waiting threads has no effect.

10.7 Messages and Global Message Queue

10.7.1 Messages

Sending messages between threads is the key means of synchronizing access to data, and the primary mechanism to perform asynchronous data processing operations.

Sending a message consists of the following operations:

- Obtain a [Message](#) object from the global message pool
- Set the message data and event fields
- Send the message to the destination message queue

While receiving a message consists of the following steps:

- Wait for a messages in the destination message queue
- Process the message data
- Return the message back to the global message pool

These operations, and the various data objects involved are discussed in more detail in the following section.

10.7.2 Message Objects

[Message](#) objects are used to communicate arbitrary data between threads in a safe and synchronous way.

The message object consists of an event code field and a data field. The event code is used to provide context to the message object, while the data field (essentially a void * data pointer) is used to provide a payload of data corresponding to the particular event.

Access to these fields is marshalled by accessors - the transmitting thread uses the `SetData()` and `SetCode()` methods to seed the data, while the receiving thread uses the `GetData()` and `GetCode()` methods to retrieve it.

By providing the data as a void data pointer instead of a fixed-size message, we achieve an unprecedented measure of simplicity and flexibility. Data can be either statically or dynamically allocated, and sized appropriately for the event without having to format and reformat data by both sending and receiving threads. The choices here are left to the user - and the kernel doesn't get in the way of efficiency.

It is worth noting that you can send messages to message queues from within ISR context. This helps maintain consistency, since the same APIs can be used to provide event-driven programming facilities throughout the whole of the OS.

10.7.3 Global Message Pool

To maintain efficiency in the messaging system (and to prevent over-allocation of data), a global pool of message objects is provided. The size of this message pool is specified in the implementation, and can be adjusted depending on the requirements of the target application as a compile-time option.

Allocating a message from the message pool is as simple as calling the

[GlobalMessagePool::Pop\(\)](#) Method.

Messages are returned back to the [GlobalMessagePool::Push\(\)](#) method once the message contents are no longer required.

One must be careful to ensure that discarded messages always are returned to the pool, otherwise a resource leak will occur, which may cripple the operating system's ability to pass data between threads.

10.7.4 Message Queues

[Message](#) objects specify data with context, but do not specify where the messages will be sent. For this purpose we have a [MessageQueue](#) object. Sending an object to a message queue involves calling the [MessageQueue::Send\(\)](#) method, passing in a pointer to the [Message](#) object as an argument.

When a message is sent to the queue, the first thread blocked on the queue (as a result of calling the [MessageQueue Receive\(\)](#) method) will wake up, with a pointer to the [Message](#) object returned.

It's worth noting that multiple threads can block on the same message queue, providing a means for multiple threads to share work in parallel.

10.7.5 Mailboxes

Another form of IPC is provided by Mark3, in the form of Mailboxes and Envelopes. Mailboxes are similar to message queues in that they provide a synchronized interface by which data can be transmitted between threads.

Where [Message](#) Queues rely on linked lists of lightweight message objects (containing only message code and a void* data-pointer), which are inherently abstract, Mailboxes use a dedicated blob of memory, which is carved up into fixed-size chunks called Envelopes (defined by the user), which are sent and received. Unlike message queues, mailbox data is copied to and from the mailboxes dedicated pool.

Mailboxes also differ in that they provide not only a blocking "receive" call, but also a blocking "send" call, providing the opportunity for threads to block on "mailbox full" as well as "mailbox empty" conditions.

All send/receive APIs support an optional timeout parameter if the `KERNEL_USE_TIMEOUTS` option has been configured in [mark3cfg.h](#)

10.7.6 Atomic Operations

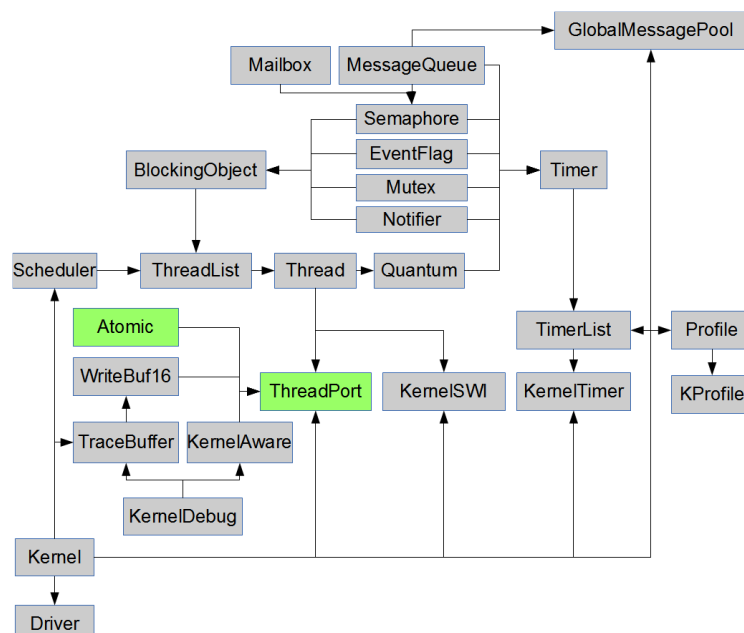


Figure 10.7: Atomic operations

This utility class provides primitives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primitives provided here include Set/Add/Delete for 8, 16, and 32-bit integer types, as well as an atomic test-and-set.

10.7.7 Drivers

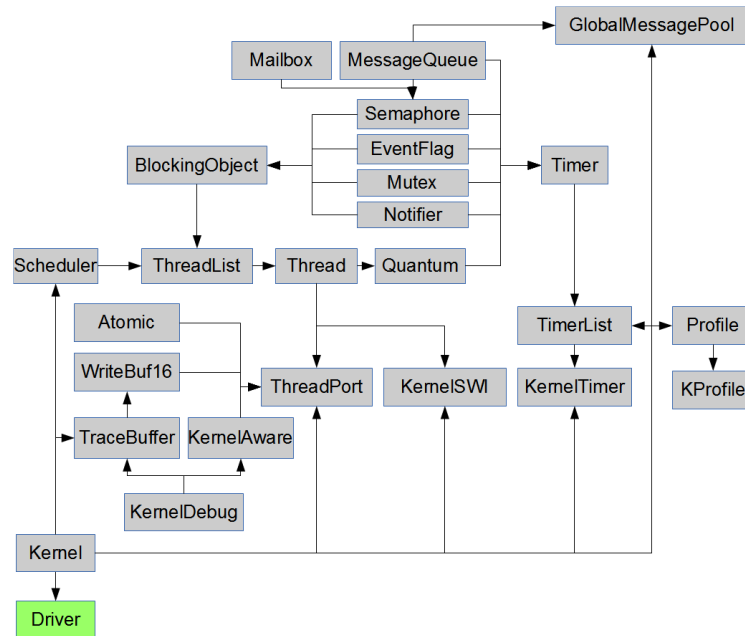


Figure 10.8: Drivers

This is the basis of the driver framework. In the context of Mark3, drivers don't necessarily have to be based on physical hardware peripherals. They can be used to represent algorithms (such as random number generators), files, or protocol stacks. Unlike FunkOS, where driver IO is protected automatically by a mutex, we do not use this kind of protection - we leave it up to the driver implementor to do what's right in its own context. This also frees up the driver to implement all sorts of other neat stuff, like sending messages to threads associated with the driver. Drivers are implemented as character devices, with the standard array of posix-style accessor methods for reading, writing, and general driver control.

A global driver list is provided as a convenient and minimal "filesystem" structure, in which devices can be accessed by name.

Driver Design

A device driver needs to be able to perform the following operations:

- Initialize a peripheral
- Start/stop a peripheral
- Handle I/O control operations
- Perform various read/write operations

At the end of the day, that's pretty much all a device driver has to do, and all of the functionality that needs to be presented to the developer.

We abstract all device drivers using a base-class which implements the following methods:

- Start/Open
- Stop/Close
- Control
- Read
- Write

A basic driver framework and API can thus be implemented in five function calls - that's it! You could even reduce that further by handling the initialize, start, and stop operations inside the "control" operation.

Driver API

In C++, we can implement this as a class to abstract these event handlers, with virtual void functions in the base class overridden by the inherited objects.

To add and remove device drivers from the global table, we use the following methods:

```
void DriverList::Add( Driver *pclDriver_ );
void DriverList::Remove( Driver *pclDriver_ );
```

`DriverList::Add()/Remove()` takes a single argument - the pointer to the object to operate on.

Once a driver has been added to the table, drivers are opened by NAME using `DriverList::FindByName("/dev/name")`. This function returns a pointer to the specified driver if successful, or to a built in /dev/null device if the path name is invalid. After a driver is open, that pointer is used for all other driver access functions.

This abstraction is incredibly useful - any peripheral or service can be accessed through a consistent set of APIs, that make it easy to substitute implementations from one platform to another. Portability is ensured, the overhead is negligible, and it emphasizes the reuse of both driver and application code as separate entities.

Consider a system with drivers for I2C, SPI, and UART peripherals - under our driver framework, an application can initialize these peripherals and write a greeting to each using the same simple API functions for all drivers:

```
pclI2C = DriverList::FindByName("/dev/i2c");
pclUART = DriverList::FindByName("/dev/tty0");
pclSPI = DriverList::FindByName("/dev/spi");

pclI2C->Write(12, "Hello World!");
pclUART->Write(12, "Hello World!");
pclSPI->Write(12, "Hello World!");
```

10.8 Kernel Proper and Porting

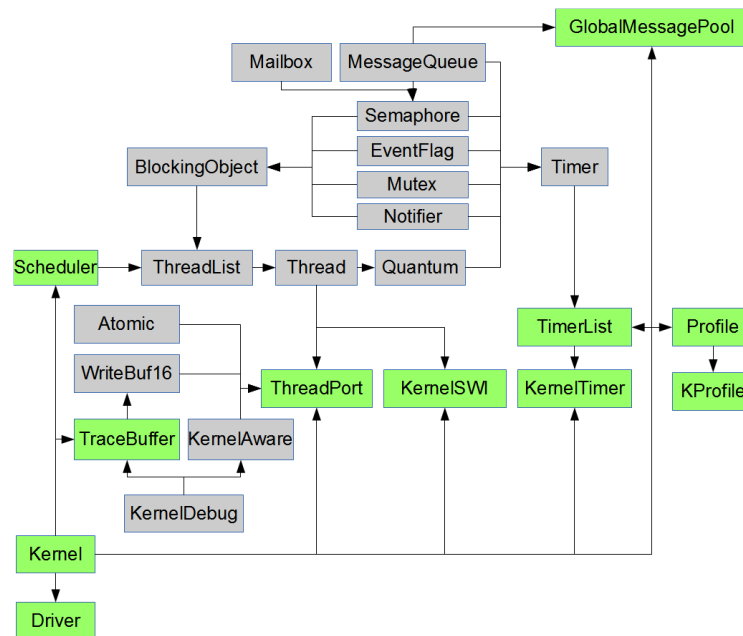


Figure 10.9: Kernel Proper and Porting

The `Kernel` class is a static class with methods to handle the initialization and startup of the RTOS, manage errors, and provide user-hooks for fatal error handling (functions called when `Kernel::Panic()` conditions are encountered), or when the Idle function is run.

Internally, `Kernel::Init()` calls the initialization routines for various kernel objects, providing a single interface by which all RTOS-related system initialization takes place.

`Kernel::Start()` is called to begin running OS functionality, and does not return. Control of the CPU is handed over to the scheduler, and the highest-priority ready thread begins execution in the RTOS environment.

Hardware Abstraction Layer

Almost all of the Mark3 kernel (and middleware) is completely platform independent, and should compile cleanly on any platform with a modern C++ compiler. However, there are a few areas within Mark3 that can only be implemented by touching hardware directly.

These interfaces generally cover four features:

- `Thread` initialization and context-switching logic
- Software interrupt control (used to generate context switches)
- Hardware timer control (support for time-based functionality, such as `Sleep()`)
- Code-execution profiling timer (not necessary to port if code-profiling is not compiled into the kernel)

The hardware abstraction layer in Mark3 provides a consistent interface for each of these four features. Mark3 is ported to new target architectures by providing an implementation for all of the interfaces declared in the abstraction layer. In the following section, we will explore how this was used to port the kernel to ARM Cortex-M0.

Real-world Porting Example – Cortex M0

This section serves as a real-world example of how Mark3 can be ported to new architectures, how the Mark3 abstraction layer works, and as a practical reference for using the RTOS support functionality baked in modern ARM Cortex-M series microcontrollers. Most of this documentation here is taken directly from the source code found in the `/kernel/cpu/cm0/ports` directory, with additional annotations to explain the port in more detail. Note that a familiarity with Cortex-M series parts will go a long way to understanding the subject matter presented, especially a basic understanding of the ARM CPU registers, exception models, and OS support features (PendSV, SysTick and SVC). If you're unfamiliar with ARM architecture, pay attention to the comments more than the source itself to illustrate the concepts.

Porting Mark3 to a new architecture consists of a few basic pieces; for developers familiar with the target architecture and the porting process, it's not a tremendously onerous endeavour to get Mark3 up-and-running somewhere new. For starters, all non-portable components are completely isolated in the source-tree under:

```
/embedded/kernel/CPU/VARIANT/TOOLCHAIN/
```

where CPU is the architecture, VARIANT is the vendor/part, and TOOLCHAIN is the compiler tool suite used to build the code.

From within the specific port folder, a developer needs only implement a few classes and headers that define the port-specific behavior of Mark3:

- [KernelSWI](#) ([kernelswi.cpp](#)/[kernelswi.h](#)) - Provides a maskable software-triggered interrupt used to perform context switching.
- [KernelTimer](#) ([kerneltimer.cpp](#)/[kerneltimer.h](#)) - Provides either a fixed-frequency or programmable-interval timer, which triggers an interrupt on expiry. This is used for implementing round-robin scheduling, thread-sleeps, and generic software timers.
- [Profiler](#) ([kprofile.cpp](#)/[kprofile.h](#)) - Contains code for runtime code-profiling. This is optional and may be stubbed out if left unimplemented (we won't cover profiling timers here).
- [ThreadPort](#) ([threadport.cpp](#)/[threadport.h](#)) - The meat-and-potatoes of the port code lives here. This class contains architecture/part-specific code used to initialize threads, implement critical-sections, perform context-switching, and start the kernel. Most of the time spent in this article focuses on the code found here.

Summarizing the above, these modules provide the following list of functionality:

```
- Thread stack initialization
- Kernel startup and first thread entry
- Context switch and SWI
- Kernel timers
- Critical Sections
.
```

The implementation of each of these pieces will be analyzed in detail in the sections that follow.

Thread Stack Initialization

Before a thread can be used, its stack must first be initialized to its default state. This default state ensures that when the thread is scheduled for the first time and its context restored, that it will cause the CPU to jump to the user's specified entry-point function.

All of the platform independent thread setup is handled by the generic kernel code. However, since every CPU architecture has its own register set, and stacks different information as part of an interrupt/exception, we have to implement this thread setup code for each platform we want the kernel to support (Combination of Architecture + Variant + Toolchain).

In the ARM Cortex-M0 architecture, the stack frame consists of the following information:

a) Exception Stack Frame

Contains the 8 registers which the ARM Cortex-M0 CPU automatically pushes to the stack when entering an exception. The following registers are included (in stack'd order):

```
[ XPSR ] <-- Highest address in context
[ PC   ]
```

```
[ LR  ]
[ R12 ]
[ R3  ]
[ R2  ]
[ R1  ]
[ R0  ]
```

XPSR – This is the CPU's status register. We need to set this to 0x01000000 (the "T" bit), which indicates that the CPU is executing in "thumb" mode. Note that ARMv6m and ARMv7m processors only run thumb2 instructions, so an exception is liable to occur if this bit ever gets cleared.

PC – Program Counter. This should be set with the initial entry point (function pointer) for that the user wishes to start executing with this thread.

LR - The base link register. Normally, this register contains the return address of the calling function, which is where the CPU jumps when a function returns. However, our threads generally don't return (and if they do, they're placed into the stop state). As a result we can leave this as 0.

The other registers in the stack frame are generic working registers, and have no special meaning, with the exception that R0 will hold the user's argument value passed into the entrypoint.

b) Complimentary CPU Register Context

```
[ R11 ]
...
[ R4  ] <-- Lowest address in context
```

These are the other general-purpose CPU registers that need to be backed up/ restored on a context switch, but aren't stacked by default on a Cortex-M0 exception. If there were any additional hardware registers to back up, then we'd also have to include them in this part of the context as well.

As a result, these registers all need to be manually pushed to the stack on stack creation, and will need to be explicitly pushed and pop as part of a normal context switch.

With this default exception state in mind, the following code is used to initialize a thread's stack for a Cortex-M0.

```
void ThreadPort::InitStack(Thread *pclThread_)
{
    K_ULONG *pulStack;
    K_ULONG *pulTemp;
    K_ULONG ulAddr;
    K_USHORT i;

    // Get the entrypoint for the thread
    ulAddr = (K_ULONG)(pclThread_>m_pfEntryPoint);

    // Get the top-of-stack pointer for the thread
    pulStack = (K_ULONG*)pclThread_>m_pwStackTop;

    // Initialize the stack to all FF's to aid in stack depth checking
    pulTemp = (K_ULONG*)pclThread_>m_pwStack;
    for (i = 0; i < pclThread_>m_usStackSize / sizeof(K_ULONG); i++)
    {
        pulTemp[i] = 0xFFFFFFFF;
    }

    PUSH_TO_STACK(pulStack, 0); // Apply one word of padding

    //-- Simulated Exception Stack Frame --
    PUSH_TO_STACK(pulStack, 0x01000000); // XPSR;set "T" bit for thumb-mode
    PUSH_TO_STACK(pulStack, ulAddr); // PC
    PUSH_TO_STACK(pulStack, 0); // LR
    PUSH_TO_STACK(pulStack, 0x12);
    PUSH_TO_STACK(pulStack, 0x3);
    PUSH_TO_STACK(pulStack, 0x2);
    PUSH_TO_STACK(pulStack, 0x1);
    PUSH_TO_STACK(pulStack, (K_ULONG)pclThread_>m_pvArg); // R0 = argument

    //-- Simulated Manually-Stacked Registers --
    PUSH_TO_STACK(pulStack, 0x11);
    PUSH_TO_STACK(pulStack, 0x10);
    PUSH_TO_STACK(pulStack, 0x09);
    PUSH_TO_STACK(pulStack, 0x08);
    PUSH_TO_STACK(pulStack, 0x07);
    PUSH_TO_STACK(pulStack, 0x06);
    PUSH_TO_STACK(pulStack, 0x05);
```

```

    PUSH_TO_STACK(pulStack, 0x04);
    pulStack++;

    pclThread->m_pwStackTop = pulStack;
}

```

Kernel Startup

The same general process applies to starting the kernel on an ARM Cortex-M0 as on other platforms. Here, we initialize and start the platform specific timer and software-interrupt modules, find the first thread to run, and then jump to that first thread.

Now, to perform that last step, we have two options:

1) Simulate a return from an exception manually to start the first thread, or.. 2) Use a software interrupt to trigger the first "Context Restore/Return from Interrupt"

For 1), we basically have to restore the whole stack manually, not relying on the CPU to do any of this for us. That's certainly doable, but not all Cortex parts support this (other members of the family support privileged modes, etc.). That, and the code required to do this is generally more complex due to all of the exception-state simulation. So, we will opt for the second option instead.

To implement a software to start our first thread, we will use the SVC instruction to generate an exception. From that exception, we can then restore the context from our first thread, set the CPU up to use the right "process" stack, and return-from-exception back to our first thread. We'll explore the code for that later.

But, before we can call the SVC exception, we're going to do a couple of things.

First, we're going to reset the default MSP stack pointer to its original top-of-stack value. The rationale here is that we no longer care about the data on the MSP stack, since calling the SVC instruction triggers a chain of events from which we never return. The MSP is also used by all exception-handling, so regaining a few words of stack here can be useful. We'll also enable all maskable exceptions at this point, since this code results in the kernel being started with the CPU executing the RTOS threads, at which point a user would expect interrupts to be enabled.

Note, the default stack pointer location is stored at address 0x00000000 on all ARM Cortex M0 parts. That explains the code below...

```

void ThreadPort_StartFirstThread( void )
{
    asm(
        " ldr r1, [r0] \n" // Reset the MSP to the default base address
        " msr msp, r1 \n"
        " cpsie i \n"      // Enable interrupts
        " svc 0 \n"        // Jump to SVC Call
    );
}

```

First Thread Entry

This handler has the job of taking the first thread object's stack, and restoring the default state data in a way that ensures that the thread starts executing when returning from the call.

We also keep in mind that there's an 8-byte offset from the beginning of the thread object to the location of the thread stack pointer. This offset is a result of the thread object inheriting from the linked-list node class, which has 8-bytes of data. This is stored first in the object, before the first element of the class, which is the "stack top" pointer.

The following assembly code shows how the SVC call is implemented in Mark3 for the purpose of starting the first thread.

```

get_thread_stack:
; Get the stack pointer for the current thread
ldr r0, g_pstCurrent
ldr r1, [r0]
add r1, #8
ldr r2, [r1]          ; r2 contains the current stack-top

load_manually_placed_context_r11_r8:
; Handle the bottom 32-bytes of the stack frame
; Start with r11-r8, because only r0-r7 can be used
; with ldmia on CM0.
add r2, #16
ldmia r2!, {r4-r7}

```



```

    mov r11, r7
    mov r10, r6
    mov r9, r5
    mov r8, r4

set_psp:
    ; Since r2 is coincidentally back to where the stack pointer should be,
    ; Set the program stack pointer such that returning from the exception handler
    msr psp, r2

load_manually_placed_context_r7_r4:
    ; Get back to the bottom of the manually stacked registers and pop.
    sub r2, #32
    ldmia r2!, {r4-r7} ; Register r4-r11 are restored.

set_thread_and_privilege_modes:
    ; Also modify the control register to force use of thread mode as well
    ; For CM3 forward-compatibility, also set user mode.
    mrs r0, control
    mov r1, #0x03
    orr r0, r1
    control, r0

set_lr:
    ; Set up the link register such that on return, the code operates
    ; in thread mode using the PSP. To do this, we or 0x0D to the value stored
    ; in the lr by the exception hardware EXC_RETURN. Alternately, we could
    ; just force lr to be 0xFFFFFFF0 (we know that's what we want from the
    ; hardware, anyway)
    mov r0, #0x0D
    mov r1, lr
    orr r0, r1

exit_exception:
    ; Return from the exception handler.
    ; The CPU will automatically unstack R0-R3, R12, PC, LR, and xPSR
    ; for us. If all goes well, our thread will start execution at the
    ; entrypoint, with the us-specified argument.
    bx r0

```

On ARM Cortex parts, there's dedicated hardware that's used primarily to support RTOS (or RTOS-like) functionality. This functionality includes the SysTick timer, and the PendSV Exception. SysTick is used for a tick-based kernel timer, while the PendSV exception is used for performing context switches. In reality, it's a "special SVC" call that's designed to be lower-overhead, in that it isn't mux'd with a bunch of other system or application functionality.

So how do we go about actually implementing a context switch here? There are a lot of different parts involved, but it essentially comes down to 3 steps:

1) Saving the context.

Thread's top-of-stack value is stored, all registers are stacked. We're good to go!

2) Swap threads

We swap the Scheduler's "next" thread with the "current" thread.

3) Restore Context

This is more or less identical to what we did when restoring the first context. Some operations may be optimized for data already stored in registers.

The code used to implement these steps on Cortex-M0 is presented below:

```

void PendSV_Handler(void)
{
    ASM(
        // Thread_SaveContext()
        " ldr r1, CURR_ \n"
        " ldr r1, [r1] \n "
        " mov r3, r1 \n "
        " add r3, #8 \n "

        // Grab the psp and adjust it by 32 based on extra registers we're going
        // to be manually stacking.
        " mrs r2, psp \n "
    )
}

```

```

" sub r2, #32 \n "

// While we're here, store the new top-of-stack value
" str r2, [r3] \n "

// And, while r2 is at the bottom of the stack frame, stack r7-r4
" stmia r2!, {r4-r7} \n "

// Stack r11-r8
" mov r7, r11 \n "
" mov r6, r10 \n "
" mov r5, r9 \n "
" mov r4, r8 \n "
" stmia r2!, {r4-r7} \n "

// Equivalent of Thread_Swap() - performs g_pstCurrent = g_pstNext
" ldr r1, CURR_ \n"
" ldr r0, NEXT_ \n"
" ldr r0, [r0] \n"
" str r0, [r1] \n"

// Thread_RestoreContext()
// Get the pointer to the next thread's stack
" add r0, #8 \n "
" ldr r2, [r0] \n "

// Stack pointer is in r2, start loading registers from
// the "manually-stacked" set
// Start with r11-r8, since these can't be accessed directly.
" add r2, #16 \n "
" ldmbia r2!, {r4-r7} \n "
" mov r11, r7 \n "
" mov r10, r6 \n "
" mov r9, r5 \n "
" mov r8, r4 \n "

// After subbing R2 #16 manually, and #16 through ldmbia, our PSP is where it
// needs to be when we return from the exception handler
" msr psp, r2 \n "

// Pop manually-stacked R4-R7
" sub r2, #32 \n "
" ldmbia r2!, {r4-r7} \n "

// lr contains the proper EXC_RETURN value
// we're done with the exception, so return back to newly-chosen thread
" bx lr \n "
" nop \n "

// Must be 4-byte aligned.
" NEXT_: .word g_pstNext \n"
" CURR_: .word g_pstCurrent \n"
);
}

```

Kernel Timers

ARM Cortex-M series microcontrollers each contain a SysTick timer, which was designed to facilitate a fixed-interval RTOS timer-tick. This timer is a precise 24-bit down-count timer, run at the main CPU clock frequency, that can be programmed to trigger an exception when the timer expires. The handler for this exception can thus be used to drive software timers throughout the system on a fixed interval.

Unfortunately, this hardware is extremely simple, and does not offer the flexibility of other timer hardware commonly implemented by MCU vendors - specifically a suitable timer prescaler that can be used to generate efficient, long-counting intervals. As a result, while the "generic" port of Mark3 for Cortex-M0 leverages the common SysTick timer interface, it only supports the tick-based version of the kernel's timer (note that specific Cortex-M0 ports such as the Atmel SAMD20 do have tickless timers).

Setting up a tick-based [KernelTimer](#) class to use the SysTick timer is, however, extremely easy, as is illustrated below:

```

void KernelTimer::Start(void)
{
    SysTick_Config(SYSTEM_FREQ / 1000); // 1KHz fixed clock...
    NVIC_EnableIRQ(SysTick_IRQn);
}

```

In [this](#) instance, the call to SysTick_Config() generates a 1kHz system-tick signal, and the NVIC_EnableIRQ() call ensures that a SysTick exception is generated for each tick. All other functions in the Cortex version of the [KernelTimer](#) class are essentially stubbed out (see the source for more details).

Note that the functions used in this call are part of the ARM Cortex Microcontroller Software Interface Standard (cmsis), and are supplied by all parts vendors selling Cortex hardware. This greatly simplifies the design of our port-code, since we can be reasonably assured that these APIs will work the same on all devices.

The handler code called when a SysTick exception occurs is basically the same as on other platforms (such as AVR), except that we explicitly clear the "exception pending" bit before returning. This is implemented in the following code:

```
\code{.cpp}
void SysTick_Handler(void)
{
    #if KERNEL_USE_TIMERS
        TimerScheduler::Process();
    #endif
    #if KERNEL_USE_QUANTUM
        Quantum::UpdateTimer();
    #endif

    // Clear the systick interrupt pending bit.
    SCB->ICSR |= SCB_ICSR_PENDSTCLR_Msk;
}
```

Critical Sections

A "critical section" is a block of code whose execution cannot be interrupted by means of context switches or an interrupt. In a traditional single-core operating system, it is typically implemented as a block of code where the interrupts are disabled - this is also the approach taken by Mark3. Given that every CPU has its own means of disabling/enabling interrupts, the implementation of the critical section APIs is also non-portable.

In the Cortex-M0 port, we implement the two critical section APIs ([CS_ENTER\(\)](#) and [CS_EXIT\(\)](#)) as function-like macros containing inline assembly. All uses of these calls are called in pairs within a function and must take place at the same level-of-scope. Also, as nesting may occur (critical section within a critical section), this must be taken into account in the code.

In general, [CS_ENTER\(\)](#) performs the following tasks:

- Cache the current interrupt-enabled state within a local variable in the thread's state
- Disable interrupts
- .

Conversely, [CS_EXIT\(\)](#) performs the following tasks:

- Read the original interrupt-enabled state from the cached value
- Restore interrupts to the original value
- .

On Cortex-M series microcontrollers, the PRIMASK special register contains a single status bit which can be used to enable/disable all maskable interrupts at once. This register can be read directly to examine or modify its state. For convenience, ARMv6m provides two instructions to enable/disable interrupts

- cpsid (disable interrupts) and cpsie (enable interrupts). Mark3 Implements these steps according to the following code:

```
//-----
#define CS_ENTER() \
{ \
    K_ULONG __ulRegState; \
    asm ( \
        " mrs r0, PRIMASK \n" \
        " mov %[STATUS], r0 \n" \
        " cpsid i \n" \
        : [STATUS] "=r" (__ulRegState) \
        ); \
}

//-----
#define CS_EXIT() \
asm ( \
    " mov r0, %[STATUS] \n" \
    " msr primask, r0 \n" \
    )
```

```
    : \
    : [STATUS] "r" (__ulRegState) \
    ); \
}
```

Summary

In this section we have investigated how the main non-portable areas of the Mark3 RTOS are implemented on a Cortex-M0 microcontroller. Mark3 leverages all of the hardware blocks designed to enable RTOS functionality on ARM Cortex-M series microcontrollers: the SVC call provides the mechanism by which we start the kernel, the PendSV exception provides the necessary software interrupt, and the SysTick timer provides an RTOS tick. As a result, Mark3 is a perfect fit for these devices - and as a result of this approach, the same RTOS port code should work with little to no modification on all ARM Cortex-M parts.

We have discussed what functionality in the RTOS is not portable, and what interfaces must be implemented in order to complete a fully-functional port. The five specific areas which are non-portable (stack initialization, kernel startup/entry, kernel timers, context switching, and critical sections) have been discussed in detail, with the platform-specific source provided as a practical reference to ARM-specific OS features, as well as Mark3's porting infrastructure. From this example (and the accompanying source), it should be possible for an experienced developers to create a port Mark3 to other microcontroller targets.

Chapter 11

Build System

In addition to providing a complete RTOS kernel with a variety of middleware, tests, and example code, Mark3 also provides a robust architecture to efficiently build these components.

The build system – including its design and use, are discussed in the following sections.

11.1 Introduction

As developers, we spend an awful lot of time talking about how our source code is written, but devote very little energy to what happens to the code after it's been written... aside from producing running executables. When I refer to “building better software”, I’m not talking about writing code – I’m talking about the technologies and processes that can be applied to manipulate source into a variety of products, including libraries, applications, tests, documentation, and performance data.

For a lot of developers – embedded or otherwise – a typical build process might look something like this:

Open the IDE, load a project and click “build”. Sometime later, check the output window and look to see that there aren't any red exclamation points to indicate build failure. Browse to your project's output folder to collect your prize: A brand new .elf file containing your new firmware! Click on the arrow to give it a quick run on your dev board, test it for a few minutes, and make sure it seems sane. Pass it off to the manufacturing guys to load it on the line, and move on. Next!

Okay, that's a bit of an exaggeration, but not too far-fetched; and not that much different from standard procedure at places I've worked in the past.

Indeed - I've come across many developers over the years who know about how their software gets built beyond the “black box” that turns their code from text to binaries with the click of the button – and they like it that way. It's entirely understandable, too. Developing from an IDE hides all those messy configuration details, command-line options, symbol definitions and environment variables that would otherwise take away from time spent actively churning out code. We all want to be more productive, of course, and it takes time to learn to make, or anything specific to an embedded toolchain.

And from a product delivery perspective, binaries are the ultimate work-products from a software team – these are the pieces that drive the microcontrollers, DSPs and CPUs in an embedded system. When its crunch time, try convincing management to back off on release date in order to ensure that documentation gets updated to reflect the as-built nature of a project. Or fix the gaps in test coverage. Or update wikis containing profiling and performance metrics. You get the picture.

But software is a living entity – it's constantly changing as it develops and is refined by individuals and teams. And source code is a medium that carries different information across multiple channels all at once – while one channel contains information about building an application, another contains information on building libraries. Another carries information on testing, and another still provides documentation relevant to consumers of the code. While not as glamorous a role as the “living firmware”, these pieces of critical metadata are absolutely necessary as they ensure that the firmware products maintain a degree of quality, performance, and conformance, and gives a degree of confidence before formal test and release activities take place.

This is especially necessary when developing for an organizations that is accountable for their development and documentation practices (for example, ISO shops), or to shareholders who expect the companies they support with their wallets to apply engineering rigour to their products.

But getting the kind of flexibility required to produce these alternative work products form the “example IDE” is not trivial, and can be difficult to apply consistently from project-to-project/IDE-to-IDE. Automating these test and documentation tasks should be considered mandatory if you care about making the most of your development hours; manually generating and updating documentation, tests, and profiling results wastes time that you could be spending solving the right kinds of problems.

The good news, though, is that using common tools available on any modern OS, you can create frameworks that make these tasks for any project, on any toolchain providing command-line tools. With a bit of make, shell-script, and python, you can automate any number of build processes in a way that yields consistent, reliable results that are transferrable from project to project.

This is the approach taken in the Mark3 project, which integrates build, testing, profiling, documentation and release processes together in order to produce predictable, verifiable, output that can be validated against quality gates prior to formal testing and release. Only code revisions that pass all quality gate can be released. In the following sections, we'll explore the phased build approach, and how it's used by the Mark3 project.

11.2 Mark3 Build Process Overview

Building software is by and large a serial process, as outputs from each build step are required in subsequent steps. We start from our source code, scripts, and makefiles, configure our environment, and use our tools to turn the source code from one form to another, leveraging the outputs from each stage in the generation of further work products – whether it be creating binaries, running tests, or packaging artifacts for release.

To simplify the design and illustrate the concepts involved, we can break down these serial process into the following distinct phases:

- Pre-build – Environment configuration, target selection, and header-file staging
- Build – Compiling libraries, and building binaries for applications and tests
- Test + Profiling - Running unit tests, integration tests, profiling code
- Release – Generation of documentation from source code and test results, packaging of build artifacts and headers

Each phase and associated activities are described in detail in the following subsections.

11.2.1 Pre-Build Phase:

Target Selection

Inputs: CPU Architecture, Variant, Toolchain variables Outputs: Environment, makefile configuration

In this phase, we select the runtime environment and configure all environment-specific variables. Specifying environment variables at this phase ensures that when the build scripts are run, the correct makefiles, libraries, binaries, and config files are used when generating outputs. This can also be used to ensure that common build setting are applied to all platform specific binaries, including optimization levels, debug symbols, linker files, and CPU flags.

Staging Headers

Inputs: All files with a .h extension, located in library or binary project /public folders Output: Headers copied to a common staging directory

In this step, header files from all platform libraries are copied to a common staging directory referenced by the build system.

This simplifies makefiles and build scripts, ensuring only a single include directory needs to be specified to gain access to all common platform libraries. This keeps library and application code clean, as relative paths can be

completely avoided. As an added benefit, these headers can later be deployed with the corresponding libraries to customers, giving them access to a set of pre-compiled libraries with APIs, but without providing the source.

11.2.2 Build Phase

Building Libraries

Input: Source code for all common libraries, staged headers Output: Static libraries that can be linked against applications Gate: All mandatory libraries must be built successfully

The project root directory is scanned recursively for directories containing makefiles. When a makefile is found in the root of a subdirectory and a library tag is encountered (in Mark3, this corresponds to the declaration "IS_LIB=1"), the project is built using the library-specific make commands for the platform. Libraries can reference other libraries implicitly, and include headers from the common include directory. Since references are resolved when building executable binary images, the executable projects are responsible for including the dependent libs.

Building Binaries

Input: Source code for individual applications, precompiled libraries, staged headers Output: Executable application and test binaries Gate: All mandatory binaries (applications and tests) must be built successfully

The project root directory is scanned recursively for directories containing makefiles. When a makefile is found in the root of a subdirectory and a binary tag is encountered (in Mark3, this corresponds to the declaration "IS_APP=1"), the project is built using the executable-specific make commands for the platform. Applications can reference all platform and toolchain libraries, and include headers from the common include directory. Care must be taken to ensure that all library dependencies are explicitly specified in the application's makefile's list.

This step will fail if necessary dependencies are not met (i.e. required libraries failed to build in a prior step).

Static Analysis:

Input: Source code for libraries/binaries Output: Static source analysis output Gate: N/A

Static analysis tools such as clang, klocwork, and lint can be run on the source to ensure that there are no critical or catastrophic problems (null pointer exceptions, variables used before initialization, incorrect argument usage, etc.) that wouldn't necessarily be caught at compile-time. Since tool availability and configurability varies, this isn't something that is enforced in the Mark3 builds. A user may opt to use clang to perform static code analysis on the build, however. The part-specific makefile contains a CLANG environment variable for this purpose.

Potential quality gates could be set up such that a failure during static analysis aborts the rest of the build.

Test + Profiling Sanity Tests

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output indicating test pass/failure status

11.2.3 Test and Profile

Unit Tests

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output indicating test pass/failure status

Code Performance Profiling

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output containing critical code performance metrics

Code Size Profiling

Input: Precompiled static libraries and binaries Output: Text output containing critical code size metrics

11.2.4 Release

Documentation

Input: Library source code and headers, commented with Doxygen tags, Profiling results, Test results Output: Doxygen-generated HTML and PDF documentation

Packaging

Input: Static libraries and application/test binaries, staged headers, compiled documentation Output: Archive (.zip) containing relevant build outputs

Chapter 12

Profiling Results

The following profiling results were obtained using an ATmega328p @ 16MHz.

The test cases are designed to make use of the kernel profiler, which accurately measures the performance of the fundamental system APIs, in order to provide information for user comparison, as well as to ensure that regressions are not being introduced into the system.

12.1 Date Performed

Mon Jan 11 21:42:30 EST 2016

12.2 Compiler Information

The kernel and test code used in these results were built using the following compiler: Using built-in specs. COLLECT_GCC=avr-gcc COLLECT_LTO_WRAPPER=/usr/lib/gcc/avr/4.8.2/lto-wrapper Target: avr Configured with: ../src/configure -v --enable-languages=c,c++ --prefix=/usr/lib --infodir=/usr/share/info --mandir=/usr/share/man --bindir=/usr/bin --libexecdir=/usr/lib --libdir=/usr/lib --enable-shared --with-system-zlib --enable-long-long --enable-nls --without-included-gettext --disable-libssp --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=avr [Thread](#) model: single gcc version 4.8.2 (GCC)

12.3 Profiling Results

- [Semaphore](#) Initialization: 40 cycles (averaged over 167 iterations)
- [Semaphore](#) Post (uncontested): 111 cycles (averaged over 167 iterations)
- [Semaphore](#) Pend (uncontested): 78 cycles (averaged over 167 iterations)
- [Semaphore](#) Flyback Time (Contested Pend): 1663 cycles (averaged over 167 iterations)
- [Mutex](#) Init: 223 cycles (averaged over 168 iterations)
- [Mutex](#) Claim: 239 cycles (averaged over 167 iterations)
- [Mutex](#) Release: 143 cycles (averaged over 167 iterations)
- [Thread](#) Initialize: 8391 cycles (averaged over 167 iterations)
- [Thread](#) Start: 839 cycles (averaged over 167 iterations)
- Context Switch: 183 cycles (averaged over 167 iterations)
- [Thread](#) Schedule: 111 cycles (averaged over 167 iterations)

Chapter 13

Code Size Profiling

The following report details the size of each module compiled into the kernel.

The size of each component is dependent on the flags specified in [mark3cfg.h](#) at compile time. Note that these sizes represent the maximum size of each module before dead code elimination and any additional link-time optimization, and represent the maximum possible size that any module can take.

The results below are for profiling on Atmel AVR atmega328p-based targets using gcc. Results are not necessarily indicative of relative or absolute performance on other platforms or toolchains.

13.1 Information

Subversion Repository Information:

- Repository Root: `svn+ssh://m0slevin.code.sf.net/p/mark3/source`
- Revision: 291
- URL: `svn+ssh://m0slevin.code.sf.net/p/mark3/source/trunk/embedded` Relative URL: `^/trunk/embedded`

Date Profiled: Mon Jan 11 21:42:32 EST 2016

13.2 Compiler Version

avr-gcc (GCC) 4.8.2 Copyright (C) 2013 Free Software Foundation, Inc. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

13.3 Profiling Results

Mark3 Module Size Report:

- Allocate-once Heap..... : 0 Bytes
- Synchronization Objects - Base Class..... : 136 Bytes
- Device [Driver](#) Framework (including /dev/null)... : 226 Bytes
- Synchronization Object - Event Flag..... : 770 Bytes
- Fundamental [Kernel](#) Linked-List Classes..... : 536 Bytes

- Message-based IPC..... : 426 Bytes
- [Mutex](#) (Synchronization Object)..... : 698 Bytes
- Notification Blocking Object..... : 538 Bytes
- Performance-profiling timers..... : 546 Bytes
- Round-Robin Scheduling Support..... : 264 Bytes
- [Thread](#) Scheduling..... : 452 Bytes
- [Semaphore](#) (Synchronization Object)..... : 540 Bytes
- [Mailbox](#) IPC Support..... : 966 Bytes
- [Thread](#) Implementation..... : 1611 Bytes
- Fundamental [Kernel](#) Thread-list Data Structures.. : 308 Bytes
- Mark3 [Kernel](#) Base Class..... : 110 Bytes
- Software [Timer Kernel](#) Object..... : 378 Bytes
- Software [Timer](#) Management..... : 645 Bytes
- Runtime [Kernel](#) Trace Implementation..... : 0 Bytes
- Atmel AVR - [Kernel](#) Aware Simulation Support..... : 250 Bytes
- Atmel AVR - Basic Threading Support..... : 598 Bytes
- Atmel AVR - [Kernel](#) Interrupt Implemenation..... : 56 Bytes
- Atmel AVR - [Kernel Timer](#) Implementation..... : 322 Bytes
- Atmel AVR - Profiling [Timer](#) Implementation..... : 256 Bytes

Mark3 [Kernel](#) Size Summary:

- [Kernel](#) : 3153 Bytes
- Synchronization Objects : 2434 Bytes
- Port : 4882 Bytes
- Features : 2059 Bytes
- Total Size : 12528 Bytes

Chapter 14

Hierarchical Index

14.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BlockingObject	81
EventFlag	95
Mutex	127
Notify	130
Semaphore	141
DriverList	92
FakeThread_t	99
GlobalMessagePool	100
Kernel	101
KernelAware	104
KernelAwareData_t	107
KernelSWI	108
KernelTimer	109
LinkList	113
CircularLinkList	82
ThreadList	153
DoubleLinkList	87
TimerList	162
LinkListNode	114
Driver	88
DevNull	84
Message	122
Thread	144
Timer	157
Mailbox	116
MessageQueue	125
Profiler	132
ProfileTimer	134
Quantum	136
Scheduler	138
ThreadPort	156
TimerScheduler	164

Chapter 15

Class Index

15.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BlockingObject	Class implementing thread-blocking primitives	81
CircularLinkedList	Circular-linked-list data type, inherited from the base LinkedList type	82
DevNull	This class implements the "default" driver (/dev/null)	84
DoubleLinkedList	Doubly-linked-list data type, inherited from the base LinkedList type	87
Driver	Base device-driver class used in hardware abstraction	88
DriverList	List of Driver objects used to keep track of all device drivers in the system	92
EventFlag	Blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system	95
FakeThread_t	If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system	99
GlobalMessagePool	Implements a list of message objects shared between all threads	100
Kernel	Class that encapsulates all of the kernel startup functions	101
KernelAware	The KernelAware class	104
KernelAwareData_t	This structure is used to communicate between the kernel and a kernel-aware host	107
KernelSWI	Class providing the software-interrupt required for context-switching in the kernel	108
KernelTimer	Hardware timer interface, used by all scheduling/timer subsystems	109
LinkedList	Abstract-data-type from which all other linked-lists are derived	113
LinkedListNode	Basic linked-list node data structure	114
Mailbox	Implements an IPC mechanism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user	116

Message	Class to provide message-based IPC services in the kernel	122
MessageQueue	List of messages, used as the channel for sending and receiving messages between threads	125
Mutex	Mutual-exclusion locks, based on BlockingObject	127
Notify	Blocking object type, that allows one or more threads to wait for an event to occur before resuming operation	130
Profiler	System profiling timer interface	132
ProfileTimer	Profiling timer	134
Quantum	Static-class used to implement Thread quantum functionality, which is a key part of round-robin scheduling	136
Scheduler	Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping	138
Semaphore	Binary & Counting semaphores, based on BlockingObject base class	141
Thread	Object providing fundamental multitasking support in the kernel	144
ThreadList	This class is used for building thread-management facilities, such as schedulers, and blocking objects	153
ThreadPort	Class defining the architecture specific functions required by the kernel	156
Timer	Timer - an event-driven execution context based on a specified time interval	157
TimerList	TimerList class - a doubly-linked-list of timer objects	162
TimerScheduler	"Static" Class used to interface a global TimerList with the rest of the kernel	164

Chapter 16

File Index

16.1 File List

Here is a list of all documented files with brief descriptions:

/home/mo/mark3-source/embedded/kernel/ atomic.cpp	
Basic Atomic Operations	167
/home/mo/mark3-source/embedded/kernel/ autoalloc.cpp	
Automatic memory allocation for kernel objects	169
/home/mo/mark3-source/embedded/kernel/ blocking.cpp	
Implementation of base class for blocking objects	170
/home/mo/mark3-source/embedded/kernel/ driver.cpp	
Device driver/hardware abstraction layer	185
/home/mo/mark3-source/embedded/kernel/ eventflag.cpp	
Event Flag Blocking Object/IPC-Object implementation	187
/home/mo/mark3-source/embedded/kernel/ kernel.cpp	
Kernel initialization and startup code	192
/home/mo/mark3-source/embedded/kernel/ kernelaware.cpp	
Kernel aware simulation support	194
/home/mo/mark3-source/embedded/kernel/ ksemaphore.cpp	
Semaphore Blocking-Object Implementation	197
/home/mo/mark3-source/embedded/kernel/ ll.cpp	
Core Linked-List implementation, from which all kernel objects are derived	200
/home/mo/mark3-source/embedded/kernel/ mailbox.cpp	
Mailbox + Envelope IPC mechanism	203
/home/mo/mark3-source/embedded/kernel/ message.cpp	
Inter-thread communications via message passing	207
/home/mo/mark3-source/embedded/kernel/ mutex.cpp	
Mutual-exclusion object	209
/home/mo/mark3-source/embedded/kernel/ notify.cpp	
Lightweight thread notification - blocking object	213
/home/mo/mark3-source/embedded/kernel/ profile.cpp	
Code profiling utilities	215
/home/mo/mark3-source/embedded/kernel/ quantum.cpp	
Thread Quantum Implementation for Round-Robin Scheduling	266
/home/mo/mark3-source/embedded/kernel/ scheduler.cpp	
Strict-Priority + Round-Robin thread scheduler implementation	268
/home/mo/mark3-source/embedded/kernel/ thread.cpp	
Platform-Independent thread class Definition	271
/home/mo/mark3-source/embedded/kernel/ threadlist.cpp	
Thread linked-list definitions	276
/home/mo/mark3-source/embedded/kernel/ timer.cpp	
Timer implementations	278

/home/mo/mark3-source/embedded/kernel/timerlist.cpp	
Implements timer list processing algorithms, responsible for all timer tick and expiry logic	280
/home/mo/mark3-source/embedded/kernel/tracebuffer.cpp	
Kernel trace buffer class definition	283
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp	
ATMega328p Profiling timer implementation	171
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp	
Kernel Software interrupt implementation for ATMega328p	173
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp	
Kernel Timer Implementation for ATMega328p	174
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp	
ATMega328p Multithreading	182
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h	
Profiling timer hardware interface	176
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h	
Kernel Software interrupt declarations	177
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h	
Kernel Timer Class declaration	178
/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h	
ATMega328p Multithreading support	179
/home/mo/mark3-source/embedded/kernel/public/atomic.h	
Basic Atomic Operations	217
/home/mo/mark3-source/embedded/kernel/public/autoalloc.h	
Automatic memory allocation for kernel objects	218
/home/mo/mark3-source/embedded/kernel/public/blocking.h	
Blocking object base class declarations	219
/home/mo/mark3-source/embedded/kernel/public/buffalogger.h	
Super-efficient, super-secure logging routines	220
/home/mo/mark3-source/embedded/kernel/public/dbg_file_list.h	??
/home/mo/mark3-source/embedded/kernel/public/driver.h	
Driver abstraction framework	221
/home/mo/mark3-source/embedded/kernel/public/eventflag.h	
Event Flag Blocking Object/IPC-Object definition	223
/home/mo/mark3-source/embedded/kernel/public/kernel.h	
Kernel initialization and startup class	224
/home/mo/mark3-source/embedded/kernel/public/kernelaware.h	
Kernel aware simulation support	225
/home/mo/mark3-source/embedded/kernel/public/kerneldebug.h	
Macros and functions used for assertions, kernel traces, etc	228
/home/mo/mark3-source/embedded/kernel/public/kerneltypes.h	
Basic data type primitives used throughout the OS	231
/home/mo/mark3-source/embedded/kernel/public/ksemaphore.h	
Semaphore Blocking Object class declarations	233
/home/mo/mark3-source/embedded/kernel/public/ll.h	
Core linked-list declarations, used by all kernel list types	234
/home/mo/mark3-source/embedded/kernel/public/mailbox.h	
Mailbox + Envelope IPC Mechanism	236
/home/mo/mark3-source/embedded/kernel/public/manual.h	
/brief Ascii-format documentation, used by doxygen to create various printable and viewable forms	238
/home/mo/mark3-source/embedded/kernel/public/mark3.h	
Single include file given to users of the Mark3 Kernel API	239
/home/mo/mark3-source/embedded/kernel/public/mark3cfg.h	
Mark3 Kernel Configuration	240
/home/mo/mark3-source/embedded/kernel/public/message.h	
Inter-thread communication via message-passing	246
/home/mo/mark3-source/embedded/kernel/public/mutex.h	
Mutual exclusion class declaration	248

/home/mo/mark3-source/embedded/kernel/public/notify.h	
Lightweight thread notification - blocking object	250
/home/mo/mark3-source/embedded/kernel/public/paniccodes.h	
Defines the reason codes thrown when a kernel panic occurs	251
/home/mo/mark3-source/embedded/kernel/public/profile.h	
High-precision profiling timers	251
/home/mo/mark3-source/embedded/kernel/public/profiling_results.h	??
/home/mo/mark3-source/embedded/kernel/public/quantum.h	
Thread Quantum declarations for Round-Robin Scheduling	253
/home/mo/mark3-source/embedded/kernel/public/scheduler.h	
Thread scheduler function declarations	254
/home/mo/mark3-source/embedded/kernel/public/sizeprofile.h	??
/home/mo/mark3-source/embedded/kernel/public/thread.h	
Platform independent thread class declarations	256
/home/mo/mark3-source/embedded/kernel/public/threadlist.h	
Thread linked-list declarations	259
/home/mo/mark3-source/embedded/kernel/public/timer.h	
Timer object declarations	260
/home/mo/mark3-source/embedded/kernel/public/timerlist.h	
Timer list declarations	263
/home/mo/mark3-source/embedded/kernel/public/timerscheduler.h	
Timer scheduler declarations	264
/home/mo/mark3-source/embedded/kernel/public/tracebuffer.h	
Kernel trace buffer class declaration	265

Chapter 17

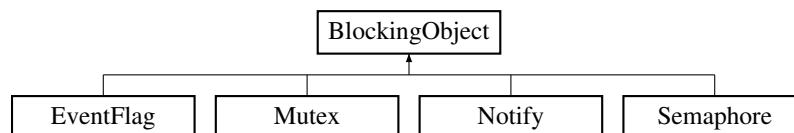
Class Documentation

17.1 BlockingObject Class Reference

Class implementing thread-blocking primitives.

```
#include <blocking.h>
```

Inheritance diagram for BlockingObject:



Protected Member Functions

- void **Block** (Thread *pclThread_)
Block.
- void **BlockPriority** (Thread *pclThread_)
BlockPriority.
- void **UnBlock** (Thread *pclThread_)
UnBlock.

Protected Attributes

- ThreadList m_clBlockList
ThreadList which is used to hold the list of threads blocked on a given object.

17.1.1 Detailed Description

Class implementing thread-blocking primitives.

used for implementing things like semaphores, mutexes, message queues, or anything else that could cause a thread to suspend execution on some external stimulus.

Definition at line 65 of file [blocking.h](#).

17.1.2 Member Function Documentation

17.1.2.1 void BlockingObject::Block (Thread * *pclThread_*) [protected]

Block.

Blocks a thread on this object. This is the fundamental operation performed by any sort of blocking operation in the operating system. All semaphores/mutexes/sleeping/messaging/etc ends up going through the blocking code at some point as part of the code that manages a transition from an "active" or "waiting" thread to a "blocked" thread.

The steps involved in blocking a thread (which are performed in the function itself) are as follows;

1) Remove the specified thread from the current owner's list (which is likely one of the scheduler's thread lists) 2) Add the thread to this object's thread list 3) Setting the thread's "current thread-list" point to reference this object's threadlist.

Parameters

<i>pclThread_</i>	Pointer to the thread object that will be blocked.
-------------------	--

Definition at line 41 of file [blocking.cpp](#).

17.1.2.2 void BlockingObject::BlockPriority (Thread * *pclThread_*) [protected]

BlockPriority.

Same as [Block\(\)](#), but ensures that threads are added to the block-list in priority-order, which optimizes the unblock procedure.

Parameters

<i>pclThread_</i>	Pointer to the Thread to Block.
-------------------	---

Definition at line 57 of file [blocking.cpp](#).

17.1.2.3 void BlockingObject::UnBlock (Thread * *pclThread_*) [protected]

UnBlock.

Unblock a thread that is already blocked on this object, returning it to the "ready" state by performing the following steps:

Parameters

<i>pclThread_</i>	Pointer to the thread to unblock.
-------------------	-----------------------------------

1) Removing the thread from this object's threadlist 2) Restoring the thread to its "original" owner's list

Definition at line 73 of file [blocking.cpp](#).

The documentation for this class was generated from the following files:

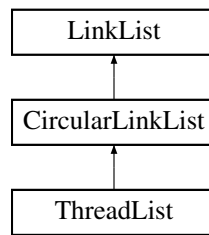
- [/home/mo/mark3-source/embedded/kernel/public/blocking.h](#)
- [/home/mo/mark3-source/embedded/kernel/blocking.cpp](#)

17.2 CircularLinkedList Class Reference

Circular-linked-list data type, inherited from the base [LinkList](#) type.

```
#include <ll.h>
```

Inheritance diagram for CircularLinkedList:



Public Member Functions

- virtual void [Add](#) ([LinkedListNode](#) *node_)
Add the linked list node to this linked list.
- virtual void [Remove](#) ([LinkedListNode](#) *node_)
Remove.
- void [PivotForward](#) ()
PivotForward.
- void [PivotBackward](#) ()
PivotBackward.
- void [InsertNodeBefore](#) ([LinkedListNode](#) *node_, [LinkedListNode](#) *insert_)
InsertNodeBefore.

Additional Inherited Members

17.2.1 Detailed Description

Circular-linked-list data type, inherited from the base [LinkedList](#) type.

Definition at line 201 of file [ll.h](#).

17.2.2 Member Function Documentation

17.2.2.1 void [CircularLinkedList::Add](#) ([LinkedListNode](#) * node_) [virtual]

Add the linked list node to this linked list.

Parameters

<i>node_</i>	Pointer to the node to add
--------------	----------------------------

Implements [LinkedList](#).

Reimplemented in [ThreadList](#).

Definition at line 108 of file [ll.cpp](#).

17.2.2.2 void [CircularLinkedList::InsertNodeBefore](#) ([LinkedListNode](#) * node_, [LinkedListNode](#) * insert_)

[InsertNodeBefore](#).

Insert a linked-list node into the list before the specified insertion point.

Parameters

<i>node_</i>	Node to insert into the list
<i>insert_</i>	Insert point.

Definition at line 191 of file [ll.cpp](#).

17.2.2.3 void CircularLinkedList::PivotBackward ()

PivotBackward.

Pivot the head of the circularly linked list backward (Head = Head->prev, Tail = Tail->prev)

Definition at line 181 of file [ll.cpp](#).

17.2.2.4 void CircularLinkedList::PivotForward ()

PivotForward.

Pivot the head of the circularly linked list forward (Head = Head->next, Tail = Tail->next)

Definition at line 171 of file [ll.cpp](#).

17.2.2.5 void CircularLinkedList::Remove (LinkListNode * node_) [virtual]

Remove.

Add the linked list node to this linked list

Parameters

<i>node_</i>	Pointer to the node to remove
--------------	-------------------------------

Implements [LinkedList](#).

Reimplemented in [ThreadList](#).

Definition at line 133 of file [ll.cpp](#).

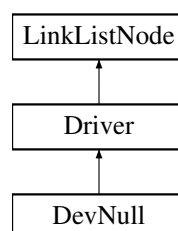
The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/ll.h](#)
- [/home/mo/mark3-source/embedded/kernel/ll.cpp](#)

17.3 DevNull Class Reference

This class implements the "default" driver (/dev/null)

Inheritance diagram for DevNull:



Public Member Functions

- virtual void [Init](#) ()
Init.
- virtual uint8_t [Open](#) ()
Open.
- virtual uint8_t [Close](#) ()
Close.
- virtual uint16_t [Read](#) (uint16_t u16Bytes_, uint8_t *pu8Data_)
Read.
- virtual uint16_t [Write](#) (uint16_t u16Bytes_, uint8_t *pu8Data_)
Write.
- virtual uint16_t [Control](#) (uint16_t u16Event_, void *pvDataIn_, uint16_t u16SizeIn_, void *pvDataOut_
, uint16_t u16SizeOut_)
Control.

Additional Inherited Members

17.3.1 Detailed Description

This class implements the "default" driver (/dev/null)

Definition at line 46 of file [driver.cpp](#).

17.3.2 Member Function Documentation

17.3.2.1 virtual uint8_t DevNull::Close () [inline], [virtual]

Close.

Close a previously-opened device driver.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements [Driver](#).

Definition at line 51 of file [driver.cpp](#).

17.3.2.2 virtual uint16_t DevNull::Control (uint16_t u16Event_, void * pvDataIn_, uint16_t u16SizeIn_, void * pvDataOut_, uint16_t u16SizeOut_) [inline], [virtual]

Control.

This is the main entry-point for device-specific io and control operations. This is used for implementing all "side-channel" communications with a device, and any device-specific IO operations that do not conform to the typical POSIX read/write paradigm. use of this funciton is analogous to the non-POSIX (yet still common) devctl() or ioctl().

Parameters

<i>u16Event_</i>	Code defining the io event (driver-specific)
------------------	--

<i>pvDataIn_</i>	Pointer to the input data
<i>u16SizeIn_</i>	Size of the input data (in bytes)
<i>pvDataOut_</i>	Pointer to the output data
<i>u16SizeOut_</i>	Size of the output data (in bytes)

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements [Driver](#).

Definition at line 59 of file [driver.cpp](#).

17.3.2.3 `virtual void DevNull::Init () [inline],[virtual]`

Init.

Initialize a driver, must be called prior to use

Implements [Driver](#).

Definition at line 49 of file [driver.cpp](#).

17.3.2.4 `virtual uint8_t DevNull::Open () [inline],[virtual]`

Open.

Open a device driver prior to use.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements [Driver](#).

Definition at line 50 of file [driver.cpp](#).

17.3.2.5 `virtual uint16_t DevNull::Read (uint16_t u16Bytes_, uint8_t* pu8Data_) [inline],[virtual]`

Read.

Read a specified number of bytes from the device into a specific buffer. Depending on the driver-specific implementation, this may be a number less than the requested number of bytes read, indicating that there was less input than desired, or that as a result of buffering, the data may not be available.

Parameters

<i>u16Bytes_</i>	Number of bytes to read (<= size of the buffer)
<i>pu8Data_</i>	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implements [Driver](#).

Definition at line 53 of file [driver.cpp](#).

17.3.2.6 `virtual uint16_t DevNull::Write (uint16_t u16Bytes_, uint8_t * pu8Data_) [inline], [virtual]`

Write.

Write a payload of data of a given length to the device. Depending on the implementation of the driver, the amount of data written to the device may be less than the requested number of bytes. A result less than the requested size may indicate that the device buffer is full, indicating that the user must retry the write at a later point with the remaining data.

Parameters

<code>u16Bytes_</code>	Number of bytes to write (<= size of the buffer)
<code>pu8Data_</code>	Pointer to a data buffer containing the data to write

Returns

Number of bytes actually written

Implements [Driver](#).

Definition at line 56 of file [driver.cpp](#).

The documentation for this class was generated from the following file:

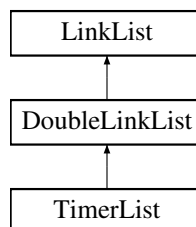
- [/home/mo/mark3-source/embedded/kernel/driver.cpp](#)

17.4 DoubleLinkedList Class Reference

Doubly-linked-list data type, inherited from the base [LinkedList](#) type.

```
#include <ll.h>
```

Inheritance diagram for DoubleLinkedList:



Public Member Functions

- [DoubleLinkedList](#) ()
DoubleLinkedList.
- virtual void [Add](#) ([LinkedListNode](#) *node_)
Add.
- virtual void [Remove](#) ([LinkedListNode](#) *node_)
Remove.

Additional Inherited Members

17.4.1 Detailed Description

Doubly-linked-list data type, inherited from the base [LinkedList](#) type.

Definition at line 168 of file [ll.h](#).

17.4.2 Constructor & Destructor Documentation

17.4.2.1 DoubleLinkedList::DoubleLinkedList () [inline]

[DoubleLinkedList](#).

Default constructor - initializes the head/tail nodes to NULL

Definition at line 176 of file [ll.h](#).

17.4.3 Member Function Documentation

17.4.3.1 void DoubleLinkedList::Add (LinkListNode * node_) [virtual]

Add.

Add the linked list node to this linked list

Parameters

<i>node_</i>	Pointer to the node to add
--------------	----------------------------

Implements [LinkedList](#).

Definition at line 47 of file [ll.cpp](#).

17.4.3.2 void DoubleLinkedList::Remove (LinkListNode * node_) [virtual]

Remove.

Add the linked list node to this linked list

Parameters

<i>node_</i>	Pointer to the node to remove
--------------	-------------------------------

Implements [LinkedList](#).

Definition at line 71 of file [ll.cpp](#).

The documentation for this class was generated from the following files:

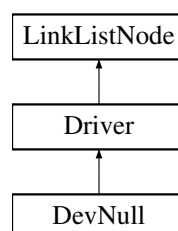
- [/home/mo/mark3-source/embedded/kernel/public/ll.h](#)
- [/home/mo/mark3-source/embedded/kernel/ll.cpp](#)

17.5 Driver Class Reference

Base device-driver class used in hardware abstraction.

```
#include <driver.h>
```

Inheritance diagram for Driver:



Public Member Functions

- virtual void [Init](#) ()=0
Init.
- virtual uint8_t [Open](#) ()=0
Open.
- virtual uint8_t [Close](#) ()=0
Close.
- virtual uint16_t [Read](#) (uint16_t u16Bytes_, uint8_t *pu8Data_)=0
Read.
- virtual uint16_t [Write](#) (uint16_t u16Bytes_, uint8_t *pu8Data_)=0
Write.
- virtual uint16_t [Control](#) (uint16_t u16Event_, void *pvDataIn_, uint16_t u16SizeIn_, void *pvDataOut_
, uint16_t u16SizeOut_)=0
Control.
- void [SetName](#) (const char *pcName_)
SetName.
- const char * [GetPath](#) ()
GetPath.

Private Attributes

- const char * [m_pcPath](#)
string pointer that holds the driver path (name)

Additional Inherited Members

17.5.1 Detailed Description

Base device-driver class used in hardware abstraction.

All other device drivers inherit from this class

Definition at line 121 of file [driver.h](#).

17.5.2 Member Function Documentation

17.5.2.1 virtual uint8_t Driver::Close () [pure virtual]

Close.

Close a previously-opened device driver.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in [DevNull](#).

17.5.2.2 `virtual uint16_t Driver::Control (uint16_t u16Event_, void * pvDataIn_, uint16_t u16SizeIn_, void * pvDataOut_,
uint16_t u16SizeOut_) [pure virtual]`

Control.

This is the main entry-point for device-specific io and control operations. This is used for implementing all "side-channel" communications with a device, and any device-specific IO operations that do not conform to the typical POSIX read/write paradigm. use of this funciton is analagous to the non-POSIX (yet still common) `devctl()` or `ioctl()`.

Parameters

<i>u16Event_</i>	Code defining the io event (driver-specific)
<i>pvDataIn_</i>	Pointer to the input data
<i>u16SizeIn_</i>	Size of the input data (in bytes)
<i>pvDataOut_</i>	Pointer to the output data
<i>u16SizeOut_</i>	Size of the output data (in bytes)

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in [DevNull](#).

17.5.2.3 `const char* Driver::GetPath () [inline]`

GetPath.

Returns a string containing the device path.

Returns

pcName_ Return the string constant representing the device path

Definition at line 225 of file [driver.h](#).

17.5.2.4 `virtual void Driver::Init () [pure virtual]`

Init.

Initialize a driver, must be called prior to use

Implemented in [DevNull](#).

17.5.2.5 `virtual uint8_t Driver::Open () [pure virtual]`

Open.

Open a device driver prior to use.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in [DevNull](#).

17.5.2.6 `virtual uint16_t Driver::Read (uint16_t u16Bytes_, uint8_t* pu8Data_) [pure virtual]`

Read.

Read a specified number of bytes from the device into a specific buffer. Depending on the driver-specific implementation, this may be a number less than the requested number of bytes read, indicating that there there was less input than desired, or that as a result of buffering, the data may not be available.

Parameters

<i>u16Bytes_</i>	Number of bytes to read (<= size of the buffer)
<i>pu8Data_</i>	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implemented in [DevNull](#).

17.5.2.7 `void Driver::SetName (const char * pcName_) [inline]`

SetName.

Set the path for the driver. Name must be set prior to access (since driver access is name-based).

Parameters

<i>pcName_</i>	String constant containing the device path
----------------	--

Definition at line 216 of file [driver.h](#).

17.5.2.8 `virtual uint16_t Driver::Write (uint16_t u16Bytes_, uint8_t * pu8Data_) [pure virtual]`

Write.

Write a payload of data of a given length to the device. Depending on the implementation of the driver, the amount of data written to the device may be less than the requested number of bytes. A result less than the requested size may indicate that the device buffer is full, indicating that the user must retry the write at a later point with the remaining data.

Parameters

<i>u16Bytes_</i>	Number of bytes to write (<= size of the buffer)
<i>pu8Data_</i>	Pointer to a data buffer containing the data to write

Returns

Number of bytes actually written

Implemented in [DevNull](#).

The documentation for this class was generated from the following file:

- [/home/mo/mark3-source/embedded/kernel/public/driver.h](#)

17.6 DriverList Class Reference

List of [Driver](#) objects used to keep track of all device drivers in the system.

```
#include <driver.h>
```

Static Public Member Functions

- static void [Init](#) ()
Init.
- static void [Add](#) ([Driver](#) **pclDriver_*)

Add.

- static void [Remove](#) ([Driver](#) *pclDriver_)

Remove.

- static [Driver](#) * [FindByPath](#) (const char *m_pcPath)

FindByPath.

Static Private Attributes

- static [DoubleLinkedList](#) m_clDriverList

LinkedList object used to implementing the driver object management.

17.6.1 Detailed Description

List of [Driver](#) objects used to keep track of all device drivers in the system.

By default, the list contains a single entity, "/dev/null".

Definition at line 238 of file [driver.h](#).

17.6.2 Member Function Documentation

17.6.2.1 static void [DriverList::Add](#) ([Driver](#) * [pclDriver_](#)) [inline], [static]

Add.

Add a [Driver](#) object to the managed global driver-list.

Parameters

pclDriver_	pointer to the driver object to add to the global driver list.
----------------------------	--

Examples:

[buffalogger/main.cpp](#).

Definition at line 258 of file [driver.h](#).

17.6.2.2 [Driver](#) * [DriverList::FindByPath](#) (const char * [m_pcPath](#)) [static]

FindByPath.

Look-up a driver in the global driver-list based on its path. In the event that the driver is not found in the list, a pointer to the default "/dev/null" object is returned. In this way, unimplemented drivers are automatically stubbed out.

Definition at line 113 of file [driver.cpp](#).

17.6.2.3 void [DriverList::Init](#) () [static]

Init.

Initialize the list of drivers. Must be called prior to using the device driver library.

Definition at line 104 of file [driver.cpp](#).

17.6.2.4 `static void DriverList::Remove (Driver * pclDriver_)` `[inline],[static]`

Remove.

Remove a driver from the global driver list.

Parameters

<code>pclDriver_</code>	Pointer to the driver object to remove from the global table
-------------------------	--

Definition at line 268 of file [driver.h](#).

The documentation for this class was generated from the following files:

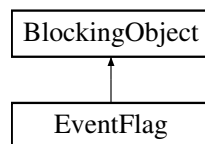
- [/home/mo/mark3-source/embedded/kernel/public/driver.h](#)
- [/home/mo/mark3-source/embedded/kernel/driver.cpp](#)

17.7 EventFlag Class Reference

The [EventFlag](#) class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

```
#include <eventflag.h>
```

Inheritance diagram for EventFlag:



Public Member Functions

- void [Init](#) ()
Init Initializes the [EventFlag](#) object prior to use.
- uint16_t [Wait](#) (uint16_t u16Mask_, [EventFlagOperation_t](#) eMode_)
Wait - Block a thread on the specific flags in this event flag group.
- uint16_t [Wait](#) (uint16_t u16Mask_, [EventFlagOperation_t](#) eMode_, uint32_t u32TimeMS_)
Wait - Block a thread on the specific flags in this event flag group.
- void [WakeMe](#) ([Thread](#) *pclOwner_)
WakeMe.
- void [Set](#) (uint16_t u16Mask_)
Set - Set additional flags in this object (logical OR).
- void [Clear](#) (uint16_t u16Mask_)
ClearFlags - Clear a specific set of flags within this object, specific by bitmask.
- uint16_t [GetMask](#) ()
GetMask Returns the state of the 16-bit bitmask within this object.

Private Member Functions

- uint16_t [Wait_i](#) (uint16_t u16Mask_, [EventFlagOperation_t](#) eMode_, uint32_t u32TimeMS_)
Wait_i.

Private Attributes

- uint16_t [m_u16SetMask](#)
Event flags currently set in this object.

Additional Inherited Members

17.7.1 Detailed Description

The [EventFlag](#) class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

Each [EventFlag](#) object contains a 16-bit bitmask, which is used to trigger events on associated threads. Threads wishing to block, waiting for a specific event to occur can wait on any pattern within this 16-bit bitmask to be set. Here, we provide the ability for a thread to block, waiting for ANY bits in a specified mask to be set, or for ALL bits within a specific mask to be set. Depending on how the object is configured, the bits that triggered the wakeup can be automatically cleared once a match has occurred.

Examples:

[lab7_events/main.cpp](#).

Definition at line 46 of file [eventflag.h](#).

17.7.2 Member Function Documentation

17.7.2.1 void EventFlag::Clear (uint16_t u16Mask_)

ClearFlags - Clear a specific set of flags within this object, specific by bitmask.

Parameters

u16Mask_	- Bitmask of flags to clear
--------------------------	-----------------------------

Examples:

[lab7_events/main.cpp](#).

Definition at line 306 of file [eventflag.cpp](#).

17.7.2.2 uint16_t EventFlag::GetMask ()

GetMask Returns the state of the 16-bit bitmask within this object.

Returns

The state of the 16-bit bitmask

Definition at line 315 of file [eventflag.cpp](#).

17.7.2.3 void EventFlag::Set (uint16_t u16Mask_)

Set - Set additional flags in this object (logical OR).

This API can potentially result in threads blocked on [Wait\(\)](#) to be unblocked.

Parameters

u16Mask_	- Bitmask of flags to set.
--------------------------	----------------------------

Examples:

[lab7_events/main.cpp](#).

Definition at line 187 of file [eventflag.cpp](#).

17.7.2.4 `uint16_t EventFlag::Wait (uint16_t u16Mask, EventFlagOperation_t eMode)`

Wait - Block a thread on the specific flags in this event flag group.

Parameters

<i>u16Mask_</i>	- 16-bit bitmask to block on
<i>eMode_</i>	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask <ul style="list-style-type: none"> • EVENT_FLAG_ALL: Thread will block on all of the bits in the mask

Returns

Bitmask condition that caused the thread to unblock, or 0 on error or timeout

Examples:

[lab7_events/main.cpp](#).

Definition at line 169 of file [eventflag.cpp](#).

17.7.2.5 `uint16_t EventFlag::Wait (uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t u32TimeMS_)`

Wait - Block a thread on the specific flags in this event flag group.

Parameters

<i>u16Mask_</i>	- 16-bit bitmask to block on
<i>eMode_</i>	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask <ul style="list-style-type: none"> • EVENT_FLAG_ALL: Thread will block on all of the bits in the mask
<i>u32TimeMS_</i>	- Time to block (in ms)

Returns

Bitmask condition that caused the thread to unblock, or 0 on error or timeout

Definition at line 180 of file [eventflag.cpp](#).

17.7.2.6 `uint16_t EventFlag::Wait_i (uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t u32TimeMS_)`
`[private]`

Wait_i.

Internal abstraction used to manage both timed and untimed wait operations

Parameters

<i>u16Mask_</i>	- 16-bit bitmask to block on
<i>eMode_</i>	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask <ul style="list-style-type: none"> • EVENT_FLAG_ALL: Thread will block on all of the bits in the mask
<i>u32TimeMS_</i>	- Time to block (in ms)

Returns

Bitmask condition that caused the thread to unblock, or 0 on error or timeout

! If the Yield operation causes a new thread to be chosen, there will ! Be a context switch at the above [CS_EXIT\(\)](#). The original calling ! thread will not return back until a matching SetFlags call is made ! or a timeout occurs.

Definition at line 76 of file [eventflag.cpp](#).

17.7.2.7 void EventFlag::WakeMe (Thread * *pclOwner_*)

WakeMe.

Wake the given thread, currently blocking on this object

Parameters

<i>pclOwner_</i>	Pointer to the owner thread to unblock.
------------------	---

Definition at line 68 of file [eventflag.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/eventflag.h](#)
- [/home/mo/mark3-source/embedded/kernel/eventflag.cpp](#)

17.8 FakeThread_t Struct Reference

If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

```
#include <thread.h>
```

Public Attributes

- [K_WORD * m_pwStackTop](#)
Pointer to the top of the thread's stack.
- [K_WORD * m_pwStack](#)
Pointer to the thread's stack.
- [uint8_t m_u8ThreadID](#)
Thread ID.
- [uint8_t m_u8Priority](#)
Default priority of the thread.
- [uint8_t m_u8CurPriority](#)
Current priority of the thread (priority inheritance)
- [ThreadState_t m_eState](#)
Enum indicating the thread's current state.

17.8.1 Detailed Description

If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

When cast to a [Thread](#), this data structure will still result in GetPriority() calls being valid, which is all that is needed to support the tick-based/tickless times – while saving a fairly decent chunk of RAM on a small micro.

Note that this struct must have the same memory layout as the [Thread](#) class up to the last item.

Definition at line 516 of file [thread.h](#).

The documentation for this struct was generated from the following file:

- [/home/mo/mark3-source/embedded/kernel/public/thread.h](#)

17.9 GlobalMessagePool Class Reference

Implements a list of message objects shared between all threads.

```
#include <message.h>
```

Static Public Member Functions

- static void [Init](#) ()
Init.
- static void [Push](#) ([Message](#) *pclMessage_)
Push.
- static [Message](#) * [Pop](#) ()
Pop.

Static Private Attributes

- static [Message](#) [m_aclMessagePool](#) [[GLOBAL_MESSAGE_POOL_SIZE](#)]
Array of message objects that make up the message pool.
- static [DoubleLinkedList](#) [m_clList](#)
Linked list used to manage the [Message](#) objects.

17.9.1 Detailed Description

Implements a list of message objects shared between all threads.

Definition at line 157 of file [message.h](#).

17.9.2 Member Function Documentation

17.9.2.1 void GlobalMessagePool::Init (void) [static]

[Init.](#)

Initialize the message queue prior to use

Definition at line 50 of file [message.cpp](#).

17.9.2.2 Message * GlobalMessagePool::Pop () [static]

[Pop.](#)

Pop a message from the global queue, returning it to the user to be popu32ated before sending by a transmitter.

Returns

Pointer to a [Message](#) object

Examples:

[lab8_messages/main.cpp](#).

Definition at line 74 of file [message.cpp](#).

17.9.2.3 void GlobalMessagePool::Push (Message * *pcIMessage_*) [static]

Push.

Return a previously-claimed message object back to the global queue. used once the message has been processed by a receiver.

Parameters

<i>pcIMessage_</i>	Pointer to the Message object to return back to the global queue
--------------------	--

Examples:

[lab8_messages/main.cpp](#).

Definition at line 62 of file [message.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/message.h](#)
- [/home/mo/mark3-source/embedded/kernel/message.cpp](#)

17.10 Kernel Class Reference

Class that encapsulates all of the kernel startup functions.

```
#include <kernel.h>
```

Static Public Member Functions

- static void [Init](#) (void)
Kernel Initialization Function, call before any other OS function.
- static void [Start](#) (void)
Start the operating system kernel - the current execution context is cancelled, all kernel services are started, and the processor resumes execution at the entrypoint for the highest-priority thread.
- static bool [IsStarted](#) ()
IsStarted.
- static void [SetPanic](#) (panic_func_t pfPanic_)
SetPanic Set a function to be called when a kernel panic occurs, giving the user to determine the behavior when a catastrophic failure is observed.
- static bool [IsPanic](#) ()
IsPanic Returns whether or not the kernel is in a panic state.
- static void [Panic](#) (uint16_t u16Cause_)
Panic Cause the kernel to enter its panic state.
- static void [SetIdleFunc](#) (idle_func_t pfIdle_)
SetIdleFunc Set the function to be called when no active threads are available to be scheduled by the scheduler.
- static void [IdleFunc](#) (void)
IdleFunc Call the low-priority idle function when no active threads are available to be scheduled.
- static Thread * [GetIdleThread](#) (void)
GetIdleThread Return a pointer to the Kernel's idle thread object to the user.

Static Private Attributes

- static bool [m_blsStarted](#)
true if kernel is running, false otherwise
- static bool [m_blsPanic](#)
true if kernel is in panic state, false otherwise
- static [panic_func_t](#) [m_pfPanic](#)
set panic function
- static [idle_func_t](#) [m_pfIdle](#)
set idle function
- static [FakeThread_t](#) [m_clIdle](#)
Idle thread object (note: not a real thread)

17.10.1 Detailed Description

Class that encapsulates all of the kernel startup functions.

Definition at line 48 of file [kernel.h](#).

17.10.2 Member Function Documentation

17.10.2.1 static Thread* Kernel::GetIdleThread(void) [inline],[static]

GetIdleThread Return a pointer to the [Kernel](#)'s idle thread object to the user.

Note that the [Thread](#) object involved is to be used for comparisons only – the thread itself is "virtual", and doesn't represent a unique execution context with its own stack.

Returns

Pointer to the [Kernel](#)'s idle thread object

Definition at line 125 of file [kernel.h](#).

17.10.2.2 void Kernel::Init(void) [static]

[Kernel](#) Initialization Function, call before any other OS function.

Initializes all global resources used by the operating system. This must be called before any other kernel function is invoked.

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 57 of file [kernel.cpp](#).

17.10.2.3 static bool Kernel::IsPanic() [inline],[static]

IsPanic Returns whether or not the kernel is in a panic state.

Returns

Whether or not the kernel is in a panic state

Definition at line 96 of file [kernel.h](#).

17.10.2.4 `static bool Kernel::IsStarted () [inline],[static]`

IsStarted.

Returns

Whether or not the kernel has started - true = running, false = not started

Definition at line 81 of file [kernel.h](#).

17.10.2.5 `void Kernel::Panic (uint16_t u16Cause_) [static]`

Panic Cause the kernel to enter its panic state.

Parameters

<i>u16Cause_</i>	Reason for the kernel panic
------------------	-----------------------------

Definition at line 102 of file [kernel.cpp](#).

17.10.2.6 `static void Kernel::SetIdleFunc (idle_func_t pfIdle_) [inline],[static]`

SetIdleFunc Set the function to be called when no active threads are available to be scheduled by the scheduler.

Parameters

<i>pfIdle_</i>	Pointer to the idle function
----------------	------------------------------

Examples:

[lab2_idle_function/main.cpp](#).

Definition at line 110 of file [kernel.h](#).

17.10.2.7 `static void Kernel::SetPanic (panic_func_t pfPanic_) [inline],[static]`

SetPanic Set a function to be called when a kernel panic occurs, giving the user to determine the behavior when a catastrophic failure is observed.

Parameters

<i>pfPanic_</i>	Panic function pointer
-----------------	------------------------

Definition at line 90 of file [kernel.h](#).

17.10.2.8 `void Kernel::Start (void) [static]`

Start the operating system kernel - the current execution context is cancelled, all kernel services are started, and the processor resumes execution at the entrypoint for the highest-priority thread.

You must have at least one thread added to the kernel before calling this function, otherwise the behavior is undefined. The exception to this is if the system is configured to use the threadless idle hook, in which case the kernel is allowed to run without any ready threads.

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 93 of file [kernel.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/kernel.h](#)
- [/home/mo/mark3-source/embedded/kernel/kernel.cpp](#)

17.11 KernelAware Class Reference

The [KernelAware](#) class.

```
#include <kernelaware.h>
```

Static Public Member Functions

- static void [ProfileInit](#) (const char *szStr_)
ProfileInit.
- static void [ProfileStart](#) (void)
ProfileStart.
- static void [ProfileStop](#) (void)
ProfileStop.
- static void [ProfileReport](#) (void)
ProfileReport.
- static void [ExitSimulator](#) (void)
ExitSimulator.
- static void [Print](#) (const char *szStr_)
Print.
- static void [Trace](#) (uint16_t u16File_, uint16_t u16Line_)
Trace.
- static void [Trace](#) (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_)
Trace.
- static void [Trace](#) (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_)
Trace.
- static bool [IsSimulatorAware](#) (void)
IsSimulatorAware.

Static Private Member Functions

- static void [Trace_i](#) (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_, [KernelAwareCommand_t](#) eCmd_)
Trace_i.

17.11.1 Detailed Description

The [KernelAware](#) class.

This class contains functions that are used to trigger kernel-aware functionality within a supported simulation environment (i.e. fIAVR).

These static methods operate on a singleton set of global variables, which are monitored for changes from within the simulator. The simulator hooks into these variables by looking for the correctly-named symbols in an elf-formatted binary being run and registering callbacks that are called whenever the variables are changed. On each change of the command variable, the kernel-aware data is analyzed and interpreted appropriately.

If these methods are run in an unsupported simulator or on actual hardware the commands generally have no effect (except for the exit-on-reset command, which will result in a jump-to-0 reset).

Definition at line 65 of file [kernelaware.h](#).

17.11.2 Member Function Documentation

17.11.2.1 void KernelAware::ExitSimulator (void) [static]

ExitSimulator.

Instruct the kernel-aware simulator to terminate (destroying the virtual CPU).

Definition at line 114 of file [kernelaware.cpp](#).

17.11.2.2 bool KernelAware::IsSimulatorAware (void) [static]

IsSimulatorAware.

use this function to determine whether or not the code is running on a simulator that is aware of the kernel.

Returns

true - the application is being run in a kernel-aware simulator. false - otherwise.

Definition at line 169 of file [kernelaware.cpp](#).

17.11.2.3 void KernelAware::Print (const char * szStr_) [static]

Print.

Instruct the kernel-aware simulator to print a char string

Parameters

<i>szStr_</i>	
---------------	--

Examples:

[lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 160 of file [kernelaware.cpp](#).

17.11.2.4 void KernelAware::ProfileInit (const char * szStr_) [static]

ProfileInit.

Initializes the kernel-aware profiler. This function instructs the kernel-aware simulator to reset its accounting variables, and prepare to start counting profiling data tagged to the given string. How this is handled is the responsibility of the simulator.

Parameters

<i>szStr_</i>	String to use as a tag for the profiling session.
---------------	---

Definition at line 87 of file [kernelaware.cpp](#).

17.11.2.5 void KernelAware::ProfileReport (void) [static]

ProfileReport.

Instruct the kernel-aware simulator to print a report for its current profiling data.

Definition at line 108 of file [kernelaware.cpp](#).

17.11.2.6 void KernelAware::ProfileStart (void) [static]

ProfileStart.

Instruct the kernel-aware simulator to begin counting cycles towards the current profiling counter.

Definition at line 96 of file [kernelaware.cpp](#).

17.11.2.7 void KernelAware::ProfileStop (void) [static]

ProfileStop.

Instruct the kernel-aware simulator to end counting cycles relative to the current profiling counter's iteration.

Definition at line 102 of file [kernelaware.cpp](#).

17.11.2.8 void KernelAware::Trace (uint16_t u16File_, uint16_t u16Line_) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

<i>u16File_</i>	16-bit code representing the file
<i>u16Line_</i>	16-bit code representing the line in the file

Examples:

[lab11_mailboxes/main.cpp](#), and [lab8_messages/main.cpp](#).

Definition at line 120 of file [kernelaware.cpp](#).

17.11.2.9 void KernelAware::Trace (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

<i>u16File_</i>	16-bit code representing the file
<i>u16Line_</i>	16-bit code representing the line in the file
<i>u16Arg1_</i>	16-bit argument to the format string.

Definition at line 127 of file [kernelaware.cpp](#).

17.11.2.10 void KernelAware::Trace (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_)
[static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

<i>u16File_</i>	16-bit code representing the file
<i>u16Line_</i>	16-bit code representing the line in the file
<i>u16Arg1_</i>	16-bit argument to the format string.
<i>u16Arg2_</i>	16-bit argument to the format string.

Definition at line 135 of file [kernelaware.cpp](#).

```
17.11.2.11 void KernelAware::Trace_i( uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_,
    KernelAwareCommand_t eCmd_ ) [static], [private]
```

Trace_i.

Private function by which the class's [Trace\(\)](#) methods are reflected, which allows u16 to realize a modest code saving.

Parameters

<i>u16File_</i>	16-bit code representing the file
<i>u16Line_</i>	16-bit code representing the line in the file
<i>u16Arg1_</i>	16-bit argument to the format string.
<i>u16Arg2_</i>	16-bit argument to the format string.
<i>eCmd_</i>	Code indicating the number of arguments to emit.

Definition at line 144 of file [kernelaware.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/kernelaware.h](#)
- [/home/mo/mark3-source/embedded/kernel/kernelaware.cpp](#)

17.12 KernelAwareData_t Union Reference

This structure is used to communicate between the kernel and a kernel- aware host.

Public Attributes

- volatile uint16_t [au16Buffer](#) [5]

Raw binary contents of the struct.

- The [Profiler](#) struct contains data related to the code-execution profiling functionality provided by a kernel-aware host simulator.

The Trace struct contains data related to the display and output of kernel-trace strings on a kernel-aware host.

The Print struct contains data related to the display of arbitrary null-terminated ASCII strings on the kernel-aware host.

17.12.1 Detailed Description

This structure is used to communicate between the kernel and a kernel- aware host.

Its data contents is interpreted differently depending on the command executed (by means of setting the `g_u8KA` Command variable, as is done in the command handlers in this module). As a result, any changes to this struct by way of modifying or adding data must be mirrored in the kernel-aware simulator.

Definition at line 48 of file [kernelaware.cpp](#).

The documentation for this union was generated from the following file:

- [/home/mo/mark3-source/embedded/kernel/kernelaware.cpp](#)

17.13 KernelSWI Class Reference

Class providing the software-interrupt required for context-switching in the kernel.

```
#include <kernelswi.h>
```

Static Public Member Functions

- static void [Config](#) (void)
Config.
- static void [Start](#) (void)
Start.
- static void [Stop](#) (void)
Stop.
- static void [Clear](#) (void)
Clear.
- static void [Trigger](#) (void)
Trigger.
- static uint8_t [DI](#) ()
DI.
- static void [RI](#) (bool bEnable_)
RI.

17.13.1 Detailed Description

Class providing the software-interrupt required for context-switching in the kernel.

Definition at line 32 of file [kernelswi.h](#).

17.13.2 Member Function Documentation

17.13.2.1 void KernelSWI::Clear (void) [static]

Clear.

Clear the software interrupt

Definition at line 71 of file [kernelswi.cpp](#).

17.13.2.2 void KernelSWI::Config (void) [static]

Config.

Configure the software interrupt - must be called before any other software interrupt functions are called.

Definition at line 29 of file [kernelswi.cpp](#).

17.13.2.3 `uint8_t KernelSWI::DI () [static]`

DI.

Disable the SWI flag itself

Returns

previous status of the SWI, prior to the DI call

Definition at line 50 of file [kernelswi.cpp](#).

17.13.2.4 `void KernelSWI::RI (bool bEnable_) [static]`

RI.

Restore the state of the SWI to the value specified

Parameters

<i>bEnable_</i>	true - enable the SWI, false - disable SWI
-----------------	--

Definition at line 58 of file [kernelswi.cpp](#).

17.13.2.5 `void KernelSWI::Start (void) [static]`

Start.

Enable ("Start") the software interrupt functionality

Definition at line 37 of file [kernelswi.cpp](#).

17.13.2.6 `void KernelSWI::Stop (void) [static]`

Stop.

Disable the software interrupt functionality

Definition at line 44 of file [kernelswi.cpp](#).

17.13.2.7 `void KernelSWI::Trigger (void) [static]`

Trigger.

Call the software interrupt

Definition at line 77 of file [kernelswi.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h](#)
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp](#)

17.14 KernelTimer Class Reference

Hardware timer interface, used by all scheduling/timer subsystems.

```
#include <kerneltimer.h>
```

Static Public Member Functions

- static void [Config](#) (void)
Config.
- static void [Start](#) (void)
Start.
- static void [Stop](#) (void)
Stop.
- static uint8_t [DI](#) (void)
DI.
- static void [RI](#) (bool bEnable_)
RI.
- static void [EI](#) (void)
EI.
- static uint32_t [SubtractExpiry](#) (uint32_t u32Interval_)
SubtractExpiry.
- static uint32_t [TimeToExpiry](#) (void)
TimeToExpiry.
- static uint32_t [SetExpiry](#) (uint32_t u32Interval_)
SetExpiry.
- static uint32_t [GetOvertime](#) (void)
GetOvertime.
- static void [ClearExpiry](#) (void)
ClearExpiry.

Static Private Member Functions

- static uint16_t [Read](#) (void)
Read.

17.14.1 Detailed Description

Hardware timer interface, used by all scheduling/timer subsystems.

Definition at line 33 of file [kerneltimer.h](#).

17.14.2 Member Function Documentation

17.14.2.1 void KernelTimer::ClearExpiry (void) [static]

ClearExpiry.

Clear the hardware timer expiry register

Definition at line 142 of file [kerneltimer.cpp](#).

17.14.2.2 void KernelTimer::Config (void) [static]

Config.

Initializes the kernel timer before use

Definition at line 33 of file [kerneltimer.cpp](#).

17.14.2.3 `uint8_t KernelTimer::DI (void) [static]`

DI.

Disable the kernel timer's expiry interrupt

Definition at line 150 of file [kerneltimer.cpp](#).

17.14.2.4 `void KernelTimer::EI (void) [static]`

EI.

Enable the kernel timer's expiry interrupt

Definition at line 163 of file [kerneltimer.cpp](#).

17.14.2.5 `uint32_t KernelTimer::GetOvertime (void) [static]`

GetOvertime.

Return the number of ticks that have elapsed since the last expiry.

Returns

Number of ticks that have elapsed after last timer expiration

Definition at line 115 of file [kerneltimer.cpp](#).

17.14.2.6 `uint16_t KernelTimer::Read (void) [static],[private]`

Read.

Safely read the current value in the timer register

Returns

Value held in the timer register

Definition at line 66 of file [kerneltimer.cpp](#).

17.14.2.7 `void KernelTimer::RI (bool bEnable_) [static]`

RI.

Retstore the state of the kernel timer's expiry interrupt.

Parameters

<i>bEnable_</i>	1 enable, 0 disable
-----------------	---------------------

Definition at line 169 of file [kerneltimer.cpp](#).

17.14.2.8 `uint32_t KernelTimer::SetExpiry (uint32_t u32Interval_) [static]`

SetExpiry.

Resets the kernel timer's expiry interval to the specified value

Parameters

<i>u32Interval_</i>	Desired interval in ticks to set the timer for
---------------------	--

Returns

Actual number of ticks set (may be less than desired)

Definition at line 121 of file [kerneltimer.cpp](#).

17.14.2.9 void KernelTimer::Start (void) [static]

Start.

Starts the kernel time (must be configured first)

Definition at line 39 of file [kerneltimer.cpp](#).

17.14.2.10 void KernelTimer::Stop (void) [static]

Stop.

Shut down the kernel timer, used when no timers are scheduled

Definition at line 54 of file [kerneltimer.cpp](#).

17.14.2.11 uint32_t KernelTimer::SubtractExpiry (uint32_t u32Interval_) [static]

SubtractExpiry.

Subtract the specified number of ticks from the timer's expiry count register. Returns the new expiry value stored in the register.

Parameters

<i>u32Interval_</i>	Time (in HW-specific) ticks to subtract
---------------------	---

Returns

Value in ticks stored in the timer's expiry register

Definition at line 84 of file [kerneltimer.cpp](#).

17.14.2.12 uint32_t KernelTimer::TimeToExpiry (void) [static]

TimeToExpiry.

Returns the number of ticks remaining before the next timer expiry.

Returns

Time before next expiry in platform-specific ticks

Definition at line 95 of file [kerneltimer.cpp](#).

The documentation for this class was generated from the following files:

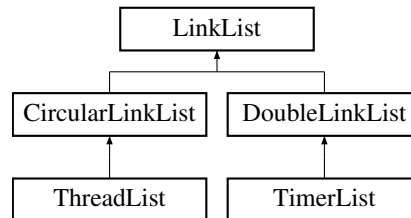
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kertimer.h](#)
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kertimer.cpp](#)

17.15 LinkedList Class Reference

Abstract-data-type from which all other linked-lists are derived.

```
#include <ll.h>
```

Inheritance diagram for LinkedList:



Public Member Functions

- void `Init ()`
Init.
- virtual void `Add (LinkedListNode *node_)=0`
Add.
- virtual void `Remove (LinkedListNode *node_)=0`
Remove.
- `LinkedListNode * GetHead ()`
GetHead.
- `LinkedListNode * GetTail ()`
GetTail.

Protected Attributes

- `LinkedListNode * m_pstHead`
Pointer to the head node in the list.
- `LinkedListNode * m_pstTail`
Pointer to the tail node in the list.

17.15.1 Detailed Description

Abstract-data-type from which all other linked-lists are derived.

Definition at line 113 of file `ll.h`.

17.15.2 Member Function Documentation

17.15.2.1 `virtual void LinkedList::Add (LinkedListNode * node_) [pure virtual]`

Add.

Add the linked list node to this linked list

Parameters

<i>node_</i>	Pointer to the node to add
--------------	----------------------------

Implemented in [CircularLinkedList](#), [DoubleLinkedList](#), and [ThreadList](#).

17.15.2.2 LinkListNode* LinkList::GetHead () [inline]

GetHead.

Get the head node in the linked list

Returns

Pointer to the head node in the list

Definition at line [152](#) of file [ll.h](#).

17.15.2.3 LinkListNode* LinkList::GetTail () [inline]

GetTail.

Get the tail node of the linked list

Returns

Pointer to the tail node in the list

Definition at line [161](#) of file [ll.h](#).

17.15.2.4 void LinkList::Init (void) [inline]

Init.

Clear the linked list.

Definition at line [125](#) of file [ll.h](#).

17.15.2.5 virtual void LinkList::Remove (LinkListNode * node_) [pure virtual]

Remove.

Add the linked list node to this linked list

Parameters

<i>node_</i>	Pointer to the node to remove
--------------	-------------------------------

Implemented in [CircularLinkedList](#), [DoubleLinkedList](#), and [ThreadList](#).

The documentation for this class was generated from the following file:

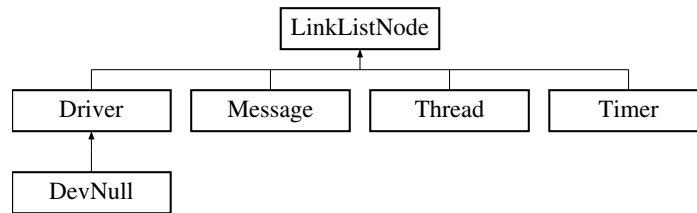
- [/home/mo/mark3-source/embedded/kernel/public/ll.h](#)

17.16 LinkListNode Class Reference

Basic linked-list node data structure.

```
#include <ll.h>
```

Inheritance diagram for LinkListNode:



Public Member Functions

- [LinkListNode * GetNext](#) (void)
GetNext.
- [LinkListNode * GetPrev](#) (void)
GetPrev.

Protected Member Functions

- void [ClearNode](#) ()
ClearNode.

Protected Attributes

- [LinkListNode * next](#)
Pointer to the next node in the list.
- [LinkListNode * prev](#)
Pointer to the previous node in the list.

Friends

- class **LinkList**
- class **DoubleLinkList**
- class **CircularLinkList**
- class **ThreadList**

17.16.1 Detailed Description

Basic linked-list node data structure.

This data is managed by the linked-list class types, and can be used transparently between them.

Definition at line 68 of file [ll.h](#).

17.16.2 Member Function Documentation

17.16.2.1 void LinkListNode::ClearNode () [protected]

ClearNode.

Initialize the linked list node, clearing its next and previous node.

Definition at line 40 of file [ll.cpp](#).

17.16.2.2 `LinkedListNode* LinkedListNode::GetNext (void)` `[inline]`

`GetNext`.

Returns a pointer to the next node in the list.

Returns

a pointer to the next node in the list.

Definition at line 92 of file [ll.h](#).

17.16.2.3 `LinkedListNode* LinkedListNode::GetPrev (void)` `[inline]`

`GetPrev`.

Returns a pointer to the previous node in the list.

Returns

a pointer to the previous node in the list.

Definition at line 101 of file [ll.h](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/ll.h](#)
- [/home/mo/mark3-source/embedded/kernel/ll.cpp](#)

17.17 Mailbox Class Reference

The [Mailbox](#) class implements an IPC mechnism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

```
#include <mailbox.h>
```

Public Member Functions

- void [Init](#) (void *pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)
Init.
- bool [Send](#) (void *pvData_)
Send.
- bool [SendTail](#) (void *pvData_)
SendTail.
- bool [Send](#) (void *pvData_, uint32_t u32TimeoutMS_)
Send.
- bool [SendTail](#) (void *pvData_, uint32_t u32TimeoutMS_)
SendTail.
- void [Receive](#) (void *pvData_)
Receive.
- void [ReceiveTail](#) (void *pvData_)
ReceiveTail.
- bool [Receive](#) (void *pvData_, uint32_t u32TimeoutMS_)
Receive.
- bool [ReceiveTail](#) (void *pvData_, uint32_t u32TimeoutMS_)
ReceiveTail.

Private Member Functions

- void * [GetHeadPointer](#) (void)
GetHeadPointer.
- void * [GetTailPointer](#) (void)
GetTailPointer.
- void [CopyData](#) (const void *src_, const void *dst_, uint16_t len_)
CopyData.
- void [MoveTailForward](#) (void)
MoveTailForward.
- void [MoveHeadForward](#) (void)
MoveHeadForward.
- void [MoveTailBackward](#) (void)
MoveTailBackward.
- void [MoveHeadBackward](#) (void)
MoveHeadBackward.
- bool [Send_i](#) (const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_)
Send_i.
- bool [Receive_i](#) (const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_)
Receive_i.

Private Attributes

- uint16_t [m_u16Head](#)
Current head index.
- uint16_t [m_u16Tail](#)
Current tail index.
- uint16_t [m_u16Count](#)
Count of items in the mailbox.
- volatile uint16_t [m_u16Free](#)
Current number of free slots in the mailbox.
- uint16_t [m_u16ElementSize](#)
Size of the objects tracked in this mailbox.
- const void * [m_pvBuffer](#)
Pointer to the data-buffer managed by this mailbox.
- [Semaphore m_clRecvSem](#)
Counting semaphore used to synchronize threads on the object.
- [Semaphore m_clSendSem](#)
Binary semaphore for send-blocked threads.

17.17.1 Detailed Description

The [Mailbox](#) class implements an IPC mechanism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

Examples:

[lab11_mailboxes/main.cpp](#).

Definition at line 35 of file [mailbox.h](#).

17.17.2 Member Function Documentation

17.17.2.1 `void Mailbox::CopyData (const void * src_, const void * dst_, uint16_t len_)` `[inline],[private]`

CopyData.

Perform a direct byte-copy from a source to a destination object.

Parameters

<i>src_</i>	Pointer to an object to read from
<i>dst_</i>	Pointer to an object to write to
<i>len_</i>	Length to copy (in bytes)

Definition at line 245 of file [mailbox.h](#).

17.17.2.2 `void* Mailbox::GetHeadPointer (void)` `[inline],[private]`

GetHeadPointer.

Return a pointer to the current head of the mailbox's internal circular buffer.

Returns

pointer to the head element in the mailbox

Definition at line 214 of file [mailbox.h](#).

17.17.2.3 `void* Mailbox::GetTailPointer (void)` `[inline],[private]`

GetTailPointer.

Return a pointer to the current tail of the mailbox's internal circular buffer.

Returns

pointer to the tail element in the mailbox

Definition at line 229 of file [mailbox.h](#).

17.17.2.4 `void Mailbox::Init (void * pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)`

Init.

Initialize the mailbox object prior to its use. This must be called before any calls can be made to the object.

Parameters

<i>pvBuffer_</i>	Pointer to the static buffer to use for the mailbox
<i>u16BufferSize_</i>	Size of the mailbox buffer, in bytes
<i>u16ElementSize_</i>	Size of each envelope, in bytes

Examples:

[lab11_mailboxes/main.cpp](#).

Definition at line 42 of file [mailbox.cpp](#).

17.17.2.5 `void Mailbox::MoveHeadBackward (void) [inline],[private]`

MoveHeadBackward.

Move the head index backward one element

Definition at line 302 of file [mailbox.h](#).

17.17.2.6 `void Mailbox::MoveHeadForward (void) [inline],[private]`

MoveHeadForward.

Move the head index forward one element

Definition at line 274 of file [mailbox.h](#).

17.17.2.7 `void Mailbox::MoveTailBackward (void) [inline],[private]`

MoveTailBackward.

Move the tail index backward one element

Definition at line 288 of file [mailbox.h](#).

17.17.2.8 `void Mailbox::MoveTailForward (void) [inline],[private]`

MoveTailForward.

Move the tail index forward one element

Definition at line 260 of file [mailbox.h](#).

17.17.2.9 `void Mailbox::Receive (void * pvData_)`

Receive.

Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

Parameters

<i>pvData_</i>	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
----------------	---

Examples:

[lab11_mailboxes/main.cpp](#).

Definition at line 80 of file [mailbox.cpp](#).

17.17.2.10 `bool Mailbox::Receive (void * pvData_, uint32_t u32TimeoutMS_)`

Receive.

Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

Parameters

<i>pvData_</i>	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
<i>u32TimeoutMS_</i> <i>S_</i>	Maximum time to wait for delivery.

Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 93 of file [mailbox.cpp](#).

17.17.2.11 `bool Mailbox::Receive_i (const void * pvData_, bool bTail_, uint32_t u32WaitTimeMS_) [private]`

Receive_i.

Internal method which implements all Read() methods in the class.

Parameters

<i>pvData_</i>	Pointer to the envelope data
<i>bTail_</i>	true - read from tail, false - read from head
<i>u32WaitTimeMS_</i> <i>S_</i>	Time to wait before timeout (in ms).

Returns

true - read successfully, false - timeout.

Definition at line 247 of file [mailbox.cpp](#).

17.17.2.12 `void Mailbox::ReceiveTail (void * pvData_)`

ReceiveTail.

Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

Parameters

<i>pvData_</i>	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
----------------	---

Definition at line 101 of file [mailbox.cpp](#).

17.17.2.13 `bool Mailbox::ReceiveTail (void * pvData_, uint32_t u32TimeoutMS_)`

ReceiveTail.

Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

Parameters

<i>pvData_</i>	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
<i>u32TimeoutMS_</i> <i>S_</i>	Maximum time to wait for delivery.

Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 114 of file [mailbox.cpp](#).

17.17.2.14 `bool Mailbox::Send (void * pvData_)`

Send.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

Parameters

<i>pvData_</i>	Pointer to the data object to send to the mailbox.
----------------	--

Returns

true - envelope was delivered, false - mailbox is full.

Examples:

[lab11_mailboxes/main.cpp](#).

Definition at line 122 of file [mailbox.cpp](#).

17.17.2.15 `bool Mailbox::Send (void * pvData_, uint32_t u32TimeoutMS_)`

Send.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

Parameters

<i>pvData_</i>	Pointer to the data object to send to the mailbox.
<i>u32TimeoutMS_</i>	Maximum time to wait for a free transmit slot

Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 147 of file [mailbox.cpp](#).

17.17.2.16 `bool Mailbox::Send_i (const void * pvData_, bool bTail_, uint32_t u32WaitTimeMS_) [private]`

Send_i.

Internal method which implements all [Send\(\)](#) methods in the class.

Parameters

<i>pvData_</i>	Pointer to the envelope data
<i>bTail_</i>	true - write to tail, false - write to head
<i>u32WaitTimeMS_</i>	Time to wait before timeout (in ms).

Returns

true - data successfully written, false - buffer full

Definition at line 165 of file [mailbox.cpp](#).

17.17.2.17 `bool Mailbox::SendTail (void * pvData_)`

SendTail.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

Parameters

<i>pvData_</i>	Pointer to the data object to send to the mailbox.
----------------	--

Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 134 of file [mailbox.cpp](#).

17.17.2.18 `bool Mailbox::SendTail (void * pvData_, uint32_t u32TimeoutMS_)`

SendTail.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

Parameters

<i>pvData_</i>	Pointer to the data object to send to the mailbox.
<i>u32TimeoutMS_</i>	Maximum time to wait for a free transmit slot

Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 155 of file [mailbox.cpp](#).

17.17.3 Member Data Documentation**17.17.3.1 Semaphore** `Mailbox::m_clSendSem` `[private]`

Binary semaphore for send-blocked threads.

Definition at line 372 of file [mailbox.h](#).

The documentation for this class was generated from the following files:

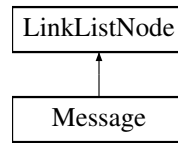
- [/home/mo/mark3-source/embedded/kernel/public/mailbox.h](#)
- [/home/mo/mark3-source/embedded/kernel/mailbox.cpp](#)

17.18 Message Class Reference

Class to provide message-based IPC services in the kernel.

```
#include <message.h>
```

Inheritance diagram for Message:



Public Member Functions

- void `Init` ()
Init.
- void `SetData` (void *pvData_)
SetData.
- void * `GetData` ()
GetData.
- void `SetCode` (uint16_t u16Code_)
SetCode.
- uint16_t `GetCode` ()
GetCode.

Private Attributes

- void * `m_pvData`
Pointer to the message data.
- uint16_t `m_u16Code`
Message code, providing context for the message.

Additional Inherited Members

17.18.1 Detailed Description

Class to provide message-based IPC services in the kernel.

Examples:

[lab8_messages/main.cpp](#).

Definition at line 99 of file [message.h](#).

17.18.2 Member Function Documentation

17.18.2.1 uint16_t Message::GetCode () `[inline]`

`GetCode.`

Return the code set in the message upon receipt

Returns

user code set in the object

Examples:

[lab8_messages/main.cpp](#).

Definition at line 143 of file [message.h](#).

17.18.2.2 `void* Message::GetData () [inline]`

GetData.

Get the data pointer stored in the message upon receipt

Returns

Pointer to the data set in the message object

Examples:

[lab8_messages/main.cpp](#).

Definition at line 125 of file [message.h](#).

17.18.2.3 `void Message::Init(void) [inline]`

Init.

Initialize the data and code in the message.

Definition at line 107 of file [message.h](#).

17.18.2.4 `void Message::SetCode (uint16_t u16Code_) [inline]`

SetCode.

Set the code in the message before transmission

Parameters

<i>u16Code_</i>	Data code to set in the object
-----------------	--------------------------------

Examples:

[lab8_messages/main.cpp](#).

Definition at line 134 of file [message.h](#).

17.18.2.5 `void Message::SetData (void * pvData_) [inline]`

SetData.

Set the data pointer for the message before transmission.

Parameters

<code>pvData_</code>	Pointer to the data object to send in the message
----------------------	---

Examples:

[lab8_messages/main.cpp](#).

Definition at line 116 of file [message.h](#).

The documentation for this class was generated from the following file:

- [/home/mo/mark3-source/embedded/kernel/public/message.h](#)

17.19 MessageQueue Class Reference

List of messages, used as the channel for sending and receiving messages between threads.

```
#include <message.h>
```

Public Member Functions

- void [Init](#) ()
Init.
- [Message](#) * [Receive](#) ()
Receive.
- [Message](#) * [Receive](#) (uint32_t u32TimeWaitMS_)
Receive.
- void [Send](#) ([Message](#) *pclSrc_)
Send.
- uint16_t [GetCount](#) ()
GetCount.

Private Member Functions

- [Message](#) * [Receive_i](#) (uint32_t u32TimeWaitMS_)
Receive_i.

Private Attributes

- [Semaphore](#) [m_clSemaphore](#)
Counting semaphore used to manage thread blocking.
- [DoubleLinkedList](#) [m_clLinkList](#)
List object used to store messages.

17.19.1 Detailed Description

List of messages, used as the channel for sending and receiving messages between threads.

Examples:

[lab8_messages/main.cpp](#).

Definition at line 201 of file [message.h](#).

17.19.2 Member Function Documentation

17.19.2.1 uint16_t MessageQueue::GetCount ()

GetCount.

Return the number of messages pending in the "receive" queue.

Returns

Count of pending messages in the queue.

Definition at line 160 of file [message.cpp](#).

17.19.2.2 void MessageQueue::Init (void)

Init.

Initialize the message queue prior to use.

Examples:

[lab8_messages/main.cpp](#).

Definition at line 90 of file [message.cpp](#).

17.19.2.3 Message * MessageQueue::Receive ()

Receive.

Receive a message from the message queue. If the message queue is empty, the thread will block until a message is available.

Returns

Pointer to a message object at the head of the queue

Examples:

[lab8_messages/main.cpp](#).

Definition at line 96 of file [message.cpp](#).

17.19.2.4 Message * MessageQueue::Receive (uint32_t u32TimeWaitMS_)

Receive.

Receive a message from the message queue. If the message queue is empty, the thread will block until a message is available for the duration specified. If no message arrives within that duration, the call will return with NULL.

Parameters

<i>u32TimeWaitMS_</i>	The amount of time in ms to wait for a message before timing out and unblocking the waiting thread.
-----------------------	---

Returns

Pointer to a message object at the head of the queue or NULL on timeout.

Definition at line 107 of file [message.cpp](#).

17.19.2.5 **Message * MessageQueue::Receive_i (uint32_t u32TimeWaitMS_)** [private]

Receive_i.

Internal function used to abstract timed and un-timed Receive calls.

Parameters

<i>u32TimeWaitMS_</i>	Time (in ms) to block, 0 for un-timed call.
-----------------------	---

Returns

Pointer to a message, or 0 on timeout.

Definition at line 115 of file [message.cpp](#).

17.19.2.6 **void MessageQueue::Send (Message * pclSrc_)**

Send.

Send a message object into this message queue. Will un-block the first waiting thread blocked on this queue if that occurs.

Parameters

<i>pclSrc_</i>	Pointer to the message object to add to the queue
----------------	---

Examples:

[lab8_messages/main.cpp](#).

Definition at line 144 of file [message.cpp](#).

The documentation for this class was generated from the following files:

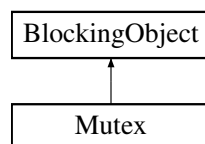
- [/home/mo/mark3-source/embedded/kernel/public/message.h](#)
- [/home/mo/mark3-source/embedded/kernel/message.cpp](#)

17.20 Mutex Class Reference

Mutual-exclusion locks, based on [BlockingObject](#).

```
#include <mutex.h>
```

Inheritance diagram for Mutex:



Public Member Functions

- void [Init](#) ()
Init.
- void [Claim](#) ()

- Claim.*
- bool [Claim](#) (uint32_t u32WaitTimeMS_)
- Claim.*
- void [WakeMe](#) ([Thread](#) *pclOwner_)
- WakeMe.*
- void [Release](#) ()
- Release.*

Private Member Functions

- uint8_t [WakeNext](#) ()
- WakeNext.*
- bool [Claim_i](#) (uint32_t u32WaitTimeMS_)
- Claim_i.*

Private Attributes

- uint8_t [m_u8Recurse](#)
- The recursive lock-count when a mutex is claimed multiple times by the same owner.*
- bool [m_bReady](#)
- State of the mutex - true = ready, false = claimed.*
- uint8_t [m_u8MaxPri](#)
- Maximum priority of thread in queue, used for priority inheritance.*
- [Thread](#) * [m_pclOwner](#)
- Pointer to the thread that owns the mutex (when claimed)*

Additional Inherited Members

17.20.1 Detailed Description

Mutual-exclusion locks, based on [BlockingObject](#).

Examples:

[lab5_mutexes/main.cpp](#).

Definition at line 68 of file [mutex.h](#).

17.20.2 Member Function Documentation

17.20.2.1 void Mutex::Claim (void)

Claim.

Claim the mutex. When the mutex is claimed, no other thread can claim a region protected by the object. If another [Thread](#) currently holds the [Mutex](#) when the Claim method is called, that [Thread](#) will block until the current owner of the mutex releases the [Mutex](#).

If the calling [Thread](#)'s priority is lower than that of a [Thread](#) that currently owns the [Mutex](#) object, then the priority of that [Thread](#) will be elevated to that of the highest-priority calling object until the [Mutex](#) is released. This property is known as "Priority Inheritance"

Note: A single thread can recursively claim a mutex up to a count of

1. Attempting to claim a mutex beyond that will cause a kernel panic.

Examples:

[lab5_mutexes/main.cpp](#).

Definition at line 215 of file [mutex.cpp](#).

17.20.2.2 bool Mutex::Claim (uint32_t u32WaitTimeMS_)

Claim.

Claim a mutex, with timeout.

Parameters

<i>u32WaitTimeMS_</i>	
<i>S_</i>	

Returns

true - mutex was claimed within the time period specified
false - mutex operation timed-out before the claim operation.

Definition at line 226 of file [mutex.cpp](#).

17.20.2.3 bool Mutex::Claim_i (uint32_t u32WaitTimeMS_) [private]

Claim_i.

Abstracts out timed/non-timed mutex claim operations.

Parameters

<i>u32WaitTimeMS_</i>	Time in MS to wait, 0 for infinite
<i>S_</i>	

Returns

true on successful claim, false otherwise

Definition at line 113 of file [mutex.cpp](#).

17.20.2.4 void Mutex::Init (void)

Init.

Initialize a mutex object for use - must call this function before using the object.

Examples:

[lab5_mutexes/main.cpp](#).

Definition at line 102 of file [mutex.cpp](#).

17.20.2.5 void Mutex::Release ()

Release.

Release the mutex. When the mutex is released, another object can enter the mutex-protected region.

If there are Threads waiting for the [Mutex](#) to become available, then the highest priority [Thread](#) will be unblocked at this time and will claim the [Mutex](#) lock immediately - this may result in an immediate context switch, depending on relative priorities.

If the calling [Thread](#)'s priority was boosted as a result of priority inheritance, the [Thread](#)'s previous priority will also be restored at this time.

Note that if a [Mutex](#) is held recursively, it must be Release'd the same number of times that it was Claim'd before it will be available for use by another [Thread](#).

Examples:

[lab5_mutexes/main.cpp](#).

Definition at line 233 of file [mutex.cpp](#).

17.20.2.6 void [Mutex::WakeMe](#) ([Thread](#) * [pclOwner_](#))

[WakeMe](#).

Wake a thread blocked on the mutex. This is an internal function used for implementing timed mutexes relying on timer callbacks. Since these do not have access to the private data of the mutex and its base classes, we have to wrap this as a public method - do not use this for any other purposes.

Parameters

pclOwner_	Thread to unblock from this object.
---------------------------	---

Definition at line 71 of file [mutex.cpp](#).

17.20.2.7 uint8_t [Mutex::WakeNext](#) () [private]

[WakeNext](#).

Wake the next thread waiting on the [Mutex](#).

Definition at line 80 of file [mutex.cpp](#).

The documentation for this class was generated from the following files:

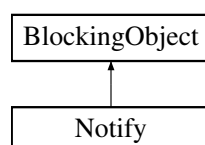
- [/home/mo/mark3-source/embedded/kernel/public/mutex.h](#)
- [/home/mo/mark3-source/embedded/kernel/mutex.cpp](#)

17.21 Notify Class Reference

The [Notify](#) class is a blocking object type, that allows one or more threads to wait for an event to occur before resuming operation.

```
#include <notify.h>
```

Inheritance diagram for [Notify](#):



Public Member Functions

- void [Init](#) (void)
Init.
- void [Signal](#) (void)
Signal.
- void [Wait](#) (bool *pbFlag_)
Wait.
- bool [Wait](#) (uint32_t u32WaitTimeMS_, bool *pbFlag_)
Wait.
- void [WakeMe](#) ([Thread](#) *pclChosenOne_)
WakeMe.

Additional Inherited Members

17.21.1 Detailed Description

The [Notify](#) class is a blocking object type, that allows one or more threads to wait for an event to occur before resuming operation.

Examples:

[lab10_notifications/main.cpp](#).

Definition at line 33 of file [notify.h](#).

17.21.2 Member Function Documentation

17.21.2.1 void Notify::Init (void)

Init.

Initialize the Notification object prior to use.

Examples:

[lab10_notifications/main.cpp](#).

Definition at line 56 of file [notify.cpp](#).

17.21.2.2 void Notify::Signal (void)

Signal.

Signal the notification object. This will cause the highest priority thread currently blocking on the object to wake. If no threads are currently blocked on the object, the call has no effect.

Examples:

[lab10_notifications/main.cpp](#).

Definition at line 62 of file [notify.cpp](#).

17.21.2.3 void Notify::Wait (bool * pbFlag_)

Wait.

Block the current thread, waiting for a signal on the object.

Parameters

<i>pbFlag_</i>	Flag set to false on block, and true upon wakeup.
----------------	---

Examples:

[lab10_notifications/main.cpp](#).

Definition at line 87 of file [notify.cpp](#).

17.21.2.4 `bool Notify::Wait (uint32_t u32WaitTimeMS_, bool * pbFlag_)`

Wait.

Block the current thread, waiting for a signal on the object.

Parameters

<i>u32WaitTimeMS_</i>	Time to wait for the notification event.
<i>pbFlag_</i>	Flag set to false on block, and true upon wakeup.

Returns

true on notification, false on timeout

Definition at line 106 of file [notify.cpp](#).

17.21.2.5 `void Notify::WakeMe (Thread * pclChosenOne_)`

WakeMe.

Wake the specified thread from its current blocking queue. Note that this is only public in order to be accessible from a timer callack.

Parameters

<i>pclChosenOne_</i>	Thread to wake up
----------------------	-----------------------------------

Definition at line 146 of file [notify.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/notify.h](#)
- [/home/mo/mark3-source/embedded/kernel/notify.cpp](#)

17.22 Profiler Class Reference

System profiling timer interface.

```
#include <kernelprofile.h>
```

Static Public Member Functions

- static void [Init](#) ()
Init.
- static void [Start](#) ()

- Start.*
- static void [Stop](#) ()
- Stop.*
- static uint16_t [Read](#) ()
- Read.*
- static void [Process](#) ()
- Process.*
- static uint32_t [GetEpoch](#) ()
- GetEpoch.*

17.22.1 Detailed Description

System profiling timer interface.

Definition at line 37 of file [kernelprofile.h](#).

17.22.2 Member Function Documentation

17.22.2.1 `static uint32_t Profiler::GetEpoch () [inline],[static]`

GetEpoch.

Return the current timer epoch

Definition at line 81 of file [kernelprofile.h](#).

17.22.2.2 `void Profiler::Init (void) [static]`

Init.

Initialize the global system profiler. Must be called prior to use.

Definition at line 32 of file [kernelprofile.cpp](#).

17.22.2.3 `void Profiler::Process (void) [static]`

Process.

Process the profiling counters from ISR.

Definition at line 70 of file [kernelprofile.cpp](#).

17.22.2.4 `uint16_t Profiler::Read () [static]`

Read.

Read the current tick count in the timer.

Definition at line 58 of file [kernelprofile.cpp](#).

17.22.2.5 `void Profiler::Start (void) [static]`

Start.

Start the global profiling timer service.

Definition at line 42 of file [kernelprofile.cpp](#).

17.22.2.6 void Profiler::Stop () [static]

Stop.

Stop the global profiling timer service

Definition at line 51 of file [kernelprofile.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h](#)
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp](#)

17.23 ProfileTimer Class Reference

Profiling timer.

```
#include <profile.h>
```

Public Member Functions

- void [Init](#) ()
Init.
- void [Start](#) ()
Start.
- void [Stop](#) ()
Stop.
- uint32_t [GetAverage](#) ()
GetAverage.
- uint32_t [GetCurrent](#) ()
GetCurrent.

Private Member Functions

- uint32_t [ComputeCurrentTicks](#) (uint16_t u16Count_, uint32_t u32Epoch_)
ComputeCurrentTicks.

Private Attributes

- uint32_t [m_u32Cumulative](#)
Cumulative tick-count for this timer.
- uint32_t [m_u32CurrentIteration](#)
Tick-count for the current iteration.
- uint16_t [m_u16Initial](#)
Initial count.
- uint32_t [m_u32InitialEpoch](#)
Initial Epoch.
- uint16_t [m_u16Iterations](#)
Number of iterations executed for this profiling timer.
- bool [m_bActive](#)
Wheter or not the timer is active or stopped.

17.23.1 Detailed Description

Profiling timer.

This class is used to perform high-performance profiling of code to see how `uint32_t` certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

Definition at line 70 of file [profile.h](#).

17.23.2 Member Function Documentation

17.23.2.1 `uint32_t ProfileTimer::ComputeCurrentTicks (uint16_t u16Count_, uint32_t u32Epoch_)` `[private]`

ComputeCurrentTicks.

Figure out how many ticks have elapsed in this iteration

Parameters

<i>u16Count_</i>	Current timer count
<i>u32Epoch_</i>	Current timer epoch

Returns

Current tick count

Definition at line 112 of file [profile.cpp](#).

17.23.2.2 `uint32_t ProfileTimer::GetAverage ()`

GetAverage.

Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 85 of file [profile.cpp](#).

17.23.2.3 `uint32_t ProfileTimer::GetCurrent ()`

GetCurrent.

Return the current tick count held by the profiler. Valid for both active and stopped timers.

Returns

The currently held tick count.

Definition at line 95 of file [profile.cpp](#).

17.23.2.4 `void ProfileTimer::Init (void)`

Init.

Initialize the profiling timer prior to use. Can also be used to reset a timer that's been used previously.

Definition at line 43 of file [profile.cpp](#).

17.23.2.5 void ProfileTimer::Start (void)

Start.

Start a profiling session, if the timer is not already active. Has no effect if the timer is already active.

Definition at line 52 of file [profile.cpp](#).

17.23.2.6 void ProfileTimer::Stop ()

Stop.

Stop the current profiling session, adding to the cumulative time for this timer, and the total iteration count.

Definition at line 66 of file [profile.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/profile.h](#)
- [/home/mo/mark3-source/embedded/kernel/profile.cpp](#)

17.24 Quantum Class Reference

Static-class used to implement [Thread](#) quantum functionality, which is a key part of round-robin scheduling.

```
#include <quantum.h>
```

Static Public Member Functions

- static void [UpdateTimer](#) ()
UpdateTimer.
- static void [AddThread](#) ([Thread](#) *pclThread_)
AddThread.
- static void [RemoveThread](#) ()
RemoveThread.
- static void [SetInTimer](#) (void)
SetInTimer.
- static void [ClearInTimer](#) (void)
ClearInTimer.

Static Private Member Functions

- static void [SetTimer](#) ([Thread](#) *pclThread_)
SetTimer.

17.24.1 Detailed Description

Static-class used to implement [Thread](#) quantum functionality, which is a key part of round-robin scheduling.

Definition at line 41 of file [quantum.h](#).

17.24.2 Member Function Documentation

17.24.2.1 void Quantum::AddThread (Thread * *pcThread_*) [static]

AddThread.

Add the thread to the quantum timer. Only one thread can own the quantum, since only one thread can be running on a core at a time.

Definition at line 88 of file [quantum.cpp](#).

17.24.2.2 static void Quantum::ClearInTimer (void) [inline],[static]

ClearInTimer.

Clear the flag once the timer callback function has been completed.

Definition at line 84 of file [quantum.h](#).

17.24.2.3 void Quantum::RemoveThread (void) [static]

RemoveThread.

Remove the thread from the quantum timer. This will cancel the timer.

Definition at line 117 of file [quantum.cpp](#).

17.24.2.4 static void Quantum::SetInTimer (void) [inline],[static]

SetInTimer.

Set a flag to indicate that the CPU is currently running within the timer-callback routine. This prevents the [Quantum](#) timer from being updated in the middle of a callback cycle, potentially resulting in the kernel timer becoming disabled.

Definition at line 77 of file [quantum.h](#).

17.24.2.5 void Quantum::SetTimer (Thread * *pcThread_*) [static],[private]

SetTimer.

Set up the quantum timer in the timer scheduler. This creates a one-shot timer, which calls a static callback in [quantum.cpp](#) that on expiry will pivot the head of the threadlist for the thread's priority. This is the mechanism that provides round-robin scheduling in the system.

Parameters

<i>pcThread_</i>	Pointer to the thread to set the Quantum timer on
------------------	---

Definition at line 78 of file [quantum.cpp](#).

17.24.2.6 void Quantum::UpdateTimer (void) [static]

UpdateTimer.

This function is called to update the thread quantum timer whenever something in the scheduler has changed. This can result in the timer being re-loaded or started. The timer is never stopped, but it may be ignored on expiry.

Definition at line 130 of file [quantum.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/quantum.h](#)

- </home/mo/mark3-source/embedded/kernel/quantum.cpp>

17.25 Scheduler Class Reference

Priority-based round-robin [Thread](#) scheduling, using ThreadLists for housekeeping.

```
#include <scheduler.h>
```

Static Public Member Functions

- static void [Init](#) ()
Init.
- static void [Schedule](#) ()
Schedule.
- static void [Add](#) ([Thread](#) *pciThread_)
Add.
- static void [Remove](#) ([Thread](#) *pciThread_)
Remove.
- static bool [SetScheduler](#) (bool bEnable_)
SetScheduler.
- static [Thread](#) * [GetCurrentThread](#) ()
GetCurrentThread.
- static volatile [Thread](#) * [GetNextThread](#) ()
GetNextThread.
- static [ThreadList](#) * [GetThreadList](#) (uint8_t u8Priority_)
GetThreadList.
- static [ThreadList](#) * [GetStopList](#) ()
GetStopList.
- static uint8_t [IsEnabled](#) ()
IsEnabled.
- static void [QueueScheduler](#) ()
QueueScheduler.

Static Private Attributes

- static bool [m_bEnabled](#)
[Scheduler](#)'s state - enabled or disabled.
- static bool [m_bQueuedSchedule](#)
Variable representing whether or not there's a queued scheduler operation.
- static [ThreadList](#) [m_clStopList](#)
[ThreadList](#) for all stopped threads.
- static [ThreadList](#) [m_aclPriorities](#) [NUM_PRIORITIES]
[ThreadLists](#) for all threads at all priorities.
- static uint8_t [m_u8PriFlag](#)
Bitmap flag for each.

17.25.1 Detailed Description

Priority-based round-robin [Thread](#) scheduling, using ThreadLists for housekeeping.

Definition at line 62 of file [scheduler.h](#).

17.25.2 Member Function Documentation

17.25.2.1 void Scheduler::Add (Thread * *pclThread_*) [static]

Add.

Add a thread to the scheduler at its current priority level.

Parameters

<i>pclThread_</i>	Pointer to the thread to add to the scheduler
-------------------	---

Definition at line 113 of file [scheduler.cpp](#).

17.25.2.2 static Thread* Scheduler::GetCurrentThread () [inline],[static]

GetCurrentThread.

Return the pointer to the currently-running thread.

Returns

Pointer to the currently-running thread

Examples:

[lab9_dynamic_threads/main.cpp](#).

Definition at line 121 of file [scheduler.h](#).

17.25.2.3 static volatile Thread* Scheduler::GetNextThread () [inline],[static]

GetNextThread.

Return the pointer to the thread that should run next, according to the last run of the scheduler.

Returns

Pointer to the next-running thread

Definition at line 131 of file [scheduler.h](#).

17.25.2.4 static ThreadList* Scheduler::GetStopList () [inline],[static]

GetStopList.

Return the pointer to the list of threads that are in the scheduler's stopped state.

Returns

Pointer to the [ThreadList](#) containing the stopped threads

Definition at line 153 of file [scheduler.h](#).

17.25.2.5 static ThreadList* Scheduler::GetThreadList (uint8_t *u8Priority_*) [inline],[static]

GetThreadList.

Return the pointer to the active list of threads that are at the given priority level in the scheduler.

Parameters

<i>u8Priority_</i>	Priority level of
--------------------	-------------------

Returns

Pointer to the [ThreadList](#) for the given priority level

Definition at line 143 of file [scheduler.h](#).

17.25.2.6 `void Scheduler::Init (void) [static]`

Init.

Intialize the scheduler, must be called before use.

Definition at line 64 of file [scheduler.cpp](#).

17.25.2.7 `static uint8_t Scheduler::IsEnabled () [inline],[static]`

IsEnabled.

Return the current state of the scheduler - whether or not scheudling is enabled or disabled.

Returns

true - scheduler enabled, false - disabled

Definition at line 163 of file [scheduler.h](#).

17.25.2.8 `static void Scheduler::QueueScheduler () [inline],[static]`

QueueScheduler.

Tell the kernel to perform a scheduling operation as soon as the scheduler is re-enabled.

Definition at line 171 of file [scheduler.h](#).

17.25.2.9 `void Scheduler::Remove (Thread * pciThread_) [static]`

Remove.

Remove a thread from the scheduler at its current priority level.

Parameters

<i>pciThread_</i>	Pointer to the thread to be removed from the scheduler
-------------------	--

Definition at line 119 of file [scheduler.cpp](#).

17.25.2.10 `void Scheduler::Schedule () [static]`

Schedule.

Run the scheduler, determines the next thread to run based on the current state of the threads. Note that the next-thread chosen from this function is only valid while in a critical section.

Definition at line 76 of file [scheduler.cpp](#).

17.25.2.11 `bool Scheduler::SetScheduler (bool bEnable_) [static]`

SetScheduler.

Set the active state of the scheduler. When the scheduler is disabled, the *next thread* is never set; the currently running thread will run forever until the scheduler is enabled again. Care must be taken to ensure that we don't end up trying to block while the scheduler is disabled, otherwise the system ends up in an unusable state.

Parameters

<i>bEnable_</i>	true to enable, false to disable the scheduler
-----------------	--

Definition at line 125 of file [scheduler.cpp](#).

The documentation for this class was generated from the following files:

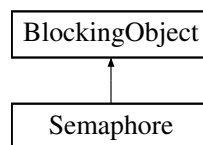
- [/home/mo/mark3-source/embedded/kernel/public/scheduler.h](#)
- [/home/mo/mark3-source/embedded/kernel/scheduler.cpp](#)

17.26 Semaphore Class Reference

Binary & Counting semaphores, based on [BlockingObject](#) base class.

```
#include <ksemaphore.h>
```

Inheritance diagram for Semaphore:



Public Member Functions

- void [Init](#) (uint16_t u16InitVal_, uint16_t u16MaxVal_)
Initialize a semaphore before use.
- bool [Post](#) ()
Increment the semaphore count.
- void [Pend](#) ()
Decrement the semaphore count.
- uint16_t [GetCount](#) ()
Return the current semaphore counter.
- bool [Pend](#) (uint32_t u32WaitTimeMS_)
Decrement the semaphore count.
- void [WakeMe](#) ([Thread](#) *pclChosenOne_)
Wake a thread blocked on the semaphore.

Private Member Functions

- uint8_t [WakeNext](#) ()
Wake the next thread waiting on the semaphore.
- bool [Pend_i](#) (uint32_t u32WaitTimeMS_)
Pend_i.

Private Attributes

- `uint16_t m_u16Value`
Current count held by the semaphore.
- `uint16_t m_u16MaxValue`
Maximum count that can be held by this semaphore.

Additional Inherited Members

17.26.1 Detailed Description

Binary & Counting semaphores, based on [BlockingObject](#) base class.

Examples:

[buffalogger/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab6_timers/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).↵

Definition at line 37 of file [ksemaphore.h](#).

17.26.2 Member Function Documentation

17.26.2.1 `uint16_t Semaphore::GetCount ()`

Return the current semaphore counter.

This can be usedd by a thread to bypass blocking on a semaphore - allowing it to do other things until a non-zero count is returned, instead of blocking until the semaphore is posted.

Returns

The current semaphore counter value.

Definition at line 241 of file [ksemaphore.cpp](#).

17.26.2.2 `void Semaphore::Init (uint16_t u16InitVal_, uint16_t u16MaxVal_)`

Initialize a semaphore before use.

Must be called before attempting post/pend operations on the object.

This initialization is required to configure the behavior of the semaphore with regards to the initial and maximum values held by the semaphore. By providing access to the raw initial and maximum count elements of the semaphore, these objects are able to be used as either counting or binary semaphores.

To configure a semaphore object for use as a binary semaphore, set values of 0 and 1 respectively for the initial/maximum value parameters.

Any other combination of values can be used to implement a counting semaphore.

Parameters

<code>u16InitVal_</code>	Initial value held by the semaphore
<code>u16MaxVal_</code>	Maximum value for the semaphore. Must be nonzero.

Examples:

[buffalogger/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab6_timers/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).↵

Definition at line 102 of file [ksemaphore.cpp](#).

17.26.2.3 void Semaphore::Pend ()

Decrement the semaphore count.

If the count is zero, the calling [Thread](#) will block until the semaphore is posted, and the [Thread](#)'s priority is higher than that of any other [Thread](#) blocked on the object.

Examples:

[buffalogger/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab6_timers/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 223 of file [ksemaphore.cpp](#).

17.26.2.4 bool Semaphore::Pend (uint32_t u32WaitTimeMS_)

Decrement the semaphore count.

If the count is zero, the thread will block until the semaphore is pended. If the specified interval expires before the thread is unblocked, then the status is returned back to the user.

Returns

true - semaphore was acquired before the timeout false - timeout occurred before the semaphore was claimed.

Definition at line 234 of file [ksemaphore.cpp](#).

17.26.2.5 bool Semaphore::Pend_i (uint32_t u32WaitTimeMS_) [private]

Pend_i.

Internal function used to abstract timed and untimed semaphore pend operations.

Parameters

u32WaitTimeMS_	Time in MS to wait
--------------------------------	--------------------

Returns

true on success, false on failure.

Definition at line 167 of file [ksemaphore.cpp](#).

17.26.2.6 bool Semaphore::Post ()

Increment the semaphore count.

If the semaphore count is zero at the time this is called, and there are threads blocked on the object, this will immediately unblock the highest-priority blocked [Thread](#).

Note that if the priority of that [Thread](#) is higher than the current thread's priority, a context switch will occur and control will be relinquished to that [Thread](#).

Returns

true if the semaphore was posted, false if the count is already maxed out.

Examples:

[buffalogger/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab6_timers/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 114 of file [ksemaphore.cpp](#).

17.26.2.7 void Semaphore::WakeMe (Thread * *pcIChosenOne_*)

Wake a thread blocked on the semaphore.

This is an internal function used for implementing timed semaphores relying on timer callbacks. Since these do not have access to the private data of the semaphore and its base classes, we have to wrap this as a public method - do not use this for any other purposes.

Definition at line 75 of file [ksemaphore.cpp](#).

17.26.2.8 uint8_t Semaphore::WakeNext () [private]

Wake the next thread waiting on the semaphore.

Used internally.

Definition at line 84 of file [ksemaphore.cpp](#).

The documentation for this class was generated from the following files:

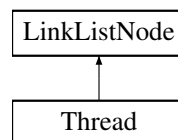
- [/home/mo/mark3-source/embedded/kernel/public/ksemaphore.h](#)
- [/home/mo/mark3-source/embedded/kernel/ksemaphore.cpp](#)

17.27 Thread Class Reference

Object providing fundamental multitasking support in the kernel.

```
#include <thread.h>
```

Inheritance diagram for Thread:



Public Member Functions

- void [Init](#) (K_WORD *pwStack_, uint16_t u16StackSize_, uint8_t u8Priority_, [ThreadEntry_t](#) pfEntryPoint_, void *pvArg_)
Init.
- void [Start](#) ()
Start.
- void [Stop](#) ()
Stop.
- [ThreadList](#) * [GetOwner](#) (void)
GetOwner.
- [ThreadList](#) * [GetCurrent](#) (void)
GetCurrent.
- uint8_t [GetPriority](#) (void)
GetPriority.
- uint8_t [GetCurPriority](#) (void)
GetCurPriority.
- void [SetQuantum](#) (uint16_t u16Quantum_)

- SetQuantum.*
- uint16_t [GetQuantum](#) (void)
- GetQuantum.*
- void [SetCurrent](#) ([ThreadList](#) *pclNewList_)
- SetCurrent.*
- void [SetOwner](#) ([ThreadList](#) *pclNewList_)
- SetOwner.*
- void [SetPriority](#) (uint8_t u8Priority_)
- SetPriority.*
- void [InheritPriority](#) (uint8_t u8Priority_)
- InheritPriority.*
- void [Exit](#) ()
- Exit.*
- void [SetID](#) (uint8_t u8ID_)
- SetID.*
- uint8_t [GetID](#) ()
- GetID.*
- uint16_t [GetStackSlack](#) ()
- GetStackSlack.*
- uint16_t [GetEventFlagMask](#) ()
- GetEventFlagMask* returns the thread's current event-flag mask, which is used in conjunction with the [EventFlag](#) blocking object type.
- void [SetEventFlagMask](#) (uint16_t u16Mask_)
- SetEventFlagMask* Sets the active event flag bitfield mask.
- void [SetEventFlagMode](#) ([EventFlagOperation_t](#) eMode_)
- SetEventFlagMode* Sets the active event flag operation mode.
- [EventFlagOperation_t](#) [GetEventFlagMode](#) ()
- GetEventFlagMode* Returns the thread's event flag's operating mode.
- [Timer](#) * [GetTimer](#) ()
- Return a pointer to the thread's timer object.*
- void [SetExpired](#) (bool bExpired_)
- SetExpired.*
- bool [GetExpired](#) ()
- GetExpired.*
- void [InitIdle](#) ()
- InitIdle* Initialize this [Thread](#) object as the [Kernel](#)'s idle thread.
- [ThreadState_t](#) [GetState](#) ()
- GetState* Returns the current state of the thread to the caller.
- void [SetState](#) ([ThreadState_t](#) eState_)
- SetState* Set the thread's state to a new value.

Static Public Member Functions

- static void [Sleep](#) (uint32_t u32TimeMs_)
- Sleep.*
- static void [USleep](#) (uint32_t u32TimeUs_)
- USleep.*
- static void [Yield](#) (void)
- Yield.*

Private Member Functions

- void [SetPriorityBase](#) (uint8_t u8Priority_)
SetPriorityBase.

Static Private Member Functions

- static void [ContextSwitchSWI](#) (void)
ContextSwitchSWI.

Private Attributes

- [K_WORD](#) * [m_pwStackTop](#)
Pointer to the top of the thread's stack.
- [K_WORD](#) * [m_pwStack](#)
Pointer to the thread's stack.
- uint8_t [m_u8ThreadID](#)
Thread ID.
- uint8_t [m_u8Priority](#)
Default priority of the thread.
- uint8_t [m_u8CurPriority](#)
Current priority of the thread (priority inheritance)
- [ThreadState_t](#) [m_eState](#)
Enum indicating the thread's current state.
- uint16_t [m_u16StackSize](#)
Size of the stack (in bytes)
- [ThreadList](#) * [m_pclCurrent](#)
Pointer to the thread-list where the thread currently resides.
- [ThreadList](#) * [m_pclOwner](#)
Pointer to the thread-list where the thread resides when active.
- [ThreadEntry_t](#) [m_pfEntryPoint](#)
The entry-point function called when the thread starts.
- void * [m_pvArg](#)
Pointer to the argument passed into the thread's entypoint.
- uint16_t [m_u16Quantum](#)
Thread quantum (in milliseconds)
- uint16_t [m_u16FlagMask](#)
Event-flag mask.
- [EventFlagOperation_t](#) [m_eFlagMode](#)
Event-flag mode.
- [Timer](#) [m_clTimer](#)
Timer used for blocking-object timeouts.
- bool [m_bExpired](#)
Indicate whether or not a blocking-object timeout has occurred.

Friends

- class [ThreadPort](#)

Additional Inherited Members

17.27.1 Detailed Description

Object providing fundamental multitasking support in the kernel.

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 72 of file [thread.h](#).

17.27.2 Member Function Documentation

17.27.2.1 void Thread::ContextSwitchSWI (void) [static],[private]

ContextSwitchSWI.

This code is used to trigger the context switch interrupt. Called whenever the kernel decides that it is necessary to swap out the current thread for the "next" thread.

Definition at line 414 of file [thread.cpp](#).

17.27.2.2 void Thread::Exit ()

Exit.

Remove the thread from being scheduled again. The thread is effectively destroyed when this occurs. This is extremely useful for cases where a thread encounters an unrecoverable error and needs to be restarted, or in the context of systems where threads need to be created and destroyed dynamically.

This must not be called on the idle thread.

Examples:

[lab9_dynamic_threads/main.cpp](#).

Definition at line 193 of file [thread.cpp](#).

17.27.2.3 uint8_t Thread::GetCurPriority (void) [inline]

GetCurPriority.

Return the priority of the current thread

Returns

Priority of the current thread

Definition at line 196 of file [thread.h](#).

17.27.2.4 ThreadList* Thread::GetCurrent (void) [inline]

GetCurrent.

Return the [ThreadList](#) where the thread is currently located

Returns

Pointer to the thread's current list

Definition at line 177 of file [thread.h](#).

17.27.2.5 `uint16_t Thread::GetEventFlagMask () [inline]`

GetEventFlagMask returns the thread's current event-flag mask, which is used in conjunction with the [EventFlag](#) blocking object type.

Returns

A copy of the thread's event flag mask

Definition at line 348 of file [thread.h](#).

17.27.2.6 `EventFlagOperation_t Thread::GetEventFlagMode () [inline]`

GetEventFlagMode Returns the thread's event flag's operating mode.

Returns

The thread's event flag mode.

Definition at line 367 of file [thread.h](#).

17.27.2.7 `bool Thread::GetExpired ()`

GetExpired.

Return the status of the most-recent blocking call on the thread.

Returns

true - call expired, false - call did not expire

Definition at line 432 of file [thread.cpp](#).

17.27.2.8 `uint8_t Thread::GetID () [inline]`

GetID.

Return the 8-bit ID corresponding to this thread.

Returns

[Thread](#)'s 8-bit ID, set by the user

Definition at line 323 of file [thread.h](#).

17.27.2.9 `ThreadList* Thread::GetOwner (void) [inline]`

GetOwner.

Return the [ThreadList](#) where the thread belongs when it's in the active/ready state in the scheduler.

Returns

Pointer to the [Thread](#)'s owner list

Definition at line 168 of file [thread.h](#).

17.27.2.10 `uint8_t Thread::GetPriority (void) [inline]`

GetPriority.

Return the priority of the current thread

Returns

Priority of the current thread

Definition at line 187 of file [thread.h](#).

17.27.2.11 `uint16_t Thread::GetQuantum (void) [inline]`

GetQuantum.

Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 215 of file [thread.h](#).

17.27.2.12 `uint16_t Thread::GetStackSlack ()`

GetStackSlack.

Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack. If you're having problems with blowing your stack, you can run this function at points in your code during development to see what operations cause problems. Also useful during development as a tool to optimally size thread stacks.

Returns

The amount of slack (unused bytes) on the stack

! ToDo: Take into account stacks that grow up

Definition at line 303 of file [thread.cpp](#).

17.27.2.13 `ThreadState_t Thread::GetState () [inline]`

GetState Returns the current state of the thread to the caller.

Can be used to determine whether or not a thread is ready (or running), stopped, or terminated/exit'd.

Returns

ThreadState_t representing the thread's current state

Examples:

[lab9_dynamic_threads/main.cpp](#).

Definition at line 411 of file [thread.h](#).

17.27.2.14 `void Thread::InheritPriority (uint8_t u8Priority_)`

InheritPriority.

Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions. This should only be called from within the implementation of blocking-objects.

Parameters

<i>u8Priority_</i>	New Priority to boost to.
--------------------	---------------------------

Definition at line 407 of file [thread.cpp](#).

```
17.27.2.15 void Thread::Init ( K_WORD * pwStack_, uint16_t u16StackSize_, uint8_t u8Priority_, ThreadEntry_t
pfEntryPoint_, void * pvArg_ )
```

Init.

Initialize a thread prior to its use. Initialized threads are placed in the stopped state, and are not scheduled until the thread's start method has been invoked first.

Parameters

<i>pwStack_</i>	Pointer to the stack to use for the thread
<i>u16StackSize_</i>	Size of the stack (in bytes)
<i>u8Priority_</i>	Priority of the thread (0 = idle, 7 = max)
<i>pfEntryPoint_</i>	This is the function that gets called when the thread is started
<i>pvArg_</i>	Pointer to the argument passed into the thread's entrypoint function.

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 46 of file [thread.cpp](#).

```
17.27.2.16 void Thread::InitIdle ( void )
```

InitIdle Initialize this [Thread](#) object as the [Kernel](#)'s idle thread.

There should only be one of these, maximum, in a given system.

Definition at line 437 of file [thread.cpp](#).

```
17.27.2.17 void Thread::SetCurrent ( ThreadList * pclNewList_ ) [inline]
```

SetCurrent.

Set the thread's current to the specified thread list

Parameters

<i>pclNewList_</i>	Pointer to the threadlist to apply thread ownership
--------------------	---

Definition at line 225 of file [thread.h](#).

```
17.27.2.18 void Thread::SetEventFlagMask ( uint16_t u16Mask_ ) [inline]
```

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

<i>u16Mask_</i>	
-----------------	--

Definition at line 354 of file [thread.h](#).

17.27.2.19 void Thread::SetEventFlagMode (EventFlagOperation_t *eMode_*) [inline]

SetEventFlagMode Sets the active event flag operation mode.

Parameters

<i>eMode_</i>	Event flag operation mode, defines the logical operator to apply to the event flag.
---------------	---

Definition at line 361 of file [thread.h](#).

17.27.2.20 void Thread::SetExpired (bool *bExpired_*)

SetExpired.

Set the status of the current blocking call on the thread.

Parameters

<i>bExpired_</i>	true - call expired, false - call did not expire
------------------	--

Definition at line 429 of file [thread.cpp](#).

17.27.2.21 void Thread::SetID (uint8_t *u8ID_*) [inline]

SetID.

Set an 8-bit ID to uniquely identify this thread.

Parameters

<i>u8ID_</i>	8-bit Thread ID, set by the user
--------------	--

Definition at line 314 of file [thread.h](#).

17.27.2.22 void Thread::SetOwner (ThreadList * *pclNewList_*) [inline]

SetOwner.

Set the thread's owner to the specified thread list

Parameters

<i>pclNewList_</i>	Pointer to the threadlist to apply thread ownership
--------------------	---

Definition at line 234 of file [thread.h](#).

17.27.2.23 void Thread::SetPriority (uint8_t *u8Priority_*)

SetPriority.

Set the priority of the [Thread](#) (running or otherwise) to a different level. This activity involves re-scheduling, and must be done so with due caution, as it may effect the determinism of the system.

This should *always* be called from within a critical section to prevent system issues.

Parameters

<i>u8Priority_</i>	New priority of the thread
--------------------	----------------------------

Definition at line 363 of file [thread.cpp](#).

17.27.2.24 void Thread::SetPriorityBase (uint8_t *u8Priority_*) [private]

SetPriorityBase.

Parameters

<i>u8Priority_</i>	
--------------------	--

Definition at line 353 of file [thread.cpp](#).

17.27.2.25 void Thread::SetQuantum (uint16_t *u16Quantum_*) [inline]

SetQuantum.

Set the thread's round-robin execution quantum.

Parameters

<i>u16Quantum_</i>	Thread 's execution quantum (in milliseconds)
--------------------	---

Examples:

[lab3_round_robin/main.cpp](#).

Definition at line 206 of file [thread.h](#).

17.27.2.26 void Thread::SetState (ThreadState_t *eState_*) [inline]

SetState Set the thread's state to a new value.

This is only to be used by code within the kernel, and is not intended for use by an end-user.

Parameters

<i>eState_</i>	New thread state to set.
----------------	--------------------------

Definition at line 420 of file [thread.h](#).

17.27.2.27 void Thread::Sleep (uint32_t *u32TimeMs_*) [static]

Sleep.

Put the thread to sleep for the specified time (in milliseconds). Actual time slept may be longer (but not less than) the interval specified.

Parameters

<i>u32TimeMs_</i>	Time to sleep (in ms)
-------------------	-----------------------

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 258 of file [thread.cpp](#).

17.27.2.28 void Thread::Start (void)

Start.

Start the thread - remove it from the stopped list, add it to the scheduler's list of threads (at the thread's set priority), and continue along.

Examples:

[buffalogger/main.cpp](#), [lab10_notifications/main.cpp](#), [lab11_mailboxes/main.cpp](#), [lab1_kernel_setup/main.cpp](#), [lab2_idle_function/main.cpp](#), [lab3_round_robin/main.cpp](#), [lab4_semaphores/main.cpp](#), [lab5_mutexes/main.cpp](#), [lab6_timers/main.cpp](#), [lab7_events/main.cpp](#), [lab8_messages/main.cpp](#), and [lab9_dynamic_threads/main.cpp](#).

Definition at line 115 of file [thread.cpp](#).

17.27.2.29 void Thread::Stop ()

Stop.

Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the [Start\(\)](#) API.

Definition at line 148 of file [thread.cpp](#).

17.27.2.30 void Thread::USleep (uint32_t u32TimeUs_) [static]

USleep.

Put the thread to sleep for the specified time (in microseconds). Actual time slept may be longer (but not less than) the interval specified.

Parameters

<i>u32TimeUs_</i>	Time to sleep (in microseconds)
-------------------	---------------------------------

Definition at line 280 of file [thread.cpp](#).

17.27.2.31 void Thread::Yield (void) [static]

Yield.

Yield the thread - this forces the system to call the scheduler and determine what thread should run next. This is typically used when threads are moved in and out of the scheduler.

Definition at line 324 of file [thread.cpp](#).

The documentation for this class was generated from the following files:

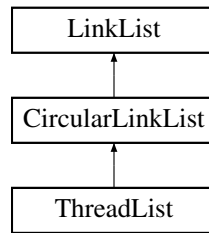
- [/home/mo/mark3-source/embedded/kernel/public/thread.h](#)
- [/home/mo/mark3-source/embedded/kernel/thread.cpp](#)

17.28 ThreadList Class Reference

This class is used for building thread-management facilities, such as schedulers, and blocking objects.

```
#include <threadlist.h>
```

Inheritance diagram for ThreadList:



Public Member Functions

- [ThreadList](#) ()
ThreadList.
- void [SetPriority](#) (uint8_t u8Priority_)
SetPriority.
- void [SetFlagPointer](#) (uint8_t *pu8Flag_)
SetFlagPointer.
- void [Add](#) ([LinkListNode](#) *node_)
Add.
- void [Add](#) ([LinkListNode](#) *node_, uint8_t *pu8Flag_, uint8_t u8Priority_)
Add.
- void [AddPriority](#) ([LinkListNode](#) *node_)
AddPriority.
- void [Remove](#) ([LinkListNode](#) *node_)
Remove.
- [Thread](#) * [HighestWaiter](#) ()
HighestWaiter.

Private Attributes

- uint8_t [m_u8Priority](#)
Priority of the threadlist.
- uint8_t * [m_pu8Flag](#)
Pointer to the bitmap/flag to set when used for scheduling.

Additional Inherited Members

17.28.1 Detailed Description

This class is used for building thread-management facilities, such as schedulers, and blocking objects.

Definition at line 34 of file [threadlist.h](#).

17.28.2 Constructor & Destructor Documentation

17.28.2.1 [ThreadList::ThreadList](#) () [inline]

[ThreadList.](#)

Default constructor - zero-initializes the data.

Definition at line 42 of file [threadlist.h](#).

17.28.3 Member Function Documentation

17.28.3.1 void ThreadList::Add (LinkListNode * *node_*) [virtual]

Add.

Add a thread to the threadlist.

Parameters

<i>node_</i>	Pointer to the thread (link list node) to add to the list
--------------	---

Reimplemented from [CircularLinkList](#).

Definition at line 52 of file [threadlist.cpp](#).

17.28.3.2 void ThreadList::Add (LinkListNode * *node_*, uint8_t * *pu8Flag_*, uint8_t *u8Priority_*)

Add.

Add a thread to the threadlist, specifying the flag and priority at the same time.

Parameters

<i>node_</i>	Pointer to the thread to add (link list node)
<i>pu8Flag_</i>	Pointer to the bitmap flag to set (if used in a scheduler context), or NULL for non-scheduler.
<i>u8Priority_</i>	Priority of the threadlist

Definition at line 104 of file [threadlist.cpp](#).

17.28.3.3 void ThreadList::AddPriority (LinkListNode * *node_*)

AddPriority.

Add a thread to the list such that threads are ordered from highest to lowest priority from the head of the list.

Parameters

<i>node_</i>	Pointer to a thread to add to the list.
--------------	---

Definition at line 65 of file [threadlist.cpp](#).

17.28.3.4 Thread * ThreadList::HighestWaiter ()

HighestWaiter.

Return a pointer to the highest-priority thread in the thread-list.

Returns

Pointer to the highest-priority thread

Definition at line 129 of file [threadlist.cpp](#).

17.28.3.5 void ThreadList::Remove (LinkListNode * *node_*) [virtual]

Remove.

Remove the specified thread from the threadlist

Parameters

<i>node_</i>	Pointer to the thread to remove
--------------	---------------------------------

Reimplemented from [CircularLinkedList](#).

Definition at line 113 of file [threadlist.cpp](#).

17.28.3.6 void ThreadList::SetFlagPointer (uint8_t * *pu8Flag_*)

SetFlagPointer.

Set the pointer to a bitmap to use for this threadlist. Once again, only needed when the threadlist is being used for scheduling purposes.

Parameters

<i>pu8Flag_</i>	Pointer to the bitmap flag
-----------------	----------------------------

Definition at line 46 of file [threadlist.cpp](#).

17.28.3.7 void ThreadList::SetPriority (uint8_t *u8Priority_*)

SetPriority.

Set the priority of this threadlist (if used for a scheduler).

Parameters

<i>u8Priority_</i>	Priority level of the thread list
--------------------	-----------------------------------

Definition at line 40 of file [threadlist.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/threadlist.h](#)
- [/home/mo/mark3-source/embedded/kernel/threadlist.cpp](#)

17.29 ThreadPort Class Reference

Class defining the architecture specific functions required by the kernel.

```
#include <threadport.h>
```

Static Public Member Functions

- static void [StartThreads](#) ()
StartThreads.

Static Private Member Functions

- static void [InitStack](#) ([Thread](#) *pstThread_)
InitStack.

Friends

- class [Thread](#)

17.29.1 Detailed Description

Class defining the architecture specific functions required by the kernel.

This is limited (at this point) to a function to start the scheduler, and a function to initialize the default stack-frame for a thread.

Definition at line 167 of file [threadport.h](#).

17.29.2 Member Function Documentation

17.29.2.1 `void ThreadPort::InitStack (Thread * pstThread_) [static], [private]`

InitStack.

Initialize the thread's stack.

Parameters

<i>pstThread_</i>	Pointer to the thread to initialize
-------------------	-------------------------------------

Definition at line 39 of file [threadport.cpp](#).

17.29.2.2 `void ThreadPort::StartThreads () [static]`

StartThreads.

Function to start the scheduler, initial threads, etc.

Definition at line 135 of file [threadport.cpp](#).

The documentation for this class was generated from the following files:

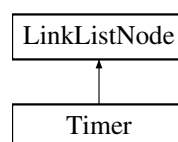
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h](#)
- [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp](#)

17.30 Timer Class Reference

[Timer](#) - an event-driven execution context based on a specified time interval.

```
#include <timer.h>
```

Inheritance diagram for Timer:



Public Member Functions

- [Timer](#) ()
Timer.
- void [Init](#) ()
Init.
- void [Start](#) (bool bRepeat_, uint32_t u32IntervalMs_, [TimerCallback_t](#) pfCallback_, void *pvData_)
Start.

- void [Start](#) (bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_, [TimerCallback_t](#) pf←
Callback_, void *pvData_)
Start.
- void [Stop](#) ()
Stop.
- void [SetFlags](#) (uint8_t u8Flags_)
SetFlags.
- void [SetCallback](#) ([TimerCallback_t](#) pfCallback_)
SetCallback.
- void [SetData](#) (void *pvData_)
SetData.
- void [SetOwner](#) ([Thread](#) *pclOwner_)
SetOwner.
- void [SetIntervalTicks](#) (uint32_t u32Ticks_)
SetIntervalTicks.
- void [SetIntervalSeconds](#) (uint32_t u32Seconds_)
SetIntervalSeconds.
- uint32_t [GetInterval](#) ()
GetInterval.
- void [SetIntervalMSeconds](#) (uint32_t u32MSeconds_)
SetIntervalMSeconds.
- void [SetIntervalUSeconds](#) (uint32_t u32USeconds_)
SetIntervalUSeconds.
- void [SetTolerance](#) (uint32_t u32Ticks_)
SetTolerance.

Private Attributes

- uint8_t [m_u8Flags](#)
Flags for the timer, defining if the timer is one-shot or repeated.
- [TimerCallback_t](#) [m_pfCallback](#)
Pointer to the callback function.
- uint32_t [m_u32Interval](#)
Interval of the timer in timer ticks.
- uint32_t [m_u32TimeLeft](#)
Time remaining on the timer.
- uint32_t [m_u32TimerTolerance](#)
Maximum tolerance (usedd for timer harmonization)
- [Thread](#) * [m_pclOwner](#)
Pointer to the owner thread.
- void * [m_pvData](#)
Pointer to the callback data.

Friends

- class [TimerList](#)

Additional Inherited Members

17.30.1 Detailed Description

[Timer](#) - an event-driven execution context based on a specified time interval.

This inherits from a [LinkedListNode](#) for ease of management by a global [TimerList](#) object.

Examples:

[lab6_timers/main.cpp](#).

Definition at line 102 of file [timer.h](#).

17.30.2 Constructor & Destructor Documentation

17.30.2.1 `Timer::Timer() [inline]`

[Timer](#).

Default Constructor - zero-initializes all internal data.

Definition at line 110 of file [timer.h](#).

17.30.3 Member Function Documentation

17.30.3.1 `uint32_t Timer::GetInterval() [inline]`

GetInterval.

Returns

Definition at line 217 of file [timer.h](#).

17.30.3.2 `void Timer::Init(void) [inline]`

Init.

Re-initialize the [Timer](#) to default values.

Definition at line 117 of file [timer.h](#).

17.30.3.3 `void Timer::SetCallback(TimerCallback_t pfCallback_) [inline]`

SetCallback.

Define the callback function to be executed on expiry of the timer

Parameters

<i>pfCallback_</i>	Pointer to the callback function to call
--------------------	--

Definition at line 173 of file [timer.h](#).

17.30.3.4 `void Timer::SetData(void * pvData_) [inline]`

SetData.

Define a pointer to be sent to the timer callbacak on timer expiry

Parameters

<i>pvData_</i>	Pointer to data to pass as argument into the callback
----------------	---

Definition at line 182 of file [timer.h](#).

17.30.3.5 void Timer::SetFlags (uint8_t *u8Flags_*) [inline]

SetFlags.

Set the timer's flags based on the bits in the *u8Flags_* argument

Parameters

<i>u8Flags_</i>	Flags to assign to the timer object. TIMERLIST_FLAG_ONE_SHOT for a one-shot timer, 0 for a continuous timer.
-----------------	--

Definition at line 164 of file [timer.h](#).

17.30.3.6 void Timer::SetIntervalMSeconds (uint32_t *u32MSeconds_*)

SetIntervalMSeconds.

Set the timer expiry interval in milliseconds (platform agnostic)

Parameters

<i>u32MSeconds_</i>	Time in milliseconds
---------------------	----------------------

Definition at line 94 of file [timer.cpp](#).

17.30.3.7 void Timer::SetIntervalSeconds (uint32_t *u32Seconds_*)

SetIntervalSeconds.

! The next three cost u16 330 bytes of flash on AVR...

Set the timer expiry interval in seconds (platform agnostic)

Parameters

<i>u32Seconds_</i>	Time in seconds
--------------------	-----------------

Definition at line 88 of file [timer.cpp](#).

17.30.3.8 void Timer::SetIntervalTicks (uint32_t *u32Ticks_*)

SetIntervalTicks.

Set the timer expiry in system-ticks (platform specific!)

Parameters

<i>u32Ticks_</i>	Time in ticks
------------------	---------------

Definition at line 80 of file [timer.cpp](#).

17.30.3.9 void Timer::SetIntervalUSeconds (uint32_t *u32USeconds_*)

SetIntervalUSeconds.

Set the timer expiry interval in microseconds (platform agnostic)

Parameters

<i>u32USeconds_</i> ↔ —	Time in microseconds
----------------------------	----------------------

Definition at line 100 of file [timer.cpp](#).

17.30.3.10 void Timer::SetOwner (Thread * *pclOwner_*) [inline]

SetOwner.

Set the owner-thread of this timer object (all timers must be owned by a thread).

Parameters

<i>pclOwner_</i>	Owner thread of this timer object
------------------	-----------------------------------

Definition at line 192 of file [timer.h](#).

17.30.3.11 void Timer::SetTolerance (uint32_t *u32Ticks_*)

SetTolerance.

Set the timer's maximum tolerance in order to synchronize timer processing with other timers in the system.

Parameters

<i>u32Ticks_</i>	Maximum tolerance in ticks
------------------	----------------------------

Definition at line 106 of file [timer.cpp](#).

17.30.3.12 void Timer::Start (bool *bRepeat_*, uint32_t *u32IntervalMs_*, TimerCallback_t *pfCallback_*, void * *pvData_*)

Start.

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

<i>bRepeat_</i>	0 - timer is one-shot. 1 - timer is repeating.
<i>u32IntervalMs_</i> ↔ —	- Interval of the timer in milliseconds
<i>pfCallback_</i>	- Function to call on timer expiry
<i>pvData_</i>	- Data to pass into the callback function

Examples:

[lab6_timers/main.cpp](#).

Definition at line 48 of file [timer.cpp](#).

17.30.3.13 void Timer::Start (bool *bRepeat_*, uint32_t *u32IntervalMs_*, uint32_t *u32ToleranceMs_*, TimerCallback_t *pfCallback_*, void * *pvData_*)

Start.

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

<i>bRepeat_</i>	0 - timer is one-shot. 1 - timer is repeating.
<i>u32IntervalMs_</i>	- Interval of the timer in milliseconds
<i>u32ToleranceMs_</i>	- Allow the timer expiry to be delayed by an additional maximum time, in order to have as many timers expire at the same time as possible.
<i>pfCallback_</i>	- Function to call on timer expiry
<i>pvData_</i>	- Data to pass into the callback function

Definition at line 67 of file [timer.cpp](#).

17.30.3.14 void Timer::Stop ()

Stop.

Stop a timer already in progress. Has no effect on timers that have already been stopped.

Definition at line 74 of file [timer.cpp](#).

The documentation for this class was generated from the following files:

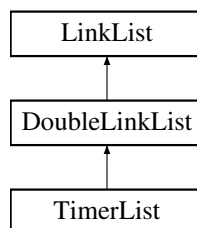
- [/home/mo/mark3-source/embedded/kernel/public/timer.h](#)
- [/home/mo/mark3-source/embedded/kernel/timer.cpp](#)

17.31 TimerList Class Reference

[TimerList](#) class - a doubly-linked-list of timer objects.

```
#include <timerlist.h>
```

Inheritance diagram for TimerList:



Public Member Functions

- void [Init](#) ()
Init.
- void [Add](#) ([Timer](#) *pclListNode_)
Add.
- void [Remove](#) ([Timer](#) *pclListNode_)
Remove.
- void [Process](#) ()
Process.

Private Attributes

- uint32_t [m_u32NextWakeup](#)

The time (in system clock ticks) of the next wakeup event.

- bool [m_bTimerActive](#)

Whether or not the timer is active.

Additional Inherited Members

17.31.1 Detailed Description

[TimerList](#) class - a doubly-linked-list of timer objects.

Definition at line 37 of file [timerlist.h](#).

17.31.2 Member Function Documentation

17.31.2.1 void TimerList::Add (Timer * *pclListNode_*)

Add.

Add a timer to the [TimerList](#).

Parameters

<i>pclListNode_</i>	Pointer to the Timer to Add
---------------------	---

Definition at line 56 of file [timerlist.cpp](#).

17.31.2.2 void TimerList::Init (void)

Init.

Initialize the [TimerList](#) object. Must be called before using the object.

Definition at line 49 of file [timerlist.cpp](#).

17.31.2.3 void TimerList::Process (void)

Process.

Process all timers in the timerlist as a result of the timer expiring. This will select a new timer epoch based on the next timer to expire. ToDo - figure out if we need to deal with any overtime here.

Definition at line 121 of file [timerlist.cpp](#).

17.31.2.4 void TimerList::Remove (Timer * *pclListNode_*)

Remove.

Remove a timer from the [TimerList](#), cancelling its expiry.

Parameters

<i>pclListNode_</i>	Pointer to the Timer to remove
---------------------	--

Definition at line 104 of file [timerlist.cpp](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/timerlist.h](#)
- [/home/mo/mark3-source/embedded/kernel/timerlist.cpp](#)

17.32 TimerScheduler Class Reference

"Static" Class used to interface a global [TimerList](#) with the rest of the kernel.

```
#include <timerscheduler.h>
```

Static Public Member Functions

- static void [Init](#) ()
Init.
- static void [Add](#) ([Timer](#) *pclListNode_)
Add.
- static void [Remove](#) ([Timer](#) *pclListNode_)
Remove.
- static void [Process](#) ()
Process.

Static Private Attributes

- static [TimerList](#) m_clTimerList
TimerList object manipulated by the Timer Scheduler.

17.32.1 Detailed Description

"Static" Class used to interface a global [TimerList](#) with the rest of the kernel.

Definition at line 38 of file [timerscheduler.h](#).

17.32.2 Member Function Documentation

17.32.2.1 static void [TimerScheduler::Add](#) ([Timer](#) * [pclListNode_](#)) [inline],[static]

Add.

Add a timer to the timer scheduler. Adding a timer implicitly starts the timer as well.

Parameters

pclListNode_	Pointer to the timer list node to add
------------------------------	---------------------------------------

Definition at line 57 of file [timerscheduler.h](#).

17.32.2.2 static void [TimerScheduler::Init](#) (void) [inline],[static]

Init.

Initialize the timer scheduler. Must be called before any timer, or timer-derived functions are used.

Definition at line 47 of file [timerscheduler.h](#).

17.32.2.3 static void [TimerScheduler::Process](#) (void) [inline],[static]

Process.

This function must be called on timer expiry (from the timer's ISR context). This will result in all timers being updated based on the epoch that just elapsed. The next timer epoch is set based on the next [Timer](#) object to expire.

Definition at line 79 of file [timerscheduler.h](#).

17.32.2.4 static void TimerScheduler::Remove (Timer * *pcListNode_*) [inline],[static]

Remove.

Remove a timer from the timer scheduler. May implicitly stop the timer if this is the only active timer scheduled.

Parameters

<i>pcListNode_</i>	Pointer to the timer list node to remove
--------------------	--

Definition at line 68 of file [timerscheduler.h](#).

The documentation for this class was generated from the following files:

- [/home/mo/mark3-source/embedded/kernel/public/timerscheduler.h](#)
- [/home/mo/mark3-source/embedded/kernel/timerlist.cpp](#)

File Documentation

[illegible]

```

00038
00039 //-----
00040 uint8_t Atomic::Set( uint8_t *pu8Source_, uint8_t u8Val_ )
00041 {
00042     uint8_t u8Ret;
00043     CS_ENTER();
00044     u8Ret = *pu8Source_;
00045     *pu8Source_ = u8Val_;
00046     CS_EXIT();
00047     return u8Ret;
00048 }
00049 //-----
00050 uint16_t Atomic::Set( uint16_t *pu16Source_, uint16_t u16Val_ )
00051 {
00052     uint16_t u16Ret;
00053     CS_ENTER();
00054     u16Ret = *pu16Source_;
00055     *pu16Source_ = u16Val_;
00056     CS_EXIT();
00057     return u16Ret;
00058 }
00059 //-----
00060 uint32_t Atomic::Set( uint32_t *pu32Source_, uint32_t u32Val_ )
00061 {
00062     uint32_t u32Ret;
00063     CS_ENTER();
00064     u32Ret = *pu32Source_;
00065     *pu32Source_ = u32Val_;
00066     CS_EXIT();
00067     return u32Ret;
00068 }
00069
00070 //-----
00071 uint8_t Atomic::Add( uint8_t *pu8Source_, uint8_t u8Val_ )
00072 {
00073     uint8_t u8Ret;
00074     CS_ENTER();
00075     u8Ret = *pu8Source_;
00076     *pu8Source_ += u8Val_;
00077     CS_EXIT();
00078     return u8Ret;
00079 }
00080
00081 //-----
00082 uint16_t Atomic::Add( uint16_t *pu16Source_, uint16_t u16Val_ )
00083 {
00084     uint16_t u16Ret;
00085     CS_ENTER();
00086     u16Ret = *pu16Source_;
00087     *pu16Source_ += u16Val_;
00088     CS_EXIT();
00089     return u16Ret;
00090 }
00091
00092 //-----
00093 uint32_t Atomic::Add( uint32_t *pu32Source_, uint32_t u32Val_ )
00094 {
00095     uint32_t u32Ret;
00096     CS_ENTER();
00097     u32Ret = *pu32Source_;
00098     *pu32Source_ += u32Val_;
00099     CS_EXIT();
00100     return u32Ret;
00101 }
00102
00103 //-----
00104 uint8_t Atomic::Sub( uint8_t *pu8Source_, uint8_t u8Val_ )
00105 {
00106     uint8_t u8Ret;
00107     CS_ENTER();
00108     u8Ret = *pu8Source_;
00109     *pu8Source_ -= u8Val_;
00110     CS_EXIT();
00111     return u8Ret;
00112 }
00113
00114 //-----
00115 uint16_t Atomic::Sub( uint16_t *pu16Source_, uint16_t u16Val_ )
00116 {
00117     uint16_t u16Ret;
00118     CS_ENTER();
00119     u16Ret = *pu16Source_;
00120     *pu16Source_ -= u16Val_;
00121     CS_EXIT();
00122     return u16Ret;
00123 }
00124

```

```

00125 //-----
00126 uint32_t Atomic::Sub( uint32_t *pu32Source_, uint32_t u32Val_ )
00127 {
00128     uint32_t u32Ret;
00129     CS_ENTER();
00130     u32Ret = *pu32Source_;
00131     *pu32Source_ -= u32Val_;
00132     CS_EXIT();
00133     return u32Ret;
00134 }
00135
00136 //-----
00137 bool Atomic::TestAndSet( bool *pbLock_ )
00138 {
00139     uint8_t u8Ret;
00140     CS_ENTER();
00141     u8Ret = *pbLock_;
00142     if (!u8Ret)
00143     {
00144         *pbLock_ = 1;
00145     }
00146     CS_EXIT();
00147     return u8Ret;
00148 }
00149
00150 #endif // KERNEL_USE_ATOMIC

```

18.3 /home/mo/mark3-source/embedded/kernel/autoalloc.cpp File Reference

Automatic memory allocation for kernel objects.

```
#include "mark3cfg.h"
#include "autoalloc.h"
#include "threadport.h"
#include "kernel.h"
```

18.3.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file [autoalloc.cpp](#).

18.4 autoalloc.cpp

```

00001 /*-----
00002
00003 |_____|_____|_____|_____|_____|_____|_____|_____|
00004 |      \      /      \      /      \      /      \      /      \
00005 |_____|_____|_____|_____|_____|_____|_____|_____|
00006 |      /      \      /      \      /      \      /      \      /
00007 |_____|_____|_____|_____|_____|_____|_____|_____|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00020 #include "mark3cfg.h"
00021 #include "autoalloc.h"
00022 #include "threadport.h"
00023 #include "kernel.h"
00024
00025 #if KERNEL_USE_AUTO_ALLOC
00026
00027 // Align to nearest word boundary
00028 #define ALLOC_ALIGN(x)  ( ((x) + (sizeof(K_ADDR)-1)) & (sizeof(K_ADDR) - 1) )
00029
00030 //-----
00031 uint8_t AutoAlloc::m_au8AutoHeap[ AUTO_ALLOC_SIZE ];
00032 K_ADDR AutoAlloc::m_aHeapTop;
00033
00034 //-----

```

```

00035 void AutoAlloc::Init(void)
00036 {
00037     m_aHeapTop = (K_ADDR) (m_au8AutoHeap);
00038 }
00039
00040 //-----
00041 void *AutoAlloc::Allocate( uint16_t u16Size_ )
00042 {
00043     void *pvRet = 0;
00044
00045     CS_ENTER();
00046     uint16_t u16AllocSize = ALLOC_ALIGN(u16Size_);
00047     if (((K_ADDR)m_aHeapTop - (K_ADDR)&m_au8AutoHeap[0]) + u16AllocSize) < AUTO_ALLOC_SIZE)
00048     {
00049         pvRet = (void*)m_aHeapTop;
00050         m_aHeapTop += u16AllocSize;
00051     }
00052     CS_EXIT();
00053
00054     if (!pvRet)
00055     {
00056         Kernel::Panic(PANIC_AUTO_HEAP_EXHUSTED);
00057     }
00058
00059     return pvRet;
00060 }
00061
00062 #endif

```

18.5 /home/mo/mark3-source/embedded/kernel/blocking.cpp File Reference

Implementation of base class for blocking objects.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "thread.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

18.5.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file [blocking.cpp](#).

18.6 blocking.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023
00024 #include "blocking.h"
00025 #include "thread.h"
00026
00027 #define _CAN_HAS_DEBUG
00028 //--[Autogenerated - Do Not Modify]-----

```

```

00029 #include "dbg_file_list.h"
00030 #include "buffalogger.h"
00031 #if defined(DBG_FILE)
00032 # error "Debug logging file token already defined! Bailing."
00033 #else
00034 # define DBG_FILE _DBG__KERNEL_BLOCKING_CPP
00035 #endif
00036 //--[End Autogenerated content]-----
00037 #include "kerneldebug.h"
00038
00039 #if KERNEL_USE_SEMAPHORE || KERNEL_USE_MUTEX
00040 //-----
00041 void BlockingObject::Block(Thread *pclThread_)
00042 {
00043     KERNEL_ASSERT( pclThread_ );
00044     KERNEL_TRACE_1( "Blocking Thread %d", (uint16_t)pclThread_>
GetID() );
00045
00046     // Remove the thread from its current thread list (the "owner" list)
00047     // ... And add the thread to this object's block list
00048     Scheduler::Remove(pclThread_);
00049     m_clBlockList.Add(pclThread_);
00050
00051     // Set the "current" list location to the blocklist for this thread
00052     pclThread_>SetCurrent(&m_clBlockList);
00053     pclThread_>SetState(THREAD_STATE_BLOCKED);
00054 }
00055
00056 //-----
00057 void BlockingObject::BlockPriority(Thread *pclThread_)
00058 {
00059     KERNEL_ASSERT( pclThread_ );
00060     KERNEL_TRACE_1( "Blocking Thread %d", (uint16_t)pclThread_>
GetID() );
00061
00062     // Remove the thread from its current thread list (the "owner" list)
00063     // ... And add the thread to this object's block list
00064     Scheduler::Remove(pclThread_);
00065     m_clBlockList.AddPriority(pclThread_);
00066
00067     // Set the "current" list location to the blocklist for this thread
00068     pclThread_>SetCurrent(&m_clBlockList);
00069     pclThread_>SetState(THREAD_STATE_BLOCKED);
00070 }
00071
00072 //-----
00073 void BlockingObject::UnBlock(Thread *pclThread_)
00074 {
00075     KERNEL_ASSERT( pclThread_ );
00076     KERNEL_TRACE_1( "Unblocking Thread %d", (uint16_t)pclThread_>
GetID() );
00077
00078     // Remove the thread from its current thread list (the "owner" list)
00079     pclThread_>GetCurrent()->Remove(pclThread_);
00080
00081     // Put the thread back in its active owner's list. This is usually
00082     // the ready-queue at the thread's original priority.
00083     Scheduler::Add(pclThread_);
00084
00085     // Tag the thread's current list location to its owner
00086     pclThread_>SetCurrent(pclThread_>GetOwner());
00087     pclThread_>SetState(THREAD_STATE_READY);
00088 }
00089
00090 #endif

```

18.7 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp File Reference

ATMega328p Profiling timer implementation.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kernelprofile.h"
#include "threadport.h"
#include <avr/io.h>
#include <avr/interrupt.h>

```

18.7.1 Detailed Description

ATMega328p Profiling timer implementation.

Definition in file [kernelprofile.cpp](#).

18.8 kernelprofile.cpp

```

00001  /*=====
00002
00003  |-----|-----|-----|-----|-----|
00004  | \    / | \    / | \    / | \    / | \    / |
00005  |  \  /  |  \  /  |  \  /  |  \  /  |  \  /  |
00006  |   \/   |   \/   |   \/   |   \/   |   \/   |
00007  |_____|   |_____|   |_____|   |_____|   |
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00020  #include "kerneltypes.h"
00021  #include "mark3cfg.h"
00022  #include "profile.h"
00023  #include "kernelprofile.h"
00024  #include "threadport.h"
00025  #include <avr/io.h>
00026  #include <avr/interrupt.h>
00027
00028  #if KERNEL_USE_PROFILER
00029  uint32_t Profiler::m_u32Epoch;
00030
00031  //-----
00032  void Profiler::Init()
00033  {
00034      TCCR0A = 0;
00035      TCCR0B = 0;
00036      TIFR0 = 0;
00037      TIMSK0 = 0;
00038      m_u32Epoch = 0;
00039  }
00040
00041  //-----
00042  void Profiler::Start()
00043  {
00044      TIFR0 = 0;
00045      TCNT0 = 0;
00046      TCCR0B |= (1 << CS01);
00047      TIMSK0 |= (1 << TOIE0);
00048  }
00049
00050  //-----
00051  void Profiler::Stop()
00052  {
00053      TIFR0 = 0;
00054      TCCR0B &= ~(1 << CS01);
00055      TIMSK0 &= ~(1 << TOIE0);
00056  }
00057  //-----
00058  uint16_t Profiler::Read()
00059  {
00060      uint16_t u16Ret;
00061      CS_ENTER();
00062      TCCR0B &= ~(1 << CS01);
00063      u16Ret = TCNT0;
00064      TCCR0B |= (1 << CS01);
00065      CS_EXIT();
00066      return u16Ret;
00067  }
00068
00069  //-----
00070  void Profiler::Process()
00071  {
00072      CS_ENTER();
00073      m_u32Epoch++;
00074      CS_EXIT();
00075  }

```



```

00076
00077 //-----
00078 ISR(TIMER0_OVF_vect)
00079 {
00080     Profiler::Process();
00081 }
00082
00083 #endif

```

18.9 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp File Reference

Kernel Software interrupt implementation for ATmega328p.

```

#include "kerneltypes.h"
#include "kernelswi.h"
#include <avr/io.h>
#include <avr/interrupt.h>

```

18.9.1 Detailed Description

Kernel Software interrupt implementation for ATmega328p.

Definition in file [kernelswi.cpp](#).

18.10 kernelswi.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "kernelswi.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 //-----
00029 void KernelSWI::Config(void)
00030 {
00031     PORTD &= ~0x04; // Clear INT0
00032     DDRD |= 0x04;    // Set PortD, bit 2 (INT0) As Output
00033     EICRA |= (1 << ISC00) | (1 << ISC01); // Rising edge on INT0
00034 }
00035
00036 //-----
00037 void KernelSWI::Start(void)
00038 {
00039     EIFR &= ~(1 << INTF0); // Clear any pending interrupts on INT0
00040     EIMSK |= (1 << INT0);  // Enable INT0 interrupt (as int32_t as I-bit is set)
00041 }
00042
00043 //-----
00044 void KernelSWI::Stop(void)
00045 {
00046     EIMSK &= ~(1 << INT0); // Disable INT0 interrupts
00047 }
00048
00049 //-----
00050 uint8_t KernelSWI::DI()
00051 {

```

```

00052     bool bEnabled = ((EIMSK & (1 << INT0)) != 0);
00053     EIMSK &= ~(1 << INT0);
00054     return bEnabled;
00055 }
00056
00057 //-----
00058 void KernelSWI::RI(bool bEnable_)
00059 {
00060     if (bEnable_)
00061     {
00062         EIMSK |= (1 << INT0);
00063     }
00064     else
00065     {
00066         EIMSK &= ~(1 << INT0);
00067     }
00068 }
00069
00070 //-----
00071 void KernelSWI::Clear(void)
00072 {
00073     EIFR &= ~(1 << INTF0);    // Clear the interrupt flag for INT0
00074 }
00075
00076 //-----
00077 void KernelSWI::Trigger(void)
00078 {
00079     //if(Thread_IsSchedulerEnabled())
00080     {
00081         PORTD &= ~0x04;
00082         PORTD |= 0x04;
00083     }
00084 }

```

18.11 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp

File Reference

[Kernel Timer](#) Implementation for ATmega328p.

```

#include "kerneltypes.h"
#include "kerneltimer.h"
#include "mark3cfg.h"
#include <avr/io.h>
#include <avr/interrupt.h>

```

18.11.1 Detailed Description

[Kernel Timer](#) Implementation for ATmega328p.

Definition in file [kerneltimer.cpp](#).

18.12 kerneltimer.cpp

```

00001 /*=====
00002
00003  _____  _____  _____  _____  _____
00004  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /
00005  |    \ /    |    \ /    |    \ /    |    \ /    |    \ /
00006  |_____/_____|_____/_____|_____/_____|_____/_____|_____/
00007
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00021 #include "kerneltypes.h"
00022 #include "kerneltimer.h"
00023 #include "mark3cfg.h"
00024

```

```

00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 #define TCCR1B_INIT      ((1 << WGM12) | (1 << CS12))
00029 #define TIMER_IMSK      (1 << OCIE1A)
00030 #define TIMER_IFR       (1 << OCF1A)
00031
00032 //-----
00033 void KernelTimer::Config(void)
00034 {
00035     TCCR1B = TCCR1B_INIT;
00036 }
00037
00038 //-----
00039 void KernelTimer::Start(void)
00040 {
00041     #if !KERNEL_TIMERS_TICKLESS
00042         TCCR1B = ((1 << WGM12) | (1 << CS11) | (1 << CS10));
00043         OCR1A = ((SYSTEM_FREQ / 1000) / 64);
00044     #else
00045         TCCR1B |= (1 << CS12);
00046     #endif
00047     TCNT1 = 0;
00048     TIFR1 &= ~TIMER_IFR;
00049     TIMSK1 |= TIMER_IMSK;
00050 }
00051
00052 //-----
00053 void KernelTimer::Stop(void)
00054 {
00055     #if KERNEL_TIMERS_TICKLESS
00056         TIFR1 &= ~TIMER_IFR;
00057         TIMSK1 &= ~TIMER_IMSK;
00058         TCCR1B &= ~(1 << CS12);    // Disable count...
00059         TCNT1 = 0;
00060         OCR1A = 0;
00061     #endif
00062 }
00063
00064 //-----
00065 uint16_t KernelTimer::Read(void)
00066 {
00067     #if KERNEL_TIMERS_TICKLESS
00068         volatile uint16_t u16Read1;
00069         volatile uint16_t u16Read2;
00070
00071         do {
00072             u16Read1 = TCNT1;
00073             u16Read2 = TCNT1;
00074         } while (u16Read1 != u16Read2);
00075
00076         return u16Read1;
00077     #else
00078         return 0;
00079     #endif
00080 }
00081
00082 //-----
00083 uint32_t KernelTimer::SubtractExpiry(uint32_t u32Interval_)
00084 {
00085     #if KERNEL_TIMERS_TICKLESS
00086         OCR1A -= (uint16_t)u32Interval_;
00087         return (uint32_t)OCR1A;
00088     #else
00089         return 0;
00090     #endif
00091 }
00092
00093 //-----
00094 uint32_t KernelTimer::TimeToExpiry(void)
00095 {
00096     #if KERNEL_TIMERS_TICKLESS
00097         uint16_t u16Read = KernelTimer::Read();
00098         uint16_t u16OCR1A = OCR1A;
00099
00100         if (u16Read >= u16OCR1A)
00101         {
00102             return 0;
00103         }
00104         else
00105         {
00106             return (uint32_t)(u16OCR1A - u16Read);
00107         }
00108     #else
00109         return 0;
00110     #endif
00111 }

```

```

00112 }
00113
00114 //-----
00115 uint32_t KernelTimer::GetOvertime(void)
00116 {
00117     return KernelTimer::Read();
00118 }
00119
00120 //-----
00121 uint32_t KernelTimer::SetExpiry(uint32_t u32Interval_)
00122 {
00123     #if KERNEL_TIMERS_TICKLESS
00124         uint16_t u16SetInterval;
00125         if (u32Interval_ > 65535)
00126         {
00127             u16SetInterval = 65535;
00128         }
00129         else
00130         {
00131             u16SetInterval = (uint16_t)u32Interval_ ;
00132         }
00133         OCR1A = u16SetInterval;
00134         return (uint32_t)u16SetInterval;
00135     #else
00136         return 0;
00137     #endif
00138 }
00139
00140 //-----
00141 void KernelTimer::ClearExpiry(void)
00142 {
00143     #if KERNEL_TIMERS_TICKLESS
00144         OCR1A = 65535; // Clear the compare value
00145     #endif
00146 }
00147
00148 //-----
00149 uint8_t KernelTimer::DI(void)
00150 {
00151     #if KERNEL_TIMERS_TICKLESS
00152         bool bEnabled = ((TIMSK1 & (TIMER_IMSK)) != 0);
00153         TIFR1 &= ~TIMER_IFR; // Clear interrupt flags
00154         TIMSK1 &= ~TIMER_IMSK; // Disable interrupt
00155         return bEnabled;
00156     #else
00157         return 0;
00158     #endif
00159 }
00160
00161 //-----
00162 void KernelTimer::EI(void)
00163 {
00164     KernelTimer::RI(0);
00165 }
00166
00167 //-----
00168 void KernelTimer::RI(bool bEnable_)
00169 {
00170     #if KERNEL_TIMERS_TICKLESS
00171         if (bEnable_)
00172         {
00173             TIMSK1 |= (1 << OCIE1A); // Enable interrupt
00174         }
00175         else
00176         {
00177             TIMSK1 &= ~(1 << OCIE1A);
00178         }
00179     #endif
00180 }
00181

```

18.13 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h File Reference

Profiling timer hardware interface.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"

```

Classes

- class Profiler

System profiling timer interface.

18.13.1 Detailed Description

Profiling timer hardware interface.

Definition in file [kernelprofile.h](#).

18.14 kernelprofile.h

```

00001 /*-----
00002
00003 |-----|-----|-----|-----|-----|-----|
00004 |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00005 |  / \  / \  |  / \  / \  |  / \  / \  |  / \  / \  |  / \  / \  |
00006 | /   \ /   \ | /   \ /   \ | /   \ /   \ | /   \ /   \ | /   \ /   \ |
00007 |-----|-----|-----|-----|-----|-----|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022 #include "ll.h"
00023
00024 #ifndef __KPROFILE_H__
00025 #define __KPROFILE_H__
00026
00027 #if KERNEL_USE_PROFILER
00028
00029 //-----
00030 #define TICKS_PER_OVERFLOW          (256)
00031 #define CLOCK_DIVIDE                (8)
00032
00033 //-----
00037 class Profiler
00038 {
00039 public:
00046     static void Init();
00047
00053     static void Start();
00054
00060     static void Stop();
00061
00067     static uint16_t Read();
00068
00074     static void Process();
00075
00081     static uint32_t GetEpoch(){ return m_u32Epoch; }
00082 private:
00083
00084     static uint32_t m_u32Epoch;
00085 };
00086
00087 #endif //KERNEL_USE_PROFILER
00088
00089 #endif
00090

```

18.15 [/home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h](#)
File Reference

Kernel Software interrupt declarations.

```
#include "kerneltypes.h"
```


18.17.1 Detailed Description

Kernel Timer Class declaration.

Definition in file [kerneltimer.h](#).

18.18 kerneltimer.h

```

00001  /*=====
00002
00003  _____
00004  /         \
00005  /         \
00006  /         \
00007  /         \
00008  /         \
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  ===== */
00021  #include "kerneltypes.h"
00022  #ifndef __KERNELTIMER_H_
00023  #define __KERNELTIMER_H_
00024
00025  //-----
00026  #define SYSTEM_FREQ      ((uint32_t)16000000)
00027  #define TIMER_FREQ      ((uint32_t)(SYSTEM_FREQ / 256)) // Timer ticks per second...
00028
00029  //-----
00033  class KernelTimer
00034  {
00035  public:
00041      static void Config(void);
00042
00048      static void Start(void);
00049
00055      static void Stop(void);
00056
00062      static uint8_t DI(void);
00063
00071      static void RI(bool bEnable_);
00072
00078      static void EI(void);
00079
00090      static uint32_t SubtractExpiry(uint32_t u32Interval_);
00091
00100      static uint32_t TimeToExpiry(void);
00101
00110      static uint32_t SetExpiry(uint32_t u32Interval_);
00111
00120      static uint32_t GetOvertime(void);
00121
00127      static void ClearExpiry(void);
00128
00129  private:
00137      static uint16_t Read(void);
00138
00139  };
00140
00141  #endif //__KERNELTIMER_H_

```

18.19 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h

File Reference

ATMega328p Multithreading support.

```

#include "kerneltypes.h"
#include "thread.h"
#include <avr/io.h>
#include <avr/interrupt.h>

```

Classes

- class [ThreadPort](#)

Class defining the architecture specific functions required by the kernel.

Macros

- #define [ASM](#)(x) asm volatile(x);
ASM Macro - simplify the use of ASM directive in C.
- #define [SR_](#) 0x3F
Status register define - map to 0x003F.
- #define [SPH_](#) 0x3E
Stack pointer define.
- #define [TOP_OF_STACK](#)(x, y) (uint8_t*) (((uint16_t)x) + (y-1))
Macro to find the top of a stack given its size and top address.
- #define [PUSH_TO_STACK](#)(x, y) *x = y; x--;
Push a value y to the stack pointer x and decrement the stack pointer.
- #define [Thread_SaveContext](#)()
Save the context of the [Thread](#).
- #define [Thread_RestoreContext](#)()
Restore the context of the [Thread](#).
- #define [CS_ENTER](#)()
These macros must be used in pairs !
- #define [CS_EXIT](#)()
Exit critical section (restore status register)
- #define [ENABLE_INTS](#)() [ASM](#)("sei");
Initiate a contex switch without using the SWI.

18.19.1 Detailed Description

ATMega328p Multithreading support.

Definition in file [threadport.h](#).

18.19.2 Macro Definition Documentation

18.19.2.1 #define CS_ENTER()

Value:

```
{ \
volatile uint8_t x; \
x = _SFR_IO8(SR_); \
ASM("cli");
```

These macros *must* be used in pairs !

Enter critical section (copy status register, disable interrupts)

Examples:

[buffalogger/main.cpp](#).

Definition at line [142](#) of file [threadport.h](#).

18.20 threadport.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00021 #ifndef __THREADPORT_H_
00022 #define __THREADPORT_H_
00023
00024 #include "kerneltypes.h"
00025 #include "thread.h"
00026
00027 #include <avr/io.h>
00028 #include <avr/interrupt.h>
00029
00030 //-----
00032 #define ASM(x)      asm volatile(x);
00033 #define SR_        0x3F
00035 #define SPH_        0x3E
00037 #define SPL_        0x3D
00038
00039
00040 //-----
00042 #define TOP_OF_STACK(x, y)      (uint8_t*) ( ((uint16_t)x) + (y-1) )
00043 #define PUSH_TO_STACK(x, y)    *x = y; x--;
00044
00045
00046 //-----
00048 #define Thread_SaveContext() \
00049 ASM("push r0"); \
00050 ASM("in r0, __SREG__"); \
00051 ASM("cli"); \
00052 ASM("push r0"); \
00053 ASM("push r1"); \
00054 ASM("clr r1"); \
00055 ASM("push r2"); \
00056 ASM("push r3"); \
00057 ASM("push r4"); \
00058 ASM("push r5"); \
00059 ASM("push r6"); \
00060 ASM("push r7"); \
00061 ASM("push r8"); \
00062 ASM("push r9"); \
00063 ASM("push r10"); \
00064 ASM("push r11"); \
00065 ASM("push r12"); \
00066 ASM("push r13"); \
00067 ASM("push r14"); \
00068 ASM("push r15"); \
00069 ASM("push r16"); \
00070 ASM("push r17"); \
00071 ASM("push r18"); \
00072 ASM("push r19"); \
00073 ASM("push r20"); \
00074 ASM("push r21"); \
00075 ASM("push r22"); \
00076 ASM("push r23"); \
00077 ASM("push r24"); \
00078 ASM("push r25"); \
00079 ASM("push r26"); \
00080 ASM("push r27"); \
00081 ASM("push r28"); \
00082 ASM("push r29"); \
00083 ASM("push r30"); \
00084 ASM("push r31"); \
00085 ASM("lds r26, g_pclCurrent"); \
00086 ASM("lds r27, g_pclCurrent + 1"); \
00087 ASM("adiw r26, 4"); \
00088 ASM("in r0, 0x3D"); \
00089 ASM("st x+, r0"); \
00090 ASM("in r0, 0x3E"); \
00091 ASM("st x+, r0");
00092
00093 //-----
00095 #define Thread_RestoreContext() \
00096 ASM("lds r26, g_pclCurrent"); \
00097 ASM("lds r27, g_pclCurrent + 1"); \
00098 ASM("adiw r26, 4"); \

```

```

00099 ASM("ld      r28, x+"); \
00100 ASM("out 0x3D, r28"); \
00101 ASM("ld      r29, x+"); \
00102 ASM("out 0x3E, r29"); \
00103 ASM("pop r31"); \
00104 ASM("pop r30"); \
00105 ASM("pop r29"); \
00106 ASM("pop r28"); \
00107 ASM("pop r27"); \
00108 ASM("pop r26"); \
00109 ASM("pop r25"); \
00110 ASM("pop r24"); \
00111 ASM("pop r23"); \
00112 ASM("pop r22"); \
00113 ASM("pop r21"); \
00114 ASM("pop r20"); \
00115 ASM("pop r19"); \
00116 ASM("pop r18"); \
00117 ASM("pop r17"); \
00118 ASM("pop r16"); \
00119 ASM("pop r15"); \
00120 ASM("pop r14"); \
00121 ASM("pop r13"); \
00122 ASM("pop r12"); \
00123 ASM("pop r11"); \
00124 ASM("pop r10"); \
00125 ASM("pop r9"); \
00126 ASM("pop r8"); \
00127 ASM("pop r7"); \
00128 ASM("pop r6"); \
00129 ASM("pop r5"); \
00130 ASM("pop r4"); \
00131 ASM("pop r3"); \
00132 ASM("pop r2"); \
00133 ASM("pop r1"); \
00134 ASM("pop r0"); \
00135 ASM("out __SREG__, r0"); \
00136 ASM("pop r0");
00137
00138 //-----
00140 //-----
00142 #define CS_ENTER() \
00143 { \
00144 volatile uint8_t x; \
00145 x = _SFR_IO8(SR_); \
00146 ASM("cli");
00147 //-----
00149 #define CS_EXIT() \
00150 _SFR_IO8(SR_) = x;\
00151 }
00152
00153 //-----
00155 #define ENABLE_INTS()      ASM("sei");
00156 #define DISABLE_INTS()    ASM("cli");
00157
00158 //-----
00159 class Thread;
00167 class ThreadPort
00168 {
00169 public:
00175     static void StartThreads();
00176     friend class Thread;
00177 private:
00178
00186     static void InitStack(Thread *pstThread_);
00187 };
00188
00189 #endif //__ThreadPORT_H_

```

18.21 /home/mo/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp

File Reference

ATMega328p Multithreading.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "threadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "timerlist.h"
#include "quantum.h"
#include "kernel.h"
#include "kernelaware.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

Functions

- **ISR** (INT0_vect) __attribute__((signal))
ISR(INT0_vect) SWI using INT0 - used to trigger a context switch.
- **ISR** (TIMER1_COMPA_vect)
ISR(TIMER1_COMPA_vect) Timer interrupt ISR - causes a tick, which may cause a context switch.

18.21.1 Detailed Description

ATMega328p Multithreading.

Definition in file [threadport.cpp](#).

18.22 threadport.cpp

```
00001 /*=====
00002
00003 
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024 #include "thread.h"
00025 #include "threadport.h"
00026 #include "kernelswi.h"
00027 #include "kerneltimer.h"
00028 #include "timerlist.h"
00029 #include "quantum.h"
00030 #include "kernel.h"
00031 #include "kernelaware.h"
00032 #include <avr/io.h>
00033 #include <avr/interrupt.h>
00034
00035 //-----
00036 Thread *g_pclCurrentThread;
00037
00038 //-----
00039 void ThreadPort::InitStack(Thread *pclThread_)
00040 {
00041     // Initialize the stack for a Thread
00042     uint16_t u16Addr;
00043     uint8_t *pu8Stack;
00044     uint16_t i;
00045
00046     // Get the address of the thread's entry function
00047     u16Addr = (uint16_t)(pclThread_>m_pfEntryPoint);
```

```

00048
00049 // Start by finding the bottom of the stack
00050 pu8Stack = (uint8_t*)pclThread->m_pwStackTop;
00051
00052 // clear the stack, and initialize it to a known-default value (easier
00053 // to debug when things go sour with stack corruption or overflow)
00054 for (i = 0; i < pclThread->m_ul6StackSize; i++)
00055 {
00056     pclThread->m_pwStack[i] = 0xFF;
00057 }
00058
00059 // Our context starts with the entry function
00060 PUSH_TO_STACK(pu8Stack, (uint8_t)(ul6Addr & 0x00FF));
00061 PUSH_TO_STACK(pu8Stack, (uint8_t)((ul6Addr >> 8) & 0x00FF));
00062
00063 // R0
00064 PUSH_TO_STACK(pu8Stack, 0x00); // R0
00065
00066 // Push status register and R1 (which is used as a constant zero)
00067 PUSH_TO_STACK(pu8Stack, 0x80); // SR
00068 PUSH_TO_STACK(pu8Stack, 0x00); // R1
00069
00070 // Push other registers
00071 for (i = 2; i <= 23; i++) //R2-R23
00072 {
00073     PUSH_TO_STACK(pu8Stack, i);
00074 }
00075
00076 // Assume that the argument is the only stack variable
00077 PUSH_TO_STACK(pu8Stack, (uint8_t)((uint16_t)(pclThread->
m_pvArg) & 0x00FF)); //R24
00078 PUSH_TO_STACK(pu8Stack, (uint8_t)((((uint16_t)(pclThread->
m_pvArg))>>8) & 0x00FF)); //R25
00079
00080 // Push the rest of the registers in the context
00081 for (i = 26; i <= 31; i++)
00082 {
00083     PUSH_TO_STACK(pu8Stack, i);
00084 }
00085
00086 // Set the top of the stack.
00087 pclThread->m_pwStackTop = (uint8_t*)pu8Stack;
00088
00089 // That's it! the thread is ready to run now.
00090 }
00091
00092 //-----
00093 static void Thread_Switch(void)
00094 {
00095     #if KERNEL_USE_IDLE_FUNC
00096     // If there's no next-thread-to-run...
00097     if (g_pclNext == Kernel::GetIdleThread())
00098     {
00099         g_pclCurrent = Kernel::GetIdleThread();
00100
00101         // Disable the SWI, and re-enable interrupts -- enter nested interrupt
00102         // mode.
00103         KernelSWI::DI();
00104
00105         uint8_t u8SR = _SFR_IO8(SR_);
00106
00107         // So long as there's no "next-to-run" thread, keep executing the Idle
00108         // function to conclusion...
00109
00110         while (g_pclNext == Kernel::GetIdleThread())
00111         {
00112             // Ensure that we run this block in an interrupt enabled context (but
00113             // with the rest of the checks being performed in an interrupt disabled
00114             // context).
00115             ASM( "sei" );
00116             Kernel::IdleFunc();
00117             ASM( "cli" );
00118         }
00119
00120         // Progress has been achieved -- an interrupt-triggered event has caused
00121         // the scheduler to run, and choose a new thread. Since we've already
00122         // saved the context of the thread we've hijacked to run idle, we can
00123         // proceed to disable the nested interrupt context and switch to the
00124         // new thread.
00125
00126         _SFR_IO8(SR_) = u8SR;
00127         KernelSWI::RI( true );
00128     }
00129     #endif
00130     g_pclCurrent = (Thread*)g_pclNext;
00131 }
00132

```

```

00133
00134 //-----
00135 void ThreadPort::StartThreads()
00136 {
00137     KernelSWI::Config();           // configure the task switch SWI
00138     KernelTimer::Config();         // configure the kernel timer
00139
00140     Scheduler::SetScheduler(1);    // enable the scheduler
00141     Scheduler::Schedule();         // run the scheduler - determine the first
                                     // thread to run
00142
00143     Thread_Switch();               // Set the next scheduled thread to the current thread
00144
00145     KernelTimer::Start();          // enable the kernel timer
00146     KernelSWI::Start();            // enable the task switch SWI
00147
00148     // Restore the context...
00149     Thread_RestoreContext();       // restore the context of the first running thread
00150     ASM("reti");                  // return from interrupt - will return to the first scheduled thread
00151 }
00152
00153 //-----
00154 //-----
00155 ISR(INT0_vect) __attribute__ ( ( signal, naked ) );
00160 ISR(INT0_vect)
00161 {
00162     Thread_SaveContext();          // Push the context (registers) of the current task
00163     Thread_Switch();              // Switch to the next task
00164     Thread_RestoreContext();       // Pop the context (registers) of the next task
00165     ASM("reti");                  // Return to the next task
00166 }
00167
00168 //-----
00173 //-----
00174 ISR(TIMER1_COMPA_vect)
00175 {
00176     #if KERNEL_USE_TIMERS
00177         TimerScheduler::Process();
00178     #endif
00179     #if KERNEL_USE_QUANTUM
00180         Quantum::UpdateTimer();
00181     #endif
00182 }

```

18.23 /home/mo/mark3-source/embedded/kernel/driver.cpp File Reference

Device driver/hardware abstraction layer.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "driver.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

Classes

- class [DevNull](#)
This class implements the "default" driver (/dev/null)

Functions

- static uint8_t [DrvCmp](#) (const char *szStr1_, const char *szStr2_)
DrvCmp.

Variables

- static [DevNull](#) [clDevNull](#)


```

00051     virtual uint8_t Close() { return 0; }
00052
00053     virtual uint16_t Read( uint16_t ul6Bytes_,
00054         uint8_t *pu8Data_){ return 0; }
00055
00056     virtual uint16_t Write( uint16_t ul6Bytes_,
00057         uint8_t *pu8Data_ ) { return 0; }
00058
00059     virtual uint16_t Control( uint16_t ul6Event_,
00060         void *pvDataIn_,
00061         uint16_t ul6SizeIn_,
00062         void *pvDataOut_,
00063         uint16_t ul6SizeOut_ ) { return 0; }
00064
00065 };
00066
00067 //-----
00068 static DevNull clDevNull;
00069
00070 //-----
00081 static uint8_t DrvCmp( const char *szStr1_, const char *szStr2_ )
00082 {
00083     char *szTmp1 = (char*) szStr1_;
00084     char *szTmp2 = (char*) szStr2_;
00085
00086     while (*szTmp1 && *szTmp2)
00087     {
00088         if (*szTmp1++ != *szTmp2++)
00089         {
00090             return 0;
00091         }
00092     }
00093
00094     // Both terminate at the same length
00095     if (!(*szTmp1) && !(*szTmp2))
00096     {
00097         return 1;
00098     }
00099
00100     return 0;
00101 }
00102
00103 //-----
00104 void DriverList::Init()
00105 {
00106     // Ensure we always have at least one entry - a default in case no match
00107     // is found (/dev/null)
00108     clDevNull.Init();
00109     Add(&clDevNull);
00110 }
00111
00112 //-----
00113 Driver *DriverList::FindByPath( const char *m_pcPath )
00114 {
00115     KERNEL_ASSERT( m_pcPath );
00116     Driver *pclTemp = static_cast<Driver*>(m_clDriverList.
00117         GetHead());
00118
00119     // Iterate through the list of drivers until we find a match, or we
00120     // exhaust our list of installed drivers
00121     while (pclTemp)
00122     {
00123         if(DrvCmp(m_pcPath, pclTemp->GetPath()))
00124         {
00125             return pclTemp;
00126         }
00127         pclTemp = static_cast<Driver*>(pclTemp->GetNext());
00128     }
00129     // No matching driver found - return a pointer to our /dev/null driver
00130     return &clDevNull;
00131 }
00132 #endif

```

18.25 /home/mo/mark3-source/embedded/kernel/eventflag.cpp File Reference

Event Flag Blocking Object/IPC-Object implementation.


```

00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined! Bailing."
00032 #else
00033 # define DBG_FILE _DBG__KERNEL_EVENTFLAG_CPP
00034 #endif
00035 //--[End Autogenerated content]-----
00036
00037 #if KERNEL_USE_EVENTFLAG
00038
00039 #if KERNEL_USE_TIMEOUTS
00040 #include "timerlist.h"
00041 //-----
00053 void TimedEventFlag_Callback(Thread *pclOwner_, void *pvData_)
00054 {
00055     EventFlag *pclEventFlag = static_cast<EventFlag*>(pvData_);
00056
00057     pclEventFlag->WakeMe(pclOwner_);
00058     pclOwner_->SetExpired(true);
00059     pclOwner_->SetEventFlagMask(0);
00060
00061     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
00062         ()->GetCurPriority())
00063     {
00064         Thread::Yield();
00065     }
00066 }
00067 //-----
00068 void EventFlag::WakeMe(Thread *pclChosenOne_)
00069 {
00070     Unblock(pclChosenOne_);
00071 }
00072 #endif
00073
00074 //-----
00075 #if KERNEL_USE_TIMEOUTS
00076 uint16_t EventFlag::Wait_i(uint16_t ul6Mask_,
00077     EventFlagOperation_t eMode_, uint32_t u32TimeMS_)
00078 #else
00079 uint16_t EventFlag::Wait_i(uint16_t ul6Mask_,
00080     EventFlagOperation_t eMode_)
00081 #endif
00082 {
00083     bool bThreadYield = false;
00084     bool bMatch = false;
00085
00086     #if KERNEL_USE_TIMEOUTS
00087     Timer clEventTimer;
00088     bool bUseTimer = false;
00089     #endif
00090
00091     // Ensure we're operating in a critical section while we determine
00092     // whether or not we need to block the current thread on this object.
00093     CS_ENTER();
00094
00095     // Check to see whether or not the current mask matches any of the
00096     // desired bits.
00097     g_pclCurrent->SetEventFlagMask(ul6Mask_);
00098
00099     if ((eMode_ == EVENT_FLAG_ALL) || (eMode_ ==
00100     EVENT_FLAG_ALL_CLEAR))
00101     {
00102         // Check to see if the flags in their current state match all of
00103         // the set flags in the event flag group, with this mask.
00104         if ((m_ul6SetMask & ul6Mask_) == ul6Mask_)
00105         {
00106             bMatch = true;
00107             g_pclCurrent->SetEventFlagMask(ul6Mask_);
00108         }
00109     }
00110     else if ((eMode_ == EVENT_FLAG_ANY) || (eMode_ ==
00111     EVENT_FLAG_ANY_CLEAR))
00112     {
00113         // Check to see if the existing flags match any of the set flags in
00114         // the event flag group with this mask
00115         if (m_ul6SetMask & ul6Mask_)
00116         {
00117             bMatch = true;
00118             g_pclCurrent->SetEventFlagMask(m_ul6SetMask & ul6Mask_);
00119         }
00120     }
00121
00122     // We're unable to match this pattern as-is, so we must block.
00123     if (!bMatch)
00124     {

```

```

00121         // Reset the current thread's event flag mask & mode
00122         g_pclCurrent->SetEventFlagMask(u16Mask_);
00123         g_pclCurrent->SetEventFlagMode(eMode_);
00124
00125 #if KERNEL_USE_TIMEOUTS
00126         if (u32TimeMS_)
00127         {
00128             g_pclCurrent->SetExpired(false);
00129             clEventTimer.Init();
00130             clEventTimer.Start(0, u32TimeMS_, TimedEventFlag_Callback, (void*)
00131                 this);
00132             bUseTimer = true;
00133         }
00134 #endif
00135         // Add the thread to the object's block-list.
00136         BlockPriority(g_pclCurrent);
00137
00138         // Trigger that
00139         bThreadYield = true;
00140     }
00141
00142     // If bThreadYield is set, it means that we've blocked the current thread,
00143     // and must therefore rerun the scheduler to determine what thread to
00144     // switch to.
00145     if (bThreadYield)
00146     {
00147         // Switch threads immediately
00148         Thread::Yield();
00149     }
00150
00151     // Exit the critical section and return back to normal execution
00152     CS_EXIT();
00153
00154 #if KERNEL_USE_TIMEOUTS
00155     if (bUseTimer && bThreadYield)
00156     {
00157         clEventTimer.Stop();
00158     }
00159 #endif
00160     return g_pclCurrent->GetEventFlagMask();
00161 }
00162
00163 //-----
00164 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_)
00165 {
00166     #if KERNEL_USE_TIMEOUTS
00167         return Wait_i(u16Mask_, eMode_, 0);
00168     #else
00169         return Wait_i(u16Mask_, eMode_);
00170     #endif
00171 }
00172
00173 #if KERNEL_USE_TIMEOUTS
00174 //-----
00175 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_,
00176     uint32_t u32TimeMS_)
00177 {
00178     return Wait_i(u16Mask_, eMode_, u32TimeMS_);
00179 }
00180 #endif
00181
00182 //-----
00183 void EventFlag::Set(uint16_t u16Mask_)
00184 {
00185     Thread *pclPrev;
00186     Thread *pclCurrent;
00187     bool bReschedule = false;
00188     uint16_t u16NewMask;
00189
00190     CS_ENTER();
00191
00192     // Walk through the whole block list, checking to see whether or not
00193     // the current flag set now matches any/all of the masks and modes of
00194     // the threads involved.
00195
00196     m_u16SetMask |= u16Mask_;
00197     u16NewMask = m_u16SetMask;
00198
00199     // Start at the head of the list, and iterate through until we hit the
00200     // "head" element in the list again. Ensure that we handle the case where
00201     // we remove the first or last elements in the list, or if there's only
00202     // one element in the list.
00203     pclCurrent = static_cast<Thread*>(m_clBlockList.GetHead());
00204
00205     // Do nothing when there are no objects blocking.

```

```

00210     if (pclCurrent)
00211     {
00212         // First loop - process every thread in the block-list and check to
00213         // see whether or not the current flags match the event-flag conditions
00214         // on the thread.
00215         do
00216         {
00217             pclPrev = pclCurrent;
00218             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00219
00220             // Read the thread's event mask/mode
00221             uint16_t ul6ThreadMask = pclPrev->GetEventFlagMask();
00222             EventFlagOperation_t eThreadMode = pclPrev->
GetEventFlagMode();
00223
00224             // For the "any" mode - unblock the blocked threads if one or more bits
00225             // in the thread's bitmask match the object's bitmask
00226             if ((EVENT_FLAG_ANY == eThreadMode) || (
EVENT_FLAG_ANY_CLEAR == eThreadMode))
00227             {
00228                 if (ul6ThreadMask & m_ul6SetMask)
00229                 {
00230                     pclPrev->SetEventFlagMode(
EVENT_FLAG_PENDING_UNBLOCK);
00231                     pclPrev->SetEventFlagMask(m_ul6SetMask & ul6ThreadMask);
00232                     bReschedule = true;
00233
00234                     // If the "clear" variant is set, then clear the bits in the mask
00235                     // that caused the thread to unblock.
00236                     if (EVENT_FLAG_ANY_CLEAR == eThreadMode)
00237                     {
00238                         ul6NewMask &=~ (ul6ThreadMask & ul6Mask_);
00239                     }
00240                 }
00241             }
00242             // For the "all" mode, every set bit in the thread's requested bitmask must
00243             // match the object's flag mask.
00244             else if ((EVENT_FLAG_ALL == eThreadMode) || (
EVENT_FLAG_ALL_CLEAR == eThreadMode))
00245             {
00246                 if ((ul6ThreadMask & m_ul6SetMask) == ul6ThreadMask)
00247                 {
00248                     pclPrev->SetEventFlagMode(
EVENT_FLAG_PENDING_UNBLOCK);
00249                     pclPrev->SetEventFlagMask(ul6ThreadMask);
00250                     bReschedule = true;
00251
00252                     // If the "clear" variant is set, then clear the bits in the mask
00253                     // that caused the thread to unblock.
00254                     if (EVENT_FLAG_ALL_CLEAR == eThreadMode)
00255                     {
00256                         ul6NewMask &=~ (ul6ThreadMask & ul6Mask_);
00257                     }
00258                 }
00259             }
00260         }
00261         // To keep looping, ensure that there's something in the list, and
00262         // that the next item isn't the head of the list.
00263         while (pclPrev != m_clBlockList.GetTail());
00264
00265         // Second loop - go through and unblock all of the threads that
00266         // were tagged for unblocking.
00267         pclCurrent = static_cast<Thread*>(m_clBlockList.
GetHead());
00268         bool bIsTail = false;
00269         do
00270         {
00271             pclPrev = pclCurrent;
00272             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00273
00274             // Check to see if this is the condition to terminate the loop
00275             if (pclPrev == m_clBlockList.GetTail())
00276             {
00277                 bIsTail = true;
00278             }
00279
00280             // If the first pass indicated that this thread should be
00281             // unblocked, then unblock the thread
00282             if (pclPrev->GetEventFlagMode() ==
EVENT_FLAG_PENDING_UNBLOCK)
00283             {
00284                 Unblock(pclPrev);
00285             }
00286         }
00287         while (!bIsTail);
00288     }
00289

```

```

00290 // If we awoke any threads, re-run the scheduler
00291 if (bReschedule)
00292 {
00293     Thread::Yield();
00294 }
00295
00296 // Update the bitmask based on any "clear" operations performed along
00297 // the way
00298 m_ul6SetMask = ul6NewMask;
00299
00300 // Restore interrupts - will potentially cause a context switch if a
00301 // thread is unblocked.
00302 CS_EXIT();
00303 }
00304
00305 //-----
00306 void EventFlag::Clear(uint16_t ul6Mask_)
00307 {
00308     // Just clear the bitfields in the local object.
00309     CS_ENTER();
00310     m_ul6SetMask &= ~ul6Mask_;
00311     CS_EXIT();
00312 }
00313
00314 //-----
00315 uint16_t EventFlag::GetMask()
00316 {
00317     // Return the presently held event flag values in this object. Ensure
00318     // we get this within a critical section to guarantee atomicity.
00319     uint16_t ul6Return;
00320     CS_ENTER();
00321     ul6Return = m_ul6SetMask;
00322     CS_EXIT();
00323     return ul6Return;
00324 }
00325
00326 #endif // KERNEL_USE_EVENTFLAG

```

18.27 /home/mo/mark3-source/embedded/kernel/kernel.cpp File Reference

[Kernel](#) initialization and startup code.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernel.h"
#include "scheduler.h"
#include "thread.h"
#include "threadport.h"
#include "timerlist.h"
#include "message.h"
#include "driver.h"
#include "profile.h"
#include "kernelprofile.h"
#include "autoalloc.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "tracebuffer.h"

```

18.27.1 Detailed Description

[Kernel](#) initialization and startup code.

Definition in file [kernel.cpp](#).

18.28 kernel.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023
00024 #include "kernel.h"
00025 #include "scheduler.h"
00026 #include "thread.h"
00027 #include "threadport.h"
00028 #include "timerlist.h"
00029 #include "message.h"
00030 #include "driver.h"
00031 #include "profile.h"
00032 #include "kernelprofile.h"
00033 #include "autoalloc.h"
00034
00035 #define _CAN_HAS_DEBUG
00036 //--[Autogenerated - Do Not Modify]-----
00037 #include "dbg_file_list.h"
00038 #include "buffalogger.h"
00039 #if defined(DBG_FILE)
00040 # error "Debug logging file token already defined! Bailing."
00041 #else
00042 # define DBG_FILE _DBG__KERNEL_KERNEL_CPP
00043 #endif
00044 //--[End Autogenerated content]-----
00045 #include "kerneldebug.h"
00046 #include "tracebuffer.h"
00047
00048 bool Kernel::m_bIsStarted;
00049 bool Kernel::m_bIsPanic;
00050 panic_func_t Kernel::m_pfPanic;
00051
00052 #if KERNEL_USE_IDLE_FUNC
00053 idle_func_t Kernel::m_pfIdle;
00054 FakeThread_t Kernel::m_clIdle;
00055 #endif
00056 //-----
00057 void Kernel::Init(void)
00058 {
00059     m_bIsStarted = false;
00060     m_bIsPanic = false;
00061     m_pfPanic = 0;
00062     #if KERNEL_USE_AUTO_ALLOC
00063     AutoAlloc::Init();
00064     #endif
00065     #if KERNEL_USE_IDLE_FUNC
00066     ((Thread*)&m_clIdle)->InitIdle();
00067     m_pfIdle = 0;
00068     #endif
00069     #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00070     TraceBuffer::Init();
00071     #endif
00072     KERNEL_TRACE( "Initializing Mark3 Kernel" );
00073
00074     // Initialize the global kernel data - scheduler, timer-scheduler, and
00075     // the global message pool.
00076     Scheduler::Init();
00077     #if KERNEL_USE_DRIVER
00078     DriverList::Init();
00079     #endif
00080     #if KERNEL_USE_TIMERS
00081     TimerScheduler::Init();
00082     #endif
00083     #if KERNEL_USE_MESSAGE
00084     GlobalMessagePool::Init();
00085     #endif
00086     #if KERNEL_USE_PROFILER
00087     Profiler::Init();
00088     #endif
00089 }
00090
00091

```

```

00092 //-----
00093 void Kernel::Start(void)
00094 {
00095     KERNEL_TRACE( "Starting Mark3 Scheduler" );
00096     m_bIsStarted = true;
00097     ThreadPort::StartThreads();
00098     KERNEL_TRACE( "Error starting Mark3 Scheduler" );
00099 }
00100
00101 //-----
00102 void Kernel::Panic(uint16_t u16Cause_)
00103 {
00104     m_bIsPanic = true;
00105     if (m_pfPanic)
00106     {
00107         m_pfPanic(u16Cause_);
00108     }
00109     else
00110     {
00111         #if KERNEL_AWARE_SIMULATION
00112             KernelAware::ExitSimulator();
00113         #endif
00114         while(1);
00115     }
00116 }

```

18.29 /home/mo/mark3-source/embedded/kernel/kernelaware.cpp File Reference

[Kernel](#) aware simulation support.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernelaware.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"

```

Classes

- union [KernelAwareData_t](#)

This structure is used to communicate between the kernel and a kernel- aware host.

Variables

- volatile bool [g_bIsKernelAware](#) = false
Will be set to true by a kernel-aware host.
- volatile uint8_t [g_u8KACommand](#)
Kernel-aware simulator command to execute.
- [KernelAwareData_t](#) [g_stKAData](#)
Data structure used to communicate with host.

18.29.1 Detailed Description

[Kernel](#) aware simulation support.

Definition in file [kernelaware.cpp](#).

18.29.2 Variable Documentation

18.29.2.1 volatile bool g_bIsKernelAware = false

Will be set to true by a kernel-aware host.

Definition at line 81 of file [kernelaware.cpp](#).

18.29.2.2 KernelAwareData_t g_stKAData

Data structure used to communicate with host.

Definition at line 83 of file [kernelaware.cpp](#).

18.30 kernelaware.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "kernelaware.h"
00024 #include "threadport.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-----
00028 #include "dbg_file_list.h"
00029 #include "buffallogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined! Bailing."
00032 #else
00033 # define DBG_FILE _DBG__KERNEL_KERNELAWARE_CPP
00034 #endif
00035 //--[End Autogenerated content]-----
00036
00037 #if KERNEL_AWARE_SIMULATION
00038
00039 //-----
00048 typedef union
00049 {
00050     volatile uint16_t au16Buffer[5];
00051
00052     struct
00053     {
00054         volatile const char *szName;
00055     } Profiler;
00056     struct
00057     {
00058         volatile uint16_t u16File;
00059         volatile uint16_t u16Line;
00060         volatile uint16_t u16Arg1;
00061         volatile uint16_t u16Arg2;
00062     } Trace;
00063     struct
00064     {
00065         volatile const char *szString;
00066     } Print;
00067 } KernelAwareData_t;
00078
00079
00080 //-----
00081 volatile bool g_bIsKernelAware = false;
00082 volatile uint8_t g_u8KACommand;
00083 KernelAwareData_t g_stKAData;
00084
00085
00086 //-----
00087 void KernelAware::ProfileInit(const char *szStr_)
00088 {
00089     CS_ENTER();
00090     g_stKAData.Profiler.szName = szStr_;

```

```

00091     g_u8KACommand = KA_COMMAND_PROFILE_INIT;
00092     CS_EXIT();
00093 }
00094
00095 //-----
00096 void KernelAware::ProfileStart(void)
00097 {
00098     g_u8KACommand = KA_COMMAND_PROFILE_START;
00099 }
00100
00101 //-----
00102 void KernelAware::ProfileStop(void)
00103 {
00104     g_u8KACommand = KA_COMMAND_PROFILE_STOP;
00105 }
00106
00107 //-----
00108 void KernelAware::ProfileReport(void)
00109 {
00110     g_u8KACommand = KA_COMMAND_PROFILE_REPORT;
00111 }
00112
00113 //-----
00114 void KernelAware::ExitSimulator(void)
00115 {
00116     g_u8KACommand = KA_COMMAND_EXIT_SIMULATOR;
00117 }
00118
00119 //-----
00120 void KernelAware::Trace( uint16_t ul6File_,
00121                          uint16_t ul6Line_ )
00122 {
00123     Trace_i( ul6File_, ul6Line_, 0, 0, KA_COMMAND_TRACE_0 );
00124 }
00125
00126 //-----
00127 void KernelAware::Trace( uint16_t ul6File_,
00128                          uint16_t ul6Line_,
00129                          uint16_t ul6Arg1_ )
00130 {
00131     Trace_i( ul6File_, ul6Line_, ul6Arg1_, 0, KA_COMMAND_TRACE_1 );
00132 }
00133
00134 //-----
00135 void KernelAware::Trace( uint16_t ul6File_,
00136                          uint16_t ul6Line_,
00137                          uint16_t ul6Arg1_,
00138                          uint16_t ul6Arg2_ )
00139 {
00140     Trace_i( ul6File_, ul6Line_, ul6Arg1_, ul6Arg2_, KA_COMMAND_TRACE_2 );
00141 }
00142
00143 //-----
00144 void KernelAware::Trace_i( uint16_t ul6File_,
00145                           uint16_t ul6Line_,
00146                           uint16_t ul6Arg1_,
00147                           uint16_t ul6Arg2_,
00148                           KernelAwareCommand_t eCmd_ )
00149 {
00150     CS_ENTER();
00151     g_stKADData.Trace.ul6File = ul6File_;
00152     g_stKADData.Trace.ul6Line = ul6Line_;
00153     g_stKADData.Trace.ul6Arg1 = ul6Arg1_;
00154     g_stKADData.Trace.ul6Arg2 = ul6Arg2_;
00155     g_u8KACommand = eCmd_;
00156     CS_EXIT();
00157 }
00158
00159 //-----
00160 void KernelAware::Print(const char *szStr_)
00161 {
00162     CS_ENTER();
00163     g_stKADData.Print.szString = szStr_;
00164     g_u8KACommand = KA_COMMAND_PRINT;
00165     CS_EXIT();
00166 }
00167
00168 //-----
00169 bool KernelAware::IsSimulatorAware(void)
00170 {
00171     return g_bIsKernelAware;
00172 }
00173
00174 #endif

```


18.31 /home/mo/mark3-source/embedded/kernel/ksemaphore.cpp File Reference

[Semaphore](#) Blocking-Object Implemenation.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "blocking.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

Functions

- void [TimedSemaphore_Callback](#) ([Thread](#) *pclOwner_, void *pvData_)
TimedSemaphore_Callback.

18.31.1 Detailed Description

[Semaphore](#) Blocking-Object Implemenation.

Definition in file [ksemaphore.cpp](#).

18.31.2 Function Documentation

18.31.2.1 void [TimedSemaphore_Callback](#) ([Thread](#) * *pclOwner_*, void * *pvData_*)

[TimedSemaphore_Callback.](#)

This function is called from the timer-expired context to trigger a timeout on this semaphore. This results in the waking of the thread that generated the semaphore pend call that was not completed in time.

Parameters

<i>pclOwner_</i>	Pointer to the thread to wake
<i>pvData_</i>	Pointer to the semaphore object that the thread is blocked on

Definition at line 57 of file [ksemaphore.cpp](#).

18.32 ksemaphore.cpp

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "ksemaphore.h"
00026 #include "blocking.h"
00027
00028 #define _CAN_HAS_DEBUG
```

```

00029 //--[Autogenerated - Do Not Modify]-----
00030 #include "dbg_file_list.h"
00031 #include "buffalogger.h"
00032 #if defined(DBG_FILE)
00033 # error "Debug logging file token already defined!  Bailing."
00034 #else
00035 # define DBG_FILE _DBG__KERNEL_KSEMAPHORE_CPP
00036 #endif
00037 //--[End Autogenerated content]-----
00038 #include "kerneldebug.h"
00039
00040
00041 #if KERNEL_USE_SEMAPHORE
00042
00043 #if KERNEL_USE_TIMEOUTS
00044 #include "timerlist.h"
00045
00046 //-----
00057 void TimedSemaphore_Callback(Thread *pclOwner_, void *pvData_)
00058 {
00059     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00060
00061     // Indicate that the semaphore has expired on the thread
00062     pclOwner_>SetExpired(true);
00063
00064     // Wake up the thread that was blocked on this semaphore.
00065     pclSemaphore->WakeMe(pclOwner_);
00066
00067
00068     if (pclOwner_>GetCurPriority() >= Scheduler::GetCurrentThread
        ()->GetCurPriority())
00069     {
00070         Thread::Yield();
00071     }
00072 }
00073
00074 //-----
00075 void Semaphore::WakeMe(Thread *pclChosenOne_)
00076 {
00077     // Remove from the semaphore waitlist and back to its ready list.
00078     Unblock(pclChosenOne_);
00079 }
00080
00081 #endif // KERNEL_USE_TIMEOUTS
00082
00083 //-----
00084 uint8_t Semaphore::WakeNext()
00085 {
00086     Thread *pclChosenOne;
00087
00088     pclChosenOne = m_clBlockList.HighestWaiter();
00089
00090     // Remove from the semaphore waitlist and back to its ready list.
00091     Unblock(pclChosenOne);
00092
00093     // Call a task switch if higher or equal priority thread
00094     if (pclChosenOne->GetCurPriority() >=
        Scheduler::GetCurrentThread()->GetCurPriority())
00095     {
00096         return 1;
00097     }
00098     return 0;
00099 }
00100
00101 //-----
00102 void Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)
00103 {
00104     // Copy the paramters into the object - set the maximum value for this
00105     // semaphore to implement either binary or counting semaphores, and set
00106     // the initial count.  Clear the wait list for this object.
00107     m_u16Value = u16InitVal_;
00108     m_u16MaxValue = u16MaxVal_;
00109
00110     m_clBlockList.Init();
00111 }
00112
00113 //-----
00114 bool Semaphore::Post()
00115 {
00116     KERNEL_TRACE_1( "Posting semaphore, Thread %d", (uint16_t)
        g_pclCurrent->GetID() );
00117
00118     bool bThreadWake = 0;
00119     bool bBail = false;
00120     // Increment the semaphore count - we can mess with threads so ensure this
00121     // is in a critical section.  We don't just disable the scheudler since
00122     // we want to be able to do this from within an interrupt context as well.

```

```

00123     CS_ENTER();
00124
00125     // If nothing is waiting for the semaphore
00126     if (m_clBlockList.GetHead() == NULL)
00127     {
00128         // Check so see if we've reached the maximum value in the semaphore
00129         if (m_ul6Value < m_ul6MaxValue)
00130         {
00131             // Increment the count value
00132             m_ul6Value++;
00133         }
00134         else
00135         {
00136             // Maximum value has been reached, bail out.
00137             bBail = true;
00138         }
00139     }
00140     else
00141     {
00142         // Otherwise, there are threads waiting for the semaphore to be
00143         // posted, so wake the next one (highest priority goes first).
00144         bThreadWake = WakeNext();
00145     }
00146
00147     CS_EXIT();
00148
00149     // If we weren't able to increment the semaphore count, fail out.
00150     if (bBail)
00151     {
00152         return false;
00153     }
00154
00155     // if bThreadWake was set, it means that a higher-priority thread was
00156     // woken. Trigger a context switch to ensure that this thread gets
00157     // to execute next.
00158     if (bThreadWake)
00159     {
00160         Thread::Yield();
00161     }
00162     return true;
00163 }
00164
00165 //-----
00166 #if KERNEL_USE_TIMEOUTS
00167 bool Semaphore::Pend_i( uint32_t u32WaitTimeMS_ )
00168 #else
00169 void Semaphore::Pend_i( void )
00170 #endif
00171 {
00172     KERNEL_TRACE_1( "Pending semaphore, Thread %d", (uint16_t)
g_pclCurrent->GetID() );
00173
00174 #if KERNEL_USE_TIMEOUTS
00175     Timer clSemTimer;
00176     bool bUseTimer = false;
00177 #endif
00178
00179     // Once again, messing with thread data - ensure
00180     // we're doing all of these operations from within a thread-safe context.
00181     CS_ENTER();
00182
00183     // Check to see if we need to take any action based on the semaphore count
00184     if (m_ul6Value != 0)
00185     {
00186         // The semaphore count is non-zero, we can just decrement the count
00187         // and go along our merry way.
00188         m_ul6Value--;
00189     }
00190     else
00191     {
00192         // The semaphore count is zero - we need to block the current thread
00193         // and wait until the semaphore is posted from elsewhere.
00194         #if KERNEL_USE_TIMEOUTS
00195             if (u32WaitTimeMS_)
00196             {
00197                 g_pclCurrent->SetExpired(false);
00198                 clSemTimer.Init();
00199                 clSemTimer.Start(0, u32WaitTimeMS_, TimedSemaphore_Callback, (void*)this
);
00200                 bUseTimer = true;
00201             }
00202         #endif
00203         BlockPriority(g_pclCurrent);
00204
00205         // Switch Threads immediately
00206         Thread::Yield();
00207     }

```

```

00208
00209     CS_EXIT();
00210
00211 #if KERNEL_USE_TIMEOUTS
00212     if (bUseTimer)
00213     {
00214         clSemTimer.Stop();
00215         return (g_pclCurrent->GetExpired() == 0);
00216     }
00217     return true;
00218 #endif
00219 }
00220
00221 //-----
00222 // Redirect the untimed pend API to the timed pend, with a null timeout.
00223 void Semaphore::Pend()
00224 {
00225     #if KERNEL_USE_TIMEOUTS
00226         Pend_i(0);
00227     #else
00228         Pend_i();
00229     #endif
00230 }
00231
00232 #if KERNEL_USE_TIMEOUTS
00233 //-----
00234 bool Semaphore::Pend( uint32_t u32WaitTimeMS_ )
00235 {
00236     return Pend_i( u32WaitTimeMS_ );
00237 }
00238 #endif
00239
00240 //-----
00241 uint16_t Semaphore::GetCount()
00242 {
00243     uint16_t ul6Ret;
00244     CS_ENTER();
00245     ul6Ret = m_ul6Value;
00246     CS_EXIT();
00247     return ul6Ret;
00248 }
00249
00250 #endif

```

18.33 /home/mo/mark3-source/embedded/kernel/ll.cpp File Reference

Core Linked-List implementation, from which all kernel objects are derived.

```

#include "kerneltypes.h"
#include "kernel.h"
#include "ll.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

18.33.1 Detailed Description

Core Linked-List implementation, from which all kernel objects are derived.

Definition in file [ll.cpp](#).

18.34 ll.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008

```

```

00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "kernel.h"
00024 #include "ll.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-----
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined! Bailing."
00032 #else
00033 # define DBG_FILE _DBG__KERNEL_LL_CPP
00034 #endif
00035 //--[End Autogenerated content]-----
00036
00037 #include "kerneldebug.h"
00038
00039 //-----
00040 void LinkListNode::ClearNode()
00041 {
00042     next = NULL;
00043     prev = NULL;
00044 }
00045
00046 //-----
00047 void DoubleLinkedList::Add(LinkListNode *node_)
00048 {
00049     KERNEL_ASSERT( node_ );
00050
00051     // Add a node to the end of the linked list.
00052     if (!m_pstHead)
00053     {
00054         // If the list is empty, initilize the nodes
00055         m_pstHead = node_;
00056         m_pstTail = node_;
00057
00058         m_pstHead->prev = NULL;
00059         m_pstTail->next = NULL;
00060         return;
00061     }
00062
00063     // Move the tail node, and assign it to the new node just passed in
00064     m_pstTail->next = node_;
00065     node_->prev = m_pstTail;
00066     node_->next = NULL;
00067     m_pstTail = node_;
00068 }
00069
00070 //-----
00071 void DoubleLinkedList::Remove(LinkListNode *node_)
00072 {
00073     KERNEL_ASSERT( node_ );
00074
00075     if (node_->prev)
00076     {
00077         #if SAFE_UNLINK
00078             if (node_->prev->next != node_)
00079             {
00080                 Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00081             }
00082         #endif
00083         node_->prev->next = node_->next;
00084     }
00085     if (node_->next)
00086     {
00087         #if SAFE_UNLINK
00088             if (node_->next->prev != node_)
00089             {
00090                 Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00091             }
00092         #endif
00093         node_->next->prev = node_->prev;
00094     }
00095     if (node_ == m_pstHead)
00096     {
00097         m_pstHead = node_->next;
00098     }
00099     if (node_ == m_pstTail)
00100     {
00101         m_pstTail = node_->prev;
00102     }
00103 }

```

```

00104     node_>ClearNode();
00105 }
00106
00107 //-----
00108 void CircularLinkedList::Add(LinkListNode *node_)
00109 {
00110     KERNEL_ASSERT( node_ );
00111
00112     // Add a node to the end of the linked list.
00113     if (!m_pstHead)
00114     {
00115         // If the list is empty, initilize the nodes
00116         m_pstHead = node_;
00117         m_pstTail = node_;
00118
00119         m_pstHead->prev = m_pstHead;
00120         m_pstHead->next = m_pstHead;
00121         return;
00122     }
00123
00124     // Move the tail node, and assign it to the new node just passed in
00125     m_pstTail->next = node_;
00126     node_>prev = m_pstTail;
00127     node_>next = m_pstHead;
00128     m_pstTail = node_;
00129     m_pstHead->prev = node_;
00130 }
00131
00132 //-----
00133 void CircularLinkedList::Remove(LinkListNode *node_)
00134 {
00135     KERNEL_ASSERT( node_ );
00136
00137     // Check to see if this is the head of the list...
00138     if ((node_ == m_pstHead) && (m_pstHead == m_pstTail))
00139     {
00140         // Clear the head and tail pointers - nothing else left.
00141         m_pstHead = NULL;
00142         m_pstTail = NULL;
00143         return;
00144     }
00145
00146 #if SAFE_UNLINK
00147     // Verify that all nodes are properly connected
00148     if ((node_>prev->next != node_) || (node_>next->prev != node_))
00149     {
00150         Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00151     }
00152 #endif
00153
00154     // This is a circularly linked list - no need to check for connection,
00155     // just remove the node.
00156     node_>next->prev = node_>prev;
00157     node_>prev->next = node_>next;
00158
00159     if (node_ == m_pstHead)
00160     {
00161         m_pstHead = m_pstHead->next;
00162     }
00163     if (node_ == m_pstTail)
00164     {
00165         m_pstTail = m_pstTail->prev;
00166     }
00167     node_>ClearNode();
00168 }
00169
00170 //-----
00171 void CircularLinkedList::PivotForward()
00172 {
00173     if (m_pstHead)
00174     {
00175         m_pstHead = m_pstHead->next;
00176         m_pstTail = m_pstTail->next;
00177     }
00178 }
00179
00180 //-----
00181 void CircularLinkedList::PivotBackward()
00182 {
00183     if (m_pstHead)
00184     {
00185         m_pstHead = m_pstHead->prev;
00186         m_pstTail = m_pstTail->prev;
00187     }
00188 }
00189
00190 //-----

```

```
00191 void CircularLinkedList::InsertNodeBefore(  
    LinkListNode *node_, LinkListNode *insert_)  
00192 {  
00193     KERNEL_ASSERT( node_ );  
00194  
00195     node_>next = insert_;  
00196     node_>prev = insert_>prev;  
00197  
00198     if (insert_>prev)  
00199     {  
00200         insert_>prev->next = node_;  
00201     }  
00202     insert_>prev = node_;  
00203 }  
00204
```

18.35 /home/mo/mark3-source/embedded/kernel/mailbox.cpp File Reference

Mailbox + Envelope IPC mechanism.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "ksemaphore.h"
#include "mailbox.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

18.35.1 Detailed Description

Mailbox + Envelope IPC mechanism.

Definition in file [mailbox.cpp](#).

18.36 mailbox.cpp

```

00001 /*-----
00002
00003 |-----|-----|-----|-----|-----|-----|
00004 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
00005 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
00006 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
00007 |-----|-----|-----|-----|-----|-----|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #include "mark3cfg.h"
00022 #include "kerneltypes.h"
00023 #include "ksemaphore.h"
00024 #include "mailbox.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-----
00028 #include "dbg_file_list.h"
00029 #include "buffallogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG__KERNEL_MAILBOX_CPP
00034 #endif
00035 //--[End Autogenerated content]-----
00036
00037 #include "kerneldebug.h"
00038
00039 #if KERNEL_USE_MAILBOX
00040
00041 //-----

```

```

00042 void Mailbox::Init( void *pvBuffer_, uint16_t ul6BufferSize_, uint16_t ul6ElementSize_ )
00043 {
00044     KERNEL_ASSERT(ul6BufferSize_);
00045     KERNEL_ASSERT(ul6ElementSize_);
00046     KERNEL_ASSERT(pvBuffer_);
00047
00048     m_pvBuffer = pvBuffer_;
00049     m_ul6ElementSize = ul6ElementSize_;
00050
00051     m_ul6Count = (ul6BufferSize_ / ul6ElementSize_);
00052     m_ul6Free = m_ul6Count;
00053
00054     m_ul6Head = 0;
00055     m_ul6Tail = 0;
00056
00057     // We use the counting semaphore to implement blocking - with one element
00058     // in the mailbox corresponding to a post/pend operation in the semaphore.
00059     m_clRecvSem.Init(0, m_ul6Free);
00060
00061 #if KERNEL_USE_TIMEOUTS
00062     // Binary semaphore is used to track any threads that are blocked on a
00063     // "send" due to lack of free slots.
00064     m_clSendSem.Init(0, 1);
00065 #endif
00066 }
00067
00068 //-----
00069 #if KERNEL_USE_AUTO_ALLOC
00070 Mailbox* Mailbox::Init( uint16_t ul6BufferSize_, uint16_t ul6ElementSize_ )
00071 {
00072     Mailbox* pclNew = (Mailbox*)AutoAlloc::Allocate(sizeof(Mailbox));
00073     void *pvBuffer = AutoAlloc::Allocate(ul6BufferSize_);
00074     pclNew->Init( pvBuffer, ul6BufferSize_, ul6ElementSize_ );
00075     return pclNew;
00076 }
00077 #endif
00078
00079 //-----
00080 void Mailbox::Receive( void *pvData_ )
00081 {
00082     KERNEL_ASSERT( pvData_ );
00083
00084 #if KERNEL_USE_TIMEOUTS
00085     Receive_i( pvData_, false, 0 );
00086 #else
00087     Receive_i( pvData_, false );
00088 #endif
00089 }
00090
00091 #if KERNEL_USE_TIMEOUTS
00092 //-----
00093 bool Mailbox::Receive( void *pvData_, uint32_t u32TimeoutMS_ )
00094 {
00095     KERNEL_ASSERT( pvData_ );
00096     return Receive_i( pvData_, false, u32TimeoutMS_ );
00097 }
00098 #endif
00099
00100 //-----
00101 void Mailbox::ReceiveTail( void *pvData_ )
00102 {
00103     KERNEL_ASSERT( pvData_ );
00104
00105 #if KERNEL_USE_TIMEOUTS
00106     Receive_i( pvData_, true, 0 );
00107 #else
00108     Receive_i( pvData_, true );
00109 #endif
00110 }
00111
00112 #if KERNEL_USE_TIMEOUTS
00113 //-----
00114 bool Mailbox::ReceiveTail( void *pvData_, uint32_t u32TimeoutMS_ )
00115 {
00116     KERNEL_ASSERT( pvData_ );
00117     return Receive_i( pvData_, true, u32TimeoutMS_ );
00118 }
00119 #endif
00120
00121 //-----
00122 bool Mailbox::Send( void *pvData_ )
00123 {
00124     KERNEL_ASSERT( pvData_ );
00125
00126 #if KERNEL_USE_TIMEOUTS
00127     return Send_i( pvData_, false, 0 );
00128 #else

```



```

00129     return Send_i( pvData_, false );
00130 #endif
00131 }
00132
00133 //-----
00134 bool Mailbox::SendTail( void *pvData_ )
00135 {
00136     KERNEL_ASSERT( pvData_ );
00137
00138 #if KERNEL_USE_TIMEOUTS
00139     return Send_i( pvData_, true, 0 );
00140 #else
00141     return Send_i( pvData_, true );
00142 #endif
00143 }
00144
00145 #if KERNEL_USE_TIMEOUTS
00146 //-----
00147 bool Mailbox::Send( void *pvData_, uint32_t u32TimeoutMS_ )
00148 {
00149     KERNEL_ASSERT( pvData_ );
00150
00151     return Send_i( pvData_, false, u32TimeoutMS_ );
00152 }
00153
00154 //-----
00155 bool Mailbox::SendTail( void *pvData_, uint32_t u32TimeoutMS_ )
00156 {
00157     KERNEL_ASSERT( pvData_ );
00158
00159     return Send_i( pvData_, true, u32TimeoutMS_ );
00160 }
00161 #endif
00162
00163 //-----
00164 #if KERNEL_USE_TIMEOUTS
00165 bool Mailbox::Send_i( const void *pvData_, bool bTail_, uint32_t u32TimeoutMS_ )
00166 #else
00167 bool Mailbox::Send_i( const void *pvData_, bool bTail_ )
00168 #endif
00169 {
00170     const void *pvDst;
00171
00172     bool bRet = false;
00173     bool bSchedState = Scheduler::SetScheduler( false );
00174
00175 #if KERNEL_USE_TIMEOUTS
00176     bool bBlock = false;
00177     bool bDone = false;
00178     while ( !bDone )
00179     {
00180         // Try to claim a slot first before resorting to blocking.
00181         if ( bBlock )
00182         {
00183             bDone = true;
00184             Scheduler::SetScheduler( bSchedState );
00185             m_clSendSem.Pend( u32TimeoutMS_ );
00186             Scheduler::SetScheduler( false );
00187         }
00188 #endif
00189
00190         CS_ENTER();
00191         // Ensure we have a free slot before we attempt to write data
00192         if ( m_ul6Free )
00193         {
00194             m_ul6Free--;
00195
00196             if ( bTail_ )
00197             {
00198                 pvDst = GetTailPointer();
00199                 MoveTailBackward();
00200             }
00201             else
00202             {
00203                 MoveHeadForward();
00204                 pvDst = GetHeadPointer();
00205             }
00206             bRet = true;
00207 #if KERNEL_USE_TIMEOUTS
00208             bDone = true;
00209 #endif
00210         }
00211
00212 #if KERNEL_USE_TIMEOUTS
00213         else if ( u32TimeoutMS_ )
00214         {
00215             bBlock = true;

```

```

00216         }
00217         else
00218         {
00219             bDone = true;
00220         }
00221     #endif
00222     CS_EXIT();
00223
00224     #if KERNEL_USE_TIMEOUTS
00225     }
00226     #endif
00227
00228     // Copy data to the claimed slot, and post the counting semaphore
00229     if (bRet)
00230     {
00231         CopyData( pvData_, pvDst, m_ul6ElementSize );
00232     }
00233
00234     Scheduler::SetScheduler( bSchedState );
00235
00236     if (bRet)
00237     {
00238         m_clRecvSem.Post();
00239     }
00240
00241     return bRet;
00242 }
00243
00244 //-----
00245 #if KERNEL_USE_TIMEOUTS
00246 bool Mailbox::Receive_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ )
00247 #else
00248 void Mailbox::Receive_i( const void *pvData_, bool bTail_ )
00249 #endif
00250 {
00251     const void *pvSrc;
00252
00253     #if KERNEL_USE_TIMEOUTS
00254     if (!m_clRecvSem.Pend( u32WaitTimeMS_ ))
00255     {
00256         // Failed to get the notification from the counting semaphore in the
00257         // time allotted. Bail.
00258         return false;
00259     }
00260     #else
00261     m_clRecvSem.Pend();
00262     #endif
00263
00264     // Disable the scheduler while we do this -- this ensures we don't have
00265     // multiple concurrent readers off the same queue, which could be problematic
00266     // if multiple writes occur during reads, etc.
00267     bool bSchedState = Scheduler::SetScheduler( false );
00268
00269     // Update the head/tail indexes, and get the associated data pointer for
00270     // the read operation.
00271     CS_ENTER();
00272
00273     m_ul6Free++;
00274     if (bTail_)
00275     {
00276         MoveTailForward();
00277         pvSrc = GetTailPointer();
00278     }
00279     else
00280     {
00281         pvSrc = GetHeadPointer();
00282         MoveHeadBackward();
00283     }
00284
00285     CS_EXIT();
00286
00287     CopyData( pvSrc, pvData_, m_ul6ElementSize );
00288
00289     Scheduler::SetScheduler( bSchedState );
00290
00291     // Unblock a thread waiting for a free slot to send to
00292     m_clSendSem.Post();
00293
00294     #if KERNEL_USE_TIMEOUTS
00295     return true;
00296     #endif
00297 }
00298
00299 #endif
00300

```

18.37 /home/mo/mark3-source/embedded/kernel/message.cpp File Reference

Inter-thread communications via message passing.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "message.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

18.37.1 Detailed Description

Inter-thread communications via message passing.

Definition in file [message.cpp](#).

18.38 message.cpp

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "message.h"
00026 #include "threadport.h"
00027
00028 #define _CAN_HAS_DEBUG
00029 //--[Autogenerated - Do Not Modify]-----
00030 #include "dbg_file_list.h"
00031 #include "buffalogger.h"
00032 #if defined(DBG_FILE)
00033 # error "Debug logging file token already defined! Bailing."
00034 #else
00035 # define DBG_FILE _DBG__KERNEL_MESSAGE_CPP
00036 #endif
00037 //--[End Autogenerated content]-----
00038 #include "kerneldebug.h"
00039
00040 #if KERNEL_USE_MESSAGE
00041
00042 #if KERNEL_USE_TIMEOUTS
00043 #include "timerlist.h"
00044 #endif
00045
00046 Message GlobalMessagePool::m_aclMessagePool[
    GLOBAL_MESSAGE_POOL_SIZE];
00047 DoubleLinkedList GlobalMessagePool::m_clList;
00048
00049 //-----
00050 void GlobalMessagePool::Init()
00051 {
00052     uint8_t i;
00053     GlobalMessagePool::m_clList.Init();
00054     for (i = 0; i < GLOBAL_MESSAGE_POOL_SIZE; i++)
00055     {
00056         GlobalMessagePool::m_aclMessagePool[i].Init();
00057         GlobalMessagePool::m_clList.Add(&(GlobalMessagePool::m_aclMessagePool[i]));
00058     }
00059 }
```

```

00060
00061 //-----
00062 void GlobalMessagePool::Push( Message *pclMessage_ )
00063 {
00064     KERNEL_ASSERT( pclMessage_ );
00065
00066     CS_ENTER();
00067
00068     GlobalMessagePool::m_clList.Add(pclMessage_);
00069
00070     CS_EXIT();
00071 }
00072
00073 //-----
00074 Message *GlobalMessagePool::Pop()
00075 {
00076     Message *pclRet;
00077     CS_ENTER();
00078
00079     pclRet = static_cast<Message*>( GlobalMessagePool::m_clList.GetHead() );
00080     if ( 0 != pclRet )
00081     {
00082         GlobalMessagePool::m_clList.Remove( static_cast<LinkListNode*>( pclRet ) );
00083     }
00084
00085     CS_EXIT();
00086     return pclRet;
00087 }
00088
00089 //-----
00090 void MessageQueue::Init()
00091 {
00092     m_clSemaphore.Init(0, GLOBAL_MESSAGE_POOL_SIZE);
00093 }
00094
00095 //-----
00096 Message *MessageQueue::Receive()
00097 {
00098     #if KERNEL_USE_TIMEOUTS
00099         return Receive_i(0);
00100     #else
00101         return Receive_i();
00102     #endif
00103 }
00104
00105 //-----
00106 #if KERNEL_USE_TIMEOUTS
00107 Message *MessageQueue::Receive( uint32_t u32TimeWaitMS_ )
00108 {
00109     return Receive_i( u32TimeWaitMS_ );
00110 }
00111 #endif
00112
00113 //-----
00114 #if KERNEL_USE_TIMEOUTS
00115 Message *MessageQueue::Receive_i( uint32_t u32TimeWaitMS_ )
00116 #else
00117 Message *MessageQueue::Receive_i( void )
00118 #endif
00119 {
00120     Message *pclRet;
00121
00122     // Block the current thread on the counting semaphore
00123     #if KERNEL_USE_TIMEOUTS
00124         if (!m_clSemaphore.Pend(u32TimeWaitMS_))
00125         {
00126             return NULL;
00127         }
00128     #else
00129         m_clSemaphore.Pend();
00130     #endif
00131
00132     CS_ENTER();
00133
00134     // Pop the head of the message queue and return it
00135     pclRet = static_cast<Message*>( m_clLinkList.GetHead() );
00136     m_clLinkList.Remove(static_cast<Message*>(pclRet));
00137
00138     CS_EXIT();
00139
00140     return pclRet;
00141 }
00142
00143 //-----
00144 void MessageQueue::Send( Message *pclSrc_ )
00145 {
00146     KERNEL_ASSERT( pclSrc_ );

```

```

00147
00148     CS_ENTER();
00149
00150     // Add the message to the head of the linked list
00151     m_clLinkedList.Add( pclSrc_ );
00152
00153     // Post the semaphore, waking the blocking thread for the queue.
00154     m_clSemaphore.Post();
00155
00156     CS_EXIT();
00157 }
00158
00159 //-----
00160 uint16_t MessageQueue::GetCount()
00161 {
00162     return m_clSemaphore.GetCount();
00163 }
00164 #endif //KERNEL_USE_MESSAGE

```

18.39 /home/mo/mark3-source/embedded/kernel/mutex.cpp File Reference

Mutual-exclusion object.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "mutex.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

Functions

- void [TimedMutex_Callback](#) (Thread *pclOwner_, void *pvData_)
TimedMutex_Callback.

18.39.1 Detailed Description

Mutual-exclusion object.

Definition in file [mutex.cpp](#).

18.39.2 Function Documentation

18.39.2.1 void TimedMutex_Callback (Thread * pclOwner_, void * pvData_)

TimedMutex_Callback.

This function is called from the timer-expired context to trigger a timeout on this mutex. This results in the waking of the thread that generated the mutex claim call that was not completed in time.

Parameters

<i>pclOwner_</i>	Pointer to the thread to wake
<i>pvData_</i>	Pointer to the mutex object that the thread is blocked on

Definition at line 54 of file [mutex.cpp](#).

18.40 mutex.cpp

```

00001  /*=====
00002
00003  _____
00004  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00005  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00006  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00007  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00020  #include "kerneltypes.h"
00021  #include "mark3cfg.h"
00022
00023  #include "blocking.h"
00024  #include "mutex.h"
00025
00026  #define _CAN_HAS_DEBUG
00027  //--[Autogenerated - Do Not Modify]-----
00028  #include "dbg_file_list.h"
00029  #include "buffalogger.h"
00030  #if defined(DBG_FILE)
00031  # error "Debug logging file token already defined!  Bailing."
00032  #else
00033  # define DBG_FILE _DBG__KERNEL_MUTEX_CPP
00034  #endif
00035  //--[End Autogenerated content]-----
00036
00037  #include "kerneldebug.h"
00038
00039  #if KERNEL_USE_MUTEX
00040
00041  #if KERNEL_USE_TIMEOUTS
00042
00043  //-----
00054  void TimedMutex_Callback(Thread *pclOwner_, void *pvData_)
00055  {
00056      Mutex *pclMutex = static_cast<Mutex*>(pvData_);
00057
00058      // Indicate that the semaphore has expired on the thread
00059      pclOwner_>SetExpired(true);
00060
00061      // Wake up the thread that was blocked on this semaphore.
00062      pclMutex->WakeMe(pclOwner_);
00063
00064      if (pclOwner_>GetCurPriority() >= Scheduler::GetCurrentThread
00065          ()->GetCurPriority())
00066      {
00067          Thread::Yield();
00068      }
00069  }
00070  //-----
00071  void Mutex::WakeMe(Thread *pclOwner_)
00072  {
00073      // Remove from the semaphore waitlist and back to its ready list.
00074      Unblock(pclOwner_);
00075  }
00076
00077  #endif
00078
00079  //-----
00080  uint8_t Mutex::WakeNext()
00081  {
00082      Thread *pclChosenOne = NULL;
00083
00084      // Get the highest priority waiter thread
00085      pclChosenOne = m_clBlockList.HighestWaiter();
00086
00087      // Unblock the thread
00088      Unblock(pclChosenOne);
00089
00090      // The chosen one now owns the mutex
00091      m_pclOwner = pclChosenOne;
00092
00093      // Signal a context switch if it's a greater than or equal to the current priority
00094      if (pclChosenOne->GetCurPriority() >=
00095          Scheduler::GetCurrentThread()->GetCurPriority())
00096      {
00097          return 1;
00098      }
00099      return 0;

```

```

00099 }
00100
00101 //-----
00102 void Mutex::Init()
00103 {
00104     // Reset the data in the mutex
00105     m_bReady = 1;           // The mutex is free.
00106     m_u8MaxPri = 0;         // Set the maximum priority inheritance state
00107     m_pclOwner = NULL;      // Clear the mutex owner
00108     m_u8Recurse = 0;        // Reset recurse count
00109 }
00110
00111 //-----
00112 #if KERNEL_USE_TIMEOUTS
00113 bool Mutex::Claim_i(uint32_t u32WaitTimeMS_)
00114 #else
00115 void Mutex::Claim_i(void)
00116 #endif
00117 {
00118     KERNEL_TRACE_1( "Claiming Mutex, Thread %d", (uint16_t)
00119         g_pclCurrent->GetID() );
00120 #if KERNEL_USE_TIMEOUTS
00121     Timer clTimer;
00122     bool bUseTimer = false;
00123 #endif
00124
00125     // Disable the scheduler while claiming the mutex - we're dealing with all
00126     // sorts of private thread data, can't have a thread switch while messing
00127     // with internal data structures.
00128     Scheduler::SetScheduler(0);
00129
00130     // Check to see if the mutex is claimed or not
00131     if (m_bReady != 0)
00132     {
00133         // Mutex isn't claimed, claim it.
00134         m_bReady = 0;
00135         m_u8Recurse = 0;
00136         m_u8MaxPri = g_pclCurrent->GetPriority();
00137         m_pclOwner = g_pclCurrent;
00138
00139         Scheduler::SetScheduler(1);
00140
00141         #if KERNEL_USE_TIMEOUTS
00142             return true;
00143         #else
00144             return;
00145         #endif
00146     }
00147
00148     // If the mutex is already claimed, check to see if this is the owner thread,
00149     // since we allow the mutex to be claimed recursively.
00150     if (g_pclCurrent == m_pclOwner)
00151     {
00152         // Ensure that we haven't exceeded the maximum recursive-lock count
00153         KERNEL_ASSERT( (m_u8Recurse < 255) );
00154         m_u8Recurse++;
00155
00156         // Increment the lock count and bail
00157         Scheduler::SetScheduler(1);
00158         #if KERNEL_USE_TIMEOUTS
00159             return true;
00160         #else
00161             return;
00162         #endif
00163     }
00164
00165     // The mutex is claimed already - we have to block now. Move the
00166     // current thread to the list of threads waiting on the mutex.
00167     #if KERNEL_USE_TIMEOUTS
00168         if (u32WaitTimeMS_)
00169         {
00170             g_pclCurrent->SetExpired(false);
00171             clTimer.Init();
00172             clTimer.Start(0, u32WaitTimeMS_, (TimerCallback_t)
00173                 TimedMutex_Callback, (void*)this);
00174             bUseTimer = true;
00175         }
00176         #endif
00177     BlockPriority(g_pclCurrent);
00178
00179     // Check if priority inheritance is necessary. We do this in order
00180     // to ensure that we don't end up with priority inversions in case
00181     // multiple threads are waiting on the same resource.
00182     if (m_u8MaxPri <= g_pclCurrent->GetPriority())
00183     {
00184         m_u8MaxPri = g_pclCurrent->GetPriority();
00185     }

```

```

00184
00185     Thread *pclTemp = static_cast<Thread*>(m_clBlockList.GetHead());
00186     while(pclTemp)
00187     {
00188         pclTemp->InheritPriority(m_u8MaxPri);
00189         if(pclTemp == static_cast<Thread*>(m_clBlockList.GetTail()) )
00190         {
00191             break;
00192         }
00193         pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00194     }
00195     m_pclOwner->InheritPriority(m_u8MaxPri);
00196 }
00197
00198 // Done with thread data -reenable the scheduler
00199 Scheduler::SetScheduler(1);
00200
00201 // Switch threads if this thread acquired the mutex
00202 Thread::Yield();
00203
00204 #if KERNEL_USE_TIMEOUTS
00205     if (bUseTimer)
00206     {
00207         clTimer.Stop();
00208         return (g_pclCurrent->GetExpired() == 0);
00209     }
00210     return true;
00211 #endif
00212 }
00213
00214 //-----
00215 void Mutex::Claim(void)
00216 {
00217     #if KERNEL_USE_TIMEOUTS
00218         Claim_i(0);
00219     #else
00220         Claim_i();
00221     #endif
00222 }
00223
00224 //-----
00225 #if KERNEL_USE_TIMEOUTS
00226 bool Mutex::Claim(uint32_t u32WaitTimeMS_)
00227 {
00228     return Claim_i(u32WaitTimeMS_);
00229 }
00230 #endif
00231
00232 //-----
00233 void Mutex::Release()
00234 {
00235     KERNEL_TRACE_1( "Releasing Mutex, Thread %d", (uint16_t)
g_pclCurrent->GetID() );
00236
00237     bool bSchedule = 0;
00238
00239     // Disable the scheduler while we deal with internal data structures.
00240     Scheduler::SetScheduler(0);
00241
00242     // This thread had better be the one that owns the mutex currently...
00243     KERNEL_ASSERT( (g_pclCurrent == m_pclOwner) );
00244
00245     // If the owner had claimed the lock multiple times, decrease the lock
00246     // count and return immediately.
00247     if (m_u8Recurse)
00248     {
00249         m_u8Recurse--;
00250         Scheduler::SetScheduler(1);
00251         return;
00252     }
00253
00254     // Restore the thread's original priority
00255     if (g_pclCurrent->GetCurPriority() != g_pclCurrent->
GetPriority())
00256     {
00257         g_pclCurrent->SetPriority(g_pclCurrent->
GetPriority());
00258
00259         // In this case, we want to reschedule
00260         bSchedule = 1;
00261     }
00262
00263     // No threads are waiting on this semaphore?
00264     if (m_clBlockList.GetHead() == NULL)
00265     {
00266         // Re-initialize the mutex to its default values
00267         m_bReady = 1;

```



```
00268     m_u8MaxPri = 0;
00269     m_pclOwner = NULL;
00270 }
00271 else
00272 {
00273     // Wake the highest priority Thread pending on the mutex
00274     if(WakeNext())
00275     {
00276         // Switch threads if it's higher or equal priority than the current thread
00277         bSchedule = 1;
00278     }
00279 }
00280
00281 // Must enable the scheduler again in order to switch threads.
00282 Scheduler::SetScheduler(1);
00283 if(bSchedule)
00284 {
00285     // Switch threads if a higher-priority thread was woken
00286     Thread::Yield();
00287 }
00288 }
00289
00290 #endif //KERNEL_USE_MUTEX
```

18.41 /home/mo/mark3-source/embedded/kernel/notify.cpp File Reference

Lightweight thread notification - blocking object.

```
#include "mark3cfg.h"
#include "notify.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
```

18.41.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file [notify.cpp](#).

18.42 notify.cpp

[illegible]

```

00041     Notify *pclNotify = static_cast<Notify*>(pvData_);
00042
00043     // Indicate that the semaphore has expired on the thread
00044     pclOwner_>SetExpired(true);
00045
00046     // Wake up the thread that was blocked on this semaphore.
00047     pclNotify->WakeMe(pclOwner_);
00048
00049     if (pclOwner_>GetCurPriority() >= Scheduler::GetCurrentThread
00050         ()->GetCurPriority())
00051     {
00052         Thread::Yield();
00053     }
00054
00055 //-----
00056 void Notify::Init(void)
00057 {
00058     m_clBlockList.Init();
00059 }
00060
00061 //-----
00062 void Notify::Signal(void)
00063 {
00064     bool bReschedule = false;
00065
00066     CS_ENTER();
00067     Thread *pclCurrent = (Thread*)m_clBlockList.GetHead();
00068     while (pclCurrent != NULL)
00069     {
00070         Unblock(pclCurrent);
00071         if ( !bReschedule &&
00072             ( pclCurrent->GetCurPriority() >=
00073               Scheduler::GetCurrentThread()->GetCurPriority() ) )
00074         {
00075             bReschedule = true;
00076         }
00077         pclCurrent = (Thread*)m_clBlockList.GetHead();
00078     }
00079     CS_EXIT();
00080
00081     if (bReschedule)
00082     {
00083         Thread::Yield();
00084     }
00085 }
00086 //-----
00087 void Notify::Wait( bool *pbFlag_ )
00088 {
00089     CS_ENTER();
00090     Block(g_pclCurrent);
00091     if (pbFlag_)
00092     {
00093         *pbFlag_ = false;
00094     }
00095     CS_EXIT();
00096
00097     Thread::Yield();
00098     if (pbFlag_)
00099     {
00100         *pbFlag_ = true;
00101     }
00102 }
00103
00104 //-----
00105 #if KERNEL_USE_TIMEOUTS
00106 bool Notify::Wait( uint32_t u32WaitTimeMS_, bool *pbFlag_ )
00107 {
00108     bool bUseTimer = false;
00109     Timer clNotifyTimer;
00110
00111     CS_ENTER();
00112     if (u32WaitTimeMS_)
00113     {
00114         bUseTimer = true;
00115         g_pclCurrent->SetExpired(false);
00116
00117         clNotifyTimer.Init();
00118         clNotifyTimer.Start(0, u32WaitTimeMS_, TimedNotify_Callback, (void*)this);
00119     }
00120
00121     Block(g_pclCurrent);
00122
00123     if (pbFlag_)
00124     {
00125         *pbFlag_ = false;

```

```
00126     }
00127     CS_EXIT();
00128
00129     Thread::Yield();
00130
00131     if (bUseTimer)
00132     {
00133         clNotifyTimer.Stop();
00134         return (g_pclCurrent->GetExpired() == 0);
00135     }
00136
00137     if (pbFlag_)
00138     {
00139         *pbFlag_ = true;
00140     }
00141
00142     return true;
00143 }
00144 #endif
00145 //-----
00146 void Notify::WakeMe(Thread *pclChosenOne_)
00147 {
00148     Unblock(pclChosenOne_);
00149 }
00150
00151 #endif
```

18.43 /home/mo/mark3-source/embedded/kernel/profile.cpp File Reference

Code profiling utilities.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kernelprofile.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

18.43.1 Detailed Description

Code profiling utilities.

Definition in file [profile.cpp](#).

18.44 profile.cpp

```
00001 /*=====
```

```
00002
```

```
00003      _____   _____   _____   _____
```

```
00004 |         / \       / \       / \       / \       |
```

```
00005 |        /   \     /   \     /   \     /   \     |
```

```
00006 |_____/_____/_\_____/_____/_\_____/_____/_\___|
```

```
00007
```

```
00008
```

```
00009 --[Mark3 Realtime Platform]-----
```

```
00010
```

```
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
```

```
00012 See license.txt for more information
```

```
00013 =====*/
```

```
00021 #include "kerneltypes.h"
```

```
00022 #include "mark3cfg.h"
```

```
00023 #include "profile.h"
```

```
00024 #include "kernelprofile.h"
```

```
00025 #include "threadport.h"
```

```
00026
```

```
00027 #define _CAN_HAS_DEBUG
```

```
00028 //--[Autogenerated - Do Not Modify]-----
```

```
00029 #include "dbg_file_list.h"
```

```

00030 #include "buffalogger.h"
00031 #if defined(DBG_FILE)
00032 # error "Debug logging file token already defined!  Bailing."
00033 #else
00034 # define DBG_FILE _DBG__KERNEL_PROFILE_CPP
00035 #endif
00036 //--[End Autogenerated content]-----
00037
00038 #include "kerneldebug.h"
00039
00040 #if KERNEL_USE_PROFILER
00041
00042 //-----
00043 void ProfileTimer::Init()
00044 {
00045     m_u32Cumulative = 0;
00046     m_u32CurrentIteration = 0;
00047     m_ul6Iterations = 0;
00048     m_bActive = 0;
00049 }
00050
00051 //-----
00052 void ProfileTimer::Start()
00053 {
00054     if (!m_bActive)
00055     {
00056         CS_ENTER();
00057         m_u32CurrentIteration = 0;
00058         m_u32InitialEpoch = Profiler::GetEpoch();
00059         m_ul6Initial = Profiler::Read();
00060         CS_EXIT();
00061         m_bActive = 1;
00062     }
00063 }
00064
00065 //-----
00066 void ProfileTimer::Stop()
00067 {
00068     if (m_bActive)
00069     {
00070         uint16_t ul6Final;
00071         uint32_t u32Epoch;
00072         CS_ENTER();
00073         ul6Final = Profiler::Read();
00074         u32Epoch = Profiler::GetEpoch();
00075         // Compute total for current iteration...
00076         m_u32CurrentIteration = ComputeCurrentTicks(ul6Final,
00077             u32Epoch);
00077         m_u32Cumulative += m_u32CurrentIteration;
00078         m_ul6Iterations++;
00079         CS_EXIT();
00080         m_bActive = 0;
00081     }
00082 }
00083
00084 //-----
00085 uint32_t ProfileTimer::GetAverage()
00086 {
00087     if (m_ul6Iterations)
00088     {
00089         return m_u32Cumulative / (uint32_t)m_ul6Iterations;
00090     }
00091     return 0;
00092 }
00093
00094 //-----
00095 uint32_t ProfileTimer::GetCurrent()
00096 {
00097     if (m_bActive)
00098     {
00099         uint16_t ul6Current;
00100         uint32_t u32Epoch;
00101         CS_ENTER();
00102         ul6Current = Profiler::Read();
00103         u32Epoch = Profiler::GetEpoch();
00104         CS_EXIT();
00105         return ComputeCurrentTicks(ul6Current, u32Epoch);
00106     }
00107     return m_u32CurrentIteration;
00108 }
00109
00110 //-----
00111 uint32_t ProfileTimer::ComputeCurrentTicks(uint16_t ul6Current_, uint32_t
00112     u32Epoch_)
00113 {
00114     uint32_t u32Total;

```

```

00115 uint32_t u32Overflows;
00116
00117 u32Overflows = u32Epoch - m_u32InitialEpoch;
00118
00119 // More than one overflow...
00120 if (u32Overflows > 1)
00121 {
00122     u32Total = ((uint32_t)(u32Overflows-1) * TICKS_PER_OVERFLOW)
00123               + (uint32_t)(TICKS_PER_OVERFLOW - m_ul6Initial) +
00124               (uint32_t)ul6Current_;
00125 }
00126 // Only one overflow, or one overflow that has yet to be processed
00127 else if (u32Overflows || (ul6Current_ < m_ul6Initial))
00128 {
00129     u32Total = (uint32_t)(TICKS_PER_OVERFLOW - m_ul6Initial) +
00130               (uint32_t)ul6Current_;
00131 }
00132 // No overflows, none pending.
00133 else
00134 {
00135     u32Total = (uint32_t)(ul6Current_ - m_ul6Initial);
00136 }
00137
00138 return u32Total;
00139 }
00140
00141 #endif

```

18.45 /home/mo/mark3-source/embedded/kernel/public/atomic.h File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "threadport.h"
```

18.45.1 Detailed Description

Basic Atomic Operations.

Definition in file [atomic.h](#).

18.46 atomic.h

```

00001 /*-----*/
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #ifndef __ATOMIC_H__
00022 #define __ATOMIC_H__
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026 #include "threadport.h"
00027
00028 #if KERNEL_USE_ATOMIC
00029
00039 class Atomic
00040 {
00041 public:
00048     static uint8_t Set( uint8_t *pu8Source_, uint8_t u8Val_ );
00049     static uint16_t Set( uint16_t *pu16Source_, uint16_t u16Val_ );
00050     static uint32_t Set( uint32_t *pu32Source_, uint32_t u32Val_ );

```


18.49 /home/mo/mark3-source/embedded/kernel/public/blocking.h File Reference

Blocking object base class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
```

Classes

- class [BlockingObject](#)
Class implementing thread-blocking primitives.

18.49.1 Detailed Description

Blocking object base class declarations.

A Blocking object in Mark3 is essentially a thread list. Any blocking object implementation (being a semaphore, mutex, event flag, etc.) can be built on top of this class, utilizing the provided functions to manipulate thread location within the [Kernel](#).

Blocking a thread results in that thread becoming de-scheduled, placed in the blocking object's own private list of threads which are waiting on the object.

Unblocking a thread results in the reverse: The thread is moved back to its original location from the blocking list.

The only difference between a blocking object based on this class is the logic used to determine what constitutes a Block or Unblock condition.

For instance, a semaphore Pend operation may result in a call to the Block() method with the currently-executing thread in order to make that thread wait for a semaphore Post. That operation would then invoke the Unblock() method, removing the blocking thread from the semaphore's list, and back into the appropriate thread inside the scheduler.

Care must be taken when implementing blocking objects to ensure that critical sections are used judiciously, otherwise asynchronous events like timers and interrupts could result in non-deterministic and often catastrophic behavior.

Definition in file [blocking.h](#).

18.50 blocking.h

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00047 #ifndef __BLOCKING_H__
00048 #define __BLOCKING_H__
00049
00050 #include "kerneltypes.h"
00051 #include "mark3cfg.h"
00052
00053 #include "ll.h"
```

```

00054 #include "threadlist.h"
00055 #include "thread.h"
00056
00057 #if KERNEL_USE_MUTEX || KERNEL_USE_SEMAPHORE || KERNEL_USE_EVENTFLAG
00058
00059 //-----
00065 class BlockingObject
00066 {
00067 protected:
00088     void Block(Thread *pclThread_ );
00089
00098     void BlockPriority(Thread *pclThread_ );
00099
00111     void Unblock(Thread *pclThread_);
00112
00117     ThreadList m_clBlockList;
00118 };
00119
00120 #endif
00121
00122 #endif

```

18.51 /home/mo/mark3-source/embedded/kernel/public/buffalogger.h File Reference

Super-efficient, super-secure logging routines.

```
#include <stdint.h>
```

18.51.1 Detailed Description

Super-efficient, super-secure logging routines.

Uses offline processing to ensure performance.

Definition in file [buffalogger.h](#).

18.52 buffalogger.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00020 #pragma once
00021 #include <stdint.h>
00022
00023 //-----
00024 #define STR1(s) #s
00025 #define STR(s) STR1(s)
00026
00027 //-----
00028 #define EMIT_DBG_STRING(str) \
00029 do { \
00030     const static volatile char log_str[] __attribute__((section (".logger"))) __attribute__((unused)) = \
00031     str; \
00032     const static volatile uint16_t line_id __attribute__((section (".logger"))) __attribute__((unused)) = \
00033     _LINE_; \
00034     const static volatile uint16_t file_id __attribute__((section (".logger"))) __attribute__((unused)) = \
00035     DBG_FILE; \
00036     const static volatile uint16_t sync __attribute__((section (".logger"))) __attribute__((unused)) = \
00037     0xCAFE; \
00038 } while(0);
00039
00040

```


18.53 /home/mo/mark3-source/embedded/kernel/public/driver.h File Reference

Driver abstraction framework.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

Classes

- class **Driver**
Base device-driver class used in hardware abstraction.
- class **DriverList**
*List of **Driver** objects used to keep track of all device drivers in the system.*

18.53.1 Detailed Description

Driver abstraction framework.

18.53.2 Intro

This is the basis of the driver framework. In the context of Mark3, drivers don't necessarily have to be based on physical hardware peripherals. They can be used to represent algorithms (such as random number generators), files, or protocol stacks. Unlike FunkOS, where driver IO is protected automatically by a mutex, we do not use this kind of protection - we leave it up to the driver implementor to do what's right in its own context. This also frees up the driver to implement all sorts of other neat stuff, like sending messages to threads associated with the driver. Drivers are implemented as character devices, with the standard array of posix-style accessor methods for reading, writing, and general driver control.

A global driver list is provided as a convenient and minimal "filesystem" structure, in which devices can be accessed by name.

18.53.3 Driver Design

A device driver needs to be able to perform the following operations: -Initialize a peripheral -Start/stop a peripheral -Handle I/O control operations -Perform various read/write operations

At the end of the day, that's pretty much all a device driver has to do, and all of the functionality that needs to be presented to the developer.

We abstract all device drivers using a base-class which implements the following methods: -Start/Open -Stop/Close -Control -Read -Write

A basic driver framework and API can thus be implemented in five function calls - that's it! You could even reduce that further by handling the initialize, start, and stop operations inside the "control" operation.

18.53.4 Driver API

In C++, we can implement this as a class to abstract these event handlers, with virtual void functions in the base class overridden by the inherited objects.

To add and remove device drivers from the global table, we use the following methods:

```
void DriverList::Add( Driver *pclDriver_ );
void DriverList::Remove( Driver *pclDriver_ );
```

`DriverList::Add()/Remove()` takes a single arguments the pointer to the object to operate on.

Once a driver has been added to the table, drivers are opened by NAME using `DriverList::FindBy<Name>("/dev/name")`. This function returns a pointer to the specified driver if successful, or to a built in /dev/null device if the path name is invalid. After a driver is open, that pointer is used for all other driver access functions.

This abstraction is incredibly useful any peripheral or service can be accessed through a consistent set of APIs, that make it easy to substitute implementations from one platform to another. Portability is ensured, the overhead is negligible, and it emphasizes the reuse of both driver and application code as separate entities.

Consider a system with drivers for I2C, SPI, and UART peripherals - under our driver framework, an application can initialize these peripherals and write a greeting to each using the same simple API functions for all drivers:

```
pcI2C = DriverList::FindByName("/dev/i2c");
pcUART = DriverList::FindByName("/dev/tty0");
pcSPI = DriverList::FindByName("/dev/spi");

pcI2C->Write(12, "Hello World!");
pcUART->Write(12, "Hello World!");
pcSPI->Write(12, "Hello World!");
```

Definition in file [driver.h](#).

18.54 driver.h

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00105 #include "kerneltypes.h"
00106 #include "mark3cfg.h"
00107
00108 #include "ll.h"
00109
00110 #ifndef __DRIVER_H__
00111 #define __DRIVER_H__
00112
00113 #if KERNEL_USE_DRIVER
00114
00115 class DriverList;
00116 //-----
00121 class Driver : public LinkListNode
00122 {
00123 public:
00129     virtual void Init() = 0;
00130
00138     virtual uint8_t Open() = 0;
00139
00147     virtual uint8_t Close() = 0;
00148
00163     virtual uint16_t Read( uint16_t u16Bytes_,
00164                           uint8_t *pu8Data_) = 0;
00165
00181     virtual uint16_t Write( uint16_t u16Bytes_,
00182                             uint8_t *pu8Data_) = 0;
00183
00202     virtual uint16_t Control( uint16_t u16Event_,
00203                               void *pvDataIn_,
00204                               uint16_t u16SizeIn_,
00205                               void *pvDataOut_,
00206                               uint16_t u16SizeOut_ ) = 0;
00207
00216     void SetName( const char *pcName_ ) { m_pcPath = pcName_; }
00217
00225     const char *GetPath() { return m_pcPath; }
00226
00227 private:
00228
00230     const char *m_pcPath;
```

```
00231 };
00232
00233 //-----
00238 class DriverList
00239 {
00240 public:
00241     static void Init();
00242
00243     static void Add( Driver *pclDriver_ ) { m_clDriverList.
00244 Add(pclDriver_); }
00245
00246     static void Remove( Driver *pclDriver_ ) { m_clDriverList.
00247 Remove(pclDriver_); }
00248
00249     static Driver *FindByPath( const char *m_pcPath );
00250
00251 private:
00252
00253     static DoubleLinkedList m_clDriverList;
00254 };
00255
00256 #endif //KERNEL_USE_DRIVER
00257
00258 #endif
```

18.55 /home/mo/mark3-source/embedded/kernel/public/eventflag.h File Reference

Event Flag Blocking Object/IPC-Object definition.

```
#include "mark3cfg.h"
#include "kernel.h"
#include "kerneltypes.h"
#include "blocking.h"
#include "thread.h"
```

Classes

- class EventFlag

The `EventFlag` class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

18.55.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file [eventflag.h](#).

18.56 eventflag.h

[illegible]

```

00024 #include "kerneltypes.h"
00025 #include "blocking.h"
00026 #include "thread.h"
00027
00028 #if KERNEL_USE_EVENTFLAG
00029
00030 //-----
00046 class EventFlag : public BlockingObject
00047 {
00048 public:
00052     void Init() { m_ul6SetMask = 0; m_clBlockList.
        Init(); }
00053
00061     uint16_t Wait(uint16_t ul6Mask_, EventFlagOperation_t eMode_);
00062
00063 #if KERNEL_USE_TIMEOUTS
00064
00072     uint16_t Wait(uint16_t ul6Mask_, EventFlagOperation_t eMode_, uint32_t
        u32TimeMS_);
00073
00081     void WakeMe(Thread *pclOwner_);
00082
00083 #endif
00084
00090     void Set(uint16_t ul6Mask_);
00091
00096     void Clear(uint16_t ul6Mask_);
00097
00102     uint16_t GetMask();
00103
00104 private:
00105
00106 #if KERNEL_USE_TIMEOUTS
00107
00119     uint16_t Wait_i(uint16_t ul6Mask_, EventFlagOperation_t eMode_, uint32_t
        u32TimeMS_);
00120 #else
00121
00131     uint16_t Wait_i(uint16_t ul6Mask_, EventFlagOperation_t eMode_);
00132 #endif
00133
00134     uint16_t m_ul6SetMask;
00135 };
00136
00137 #endif //KERNEL_USE_EVENTFLAG
00138 #endif //__EVENTFLAG_H__
00139

```

18.57 /home/mo/mark3-source/embedded/kernel/public/kernel.h File Reference

[Kernel](#) initialization and startup class.

```

#include "mark3cfg.h"
#include "kerneltypes.h"
#include "paniccodes.h"
#include "thread.h"

```

Classes

- class [Kernel](#)

Class that encapsulates all of the kernel startup functions.

18.57.1 Detailed Description

[Kernel](#) initialization and startup class.

The [Kernel](#) namespace provides functions related to initializing and starting up the kernel.

The [Kernel::Init\(\)](#) function must be called before any of the other functions in the kernel can be used.

Once the initial kernel configuration has been completed (i.e. first threads have been added to the scheduler), the `Kernel::Start()` function can then be called, which will transition code execution from the "main()" context to the threads in the scheduler.

Definition in file [kernel.h](#).

18.58 kernel.h

```

00001 /*-----
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00032 #ifndef __KERNEL_H__
00033 #define __KERNEL_H__
00034
00035 #include "mark3cfg.h"
00036 #include "kerneltypes.h"
00037 #include "paniccodes.h"
00038 #include "thread.h"
00039
00040 #if KERNEL_USE_IDLE_FUNC
00041 typedef void (*idle_func_t)(void);
00042 #endif
00043
00044 //-----
00048 class Kernel
00049 {
00050 public:
00051     static void Init(void);
00052
00053     static void Start(void);
00054
00055     static bool IsStarted() { return m_bIsStarted; }
00056
00057     static void SetPanic( panic_func_t pfPanic_ ) {
00058         m_pfPanic = pfPanic_; }
00059
00060     static bool IsPanic() { return m_bIsPanic; }
00061
00062     static void Panic(uint16_t ul6Cause_);
00063
00064 #if KERNEL_USE_IDLE_FUNC
00065     static void SetIdleFunc( idle_func_t pfIdle_ ) { m_pfIdle = pfIdle_; }
00066
00067     static void IdleFunc(void) { if (m_pfIdle != 0){ m_pfIdle(); } }
00068
00069     static Thread *GetIdleThread(void) { return (Thread*)&
00070         m_clIdle; }
00071 #endif
00072
00073 private:
00074     static bool m_bIsStarted;
00075     static bool m_bIsPanic;
00076     static panic_func_t m_pfPanic;
00077 #if KERNEL_USE_IDLE_FUNC
00078     static idle_func_t m_pfIdle;
00079     static FakeThread_t m_clIdle;
00080 #endif
00081 };
00082
00083 #endif
00084
00085

```

18.59 /home/mo/mark3-source/embedded/kernel/public/kernelaware.h File Reference

Kernel aware simulation support.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
```

Classes

- class [KernelAware](#)

The [KernelAware](#) class.

Enumerations

- enum [KernelAwareCommand_t](#) {
[KA_COMMAND_IDLE](#) = 0, [KA_COMMAND_PROFILE_INIT](#), [KA_COMMAND_PROFILE_START](#), [KA_COMMAND_PROFILE_STOP](#),
[KA_COMMAND_PROFILE_REPORT](#), [KA_COMMAND_EXIT_SIMULATOR](#), [KA_COMMAND_TRACE_0](#),
[KA_COMMAND_TRACE_1](#),
[KA_COMMAND_TRACE_2](#), [KA_COMMAND_PRINT](#) }

This enumeration contains a list of supported commands that can be executed to invoke a response from a kernel aware host.

18.59.1 Detailed Description

[Kernel](#) aware simulation support.

Definition in file [kernelaware.h](#).

18.59.2 Enumeration Type Documentation

18.59.2.1 enum [KernelAwareCommand_t](#)

This enumeration contains a list of supported commands that can be executed to invoke a response from a kernel aware host.

Enumerator

[KA_COMMAND_IDLE](#) Null command, does nothing.

[KA_COMMAND_PROFILE_INIT](#) Initialize a new profiling session.

[KA_COMMAND_PROFILE_START](#) Begin a profiling sample.

[KA_COMMAND_PROFILE_STOP](#) End a profiling sample.

[KA_COMMAND_PROFILE_REPORT](#) Report current profiling session.

[KA_COMMAND_EXIT_SIMULATOR](#) Terminate the host simulator.

[KA_COMMAND_TRACE_0](#) 0-argument kernel trace

[KA_COMMAND_TRACE_1](#) 1-argument kernel trace

[KA_COMMAND_TRACE_2](#) 2-argument kernel trace

[KA_COMMAND_PRINT](#) Print an arbitrary string of data.

Definition at line 33 of file [kernelaware.h](#).

18.60 kernelaware.h

```

00001  /*=====
00002
00003  _____
00004  |   \   /   |   \   /   |   \   /   |   \   /   |   \   /   |
00005  |  / \ / \  |  / \ / \  |  / \ / \  |  / \ / \  |  / \ / \  |
00006  | /   \   \ | /   \   \ | /   \   \ | /   \   \ | /   \   \ |
00007  |_____|   |_____|   |_____|   |_____|   |_____|   |_____|
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00021  #ifndef __KERNEL_AWARE_H__
00022  #define __KERNEL_AWARE_H__
00023
00024  #include "kerneltypes.h"
00025  #include "mark3cfg.h"
00026
00027  #if KERNEL_AWARE_SIMULATION
00028  //-----
00033  typedef enum
00034  {
00035      KA_COMMAND_IDLE = 0,
00036      KA_COMMAND_PROFILE_INIT,
00037      KA_COMMAND_PROFILE_START,
00038      KA_COMMAND_PROFILE_STOP,
00039      KA_COMMAND_PROFILE_REPORT,
00040      KA_COMMAND_EXIT_SIMULATOR,
00041      KA_COMMAND_TRACE_0,
00042      KA_COMMAND_TRACE_1,
00043      KA_COMMAND_TRACE_2,
00044      KA_COMMAND_PRINT
00045  } KernelAwareCommand_t;
00046
00047  //-----
00065  class KernelAware
00066  {
00067  public:
00068      //-----
00079      static void ProfileInit( const char *szStr_ );
00080
00081      //-----
00089      static void ProfileStart( void );
00090
00091      //-----
00098      static void ProfileStop( void );
00099
00100      //-----
00108      static void ProfileReport( void );
00109
00110      //-----
00118      static void ExitSimulator( void );
00119
00120      //-----
00128      static void Print( const char *szStr_ );
00129
00130      //-----
00140      static void Trace( uint16_t ul6File_,
00141                        uint16_t ul6Line_);
00142
00143      //-----
00154      static void Trace( uint16_t ul6File_,
00155                        uint16_t ul6Line_,
00156                        uint16_t ul6Arg1_);
00157
00158      //-----
00170      static void Trace( uint16_t ul6File_,
00171                        uint16_t ul6Line_,
00172                        uint16_t ul6Arg1_,
00173                        uint16_t ul6Arg2_);
00174
00175      //-----
00185      static bool IsSimulatorAware(void);
00186
00187  private:
00188
00189      //-----
00202      static void Trace_i( uint16_t ul6File_,
00203                          uint16_t ul6Line_,
00204                          uint16_t ul6Arg1_,
00205                          uint16_t ul6Arg2_,
00206                          KernelAwareCommand_t eCmd_);
00207  };

```



```

00033
00034 //-----
00035 #define KERNEL_TRACE( x ) \
00036 { \
00037     EMIT_DBG_STRING( x ); \
00038     uint16_t aul6Msg__[4]; \
00039     aul6Msg__[0] = 0xACDC; \
00040     aul6Msg__[1] = DBG_FILE; \
00041     aul6Msg__[2] = __LINE__; \
00042     aul6Msg__[3] = TraceBuffer::Increment(); \
00043     TraceBuffer::Write(aul6Msg__, 4); \
00044 };
00045
00046 //-----
00047 #define KERNEL_TRACE_1( x, arg1 ) \
00048 { \
00049     EMIT_DBG_STRING( x ); \
00050     uint16_t aul6Msg__[5]; \
00051     aul6Msg__[0] = 0xACDC; \
00052     aul6Msg__[1] = DBG_FILE; \
00053     aul6Msg__[2] = __LINE__; \
00054     aul6Msg__[3] = TraceBuffer::Increment(); \
00055     aul6Msg__[4] = arg1; \
00056     TraceBuffer::Write(aul6Msg__, 5); \
00057 };
00058
00059 //-----
00060 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00061 { \
00062     EMIT_DBG_STRING( x ); \
00063     uint16_t aul6Msg__[6]; \
00064     aul6Msg__[0] = 0xACDC; \
00065     aul6Msg__[1] = DBG_FILE; \
00066     aul6Msg__[2] = __LINE__; \
00067     aul6Msg__[3] = TraceBuffer::Increment(); \
00068     aul6Msg__[4] = arg1; \
00069     aul6Msg__[5] = arg2; \
00070     TraceBuffer::Write(aul6Msg__, 6); \
00071 };
00072
00073 //-----
00074 #define KERNEL_ASSERT( x ) \
00075 { \
00076     if( ( x ) == false ) \
00077     { \
00078         EMIT_DBG_STRING( "ASSERT FAILED" ); \
00079         uint16_t aul6Msg__[4]; \
00080         aul6Msg__[0] = 0xACDC; \
00081         aul6Msg__[1] = DBG_FILE; \
00082         aul6Msg__[2] = __LINE__; \
00083         aul6Msg__[3] = TraceBuffer::Increment(); \
00084         TraceBuffer::Write(aul6Msg__, 4); \
00085         Kernel::Panic(PANIC_ASSERT_FAILED); \
00086     } \
00087 };
00088 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_LOGGING)
00089
00090 //-----
00091 #define KERNEL_TRACE( x ) \
00092 { \
00093     EMIT_DBG_STRING( x ); \
00094     KernelAware::Trace( DBG_FILE, __LINE__ ); \
00095 };
00096
00097 //-----
00098 #define KERNEL_TRACE_1( x, arg1 ) \
00099 { \
00100     EMIT_DBG_STRING( x ); \
00101     KernelAware::Trace( DBG_FILE, __LINE__, arg1 ); \
00102 };
00103
00104 //-----
00105 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00106 { \
00107     EMIT_DBG_STRING( x ); \
00108     KernelAware::Trace( DBG_FILE, __LINE__, arg1, arg2 ); \
00109 };
00110
00111 //-----
00112 #define KERNEL_ASSERT( x ) \
00113 { \
00114     if( ( x ) == false ) \
00115     { \
00116         EMIT_DBG_STRING( "ASSERT FAILED" ); \
00117         KernelAware::Trace( DBG_FILE, __LINE__ ); \
00118         Kernel::Panic( PANIC_ASSERT_FAILED ); \
00119     } \

```

```

00120 }
00121
00122 #else
00123 //-----
00124 // Note -- when kernel-debugging is disabled, we still have to define the
00125 // macros to ensure that the expressions compile (albeit, by elimination
00126 // during pre-processing).
00127 //-----
00128 #define KERNEL_TRACE( x )
00129 //-----
00130 #define KERNEL_TRACE_1( x, arg1 )
00131 //-----
00132 #define KERNEL_TRACE_2( x, arg1, arg2 )
00133 //-----
00134 #define KERNEL_ASSERT( x )
00135
00136 #endif // KERNEL_USE_DEBUG
00137
00138
00139 //-----
00140 #if (KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_USER_LOGGING)
00141
00142 //-----
00143 #define USER_TRACE( x ) \
00144 { \
00145     EMIT_DBG_STRING( x ); \
00146     uint16_t aul6Msg__[4]; \
00147     aul6Msg__[0] = 0xACDC; \
00148     aul6Msg__[1] = DBG_FILE; \
00149     aul6Msg__[2] = __LINE__; \
00150     aul6Msg__[3] = TraceBuffer::Increment(); \
00151     TraceBuffer::Write(aul6Msg__, 4); \
00152 };
00153
00154 //-----
00155 #define USER_TRACE_1( x, arg1 ) \
00156 { \
00157     EMIT_DBG_STRING( x ); \
00158     uint16_t aul6Msg__[5]; \
00159     aul6Msg__[0] = 0xACDC; \
00160     aul6Msg__[1] = DBG_FILE; \
00161     aul6Msg__[2] = __LINE__; \
00162     aul6Msg__[3] = TraceBuffer::Increment(); \
00163     aul6Msg__[4] = arg1; \
00164     TraceBuffer::Write(aul6Msg__, 5); \
00165 };
00166
00167 //-----
00168 #define USER_TRACE_2( x, arg1, arg2 ) \
00169 { \
00170     EMIT_DBG_STRING( x ); \
00171     uint16_t aul6Msg__[6]; \
00172     aul6Msg__[0] = 0xACDC; \
00173     aul6Msg__[1] = DBG_FILE; \
00174     aul6Msg__[2] = __LINE__; \
00175     aul6Msg__[3] = TraceBuffer::Increment(); \
00176     aul6Msg__[4] = arg1; \
00177     aul6Msg__[5] = arg2; \
00178     TraceBuffer::Write(aul6Msg__, 6); \
00179 };
00180
00181 //-----
00182 #define USER_ASSERT( x ) \
00183 { \
00184     if( ( x ) == false ) \
00185     { \
00186         EMIT_DBG_STRING( "ASSERT FAILED" ); \
00187         uint16_t aul6Msg__[4]; \
00188         aul6Msg__[0] = 0xACDC; \
00189         aul6Msg__[1] = DBG_FILE; \
00190         aul6Msg__[2] = __LINE__; \
00191         aul6Msg__[3] = TraceBuffer::Increment(); \
00192         TraceBuffer::Write(aul6Msg__, 4); \
00193         Kernel::Panic(PANIC_ASSERT_FAILED); \
00194     } \
00195 };
00196 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_USER_LOGGING)
00197
00198 //-----
00199 #define USER_TRACE( x ) \
00200 { \
00201     EMIT_DBG_STRING( x ); \
00202     KernelAware::Trace( DBG_FILE, __LINE__ ); \
00203 };
00204
00205 //-----
00206 #define USER_TRACE_1( x, arg1 ) \

```

```

00207 { \
00208     EMIT_DBG_STRING( x ); \
00209     KernelAware::Trace( DBG_FILE, __LINE__, arg1 ); \
00210 }
00211
00212 //-----
00213 #define USER_TRACE_2( x, arg1, arg2 ) \
00214 { \
00215     EMIT_DBG_STRING( x ); \
00216     KernelAware::Trace( DBG_FILE, __LINE__, arg1, arg2 ); \
00217 }
00218
00219 //-----
00220 #define USER_ASSERT( x ) \
00221 { \
00222     if( ( x ) == false ) \
00223     { \
00224         EMIT_DBG_STRING( "ASSERT FAILED" ); \
00225         KernelAware::Trace( DBG_FILE, __LINE__ ); \
00226         Kernel::Panic( PANIC_ASSERT_FAILED ); \
00227     } \
00228 }
00229
00230 #else
00231 //-----
00232 // Note -- when kernel-debugging is disabled, we still have to define the
00233 // macros to ensure that the expressions compile (albeit, by elimination
00234 // during pre-processing).
00235 //-----
00236 #define USER_TRACE( x )
00237 //-----
00238 #define USER_TRACE_1( x, arg1 )
00239 //-----
00240 #define USER_TRACE_2( x, arg1, arg2 )
00241 //-----
00242 #define USER_ASSERT( x )
00243
00244 #endif // KERNEL_USE_DEBUG
00245
00246 #endif

```

18.63 /home/mo/mark3-source/embedded/kernel/public/kerneltypes.h File Reference

Basic data type primitives used throughout the OS.

```
#include <stdint.h>
```

Macros

- `#define K_ADDR uint32_t`
Primitive datatype representing address-size.
- `#define K_WORD uint32_t`
Primitive datatype representing a data word.

Typedefs

- `typedef void(* panic_func_t)(uint16_t u16PanicCode_)`
Function pointer type used to implement kernel-panic handlers.

Enumerations

- `enum EventFlagOperation_t {`
`EVENT_FLAG_ALL, EVENT_FLAG_ANY, EVENT_FLAG_ALL_CLEAR, EVENT_FLAG_ANY_CLEAR,`
`EVENT_FLAG_MODES, EVENT_FLAG_PENDING_UNBLOCK }`
This enumeration describes the different operations supported by the event flag blocking object.

18.63.1 Detailed Description

Basic data type primitives used throughout the OS.

Definition in file [kerneltypes.h](#).

18.63.2 Enumeration Type Documentation

18.63.2.1 enum EventFlagOperation_t

This enumeration describes the different operations supported by the event flag blocking object.

Enumerator

EVENT_FLAG_ALL Block until all bits in the specified bitmask are set.

EVENT_FLAG_ANY Block until any bits in the specified bitmask are set.

EVENT_FLAG_ALL_CLEAR Block until all bits in the specified bitmask are cleared.

EVENT_FLAG_ANY_CLEAR Block until any bits in the specified bitmask are cleared.

EVENT_FLAG_MODES Count of event-flag modes. Not used by user

EVENT_FLAG_PENDING_UNBLOCK Special code. Not used by user

Definition at line 43 of file [kerneltypes.h](#).

18.64 kerneltypes.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00019 #include <stdint.h>
00020
00021 #ifndef __KERNELTYPES_H__
00022 #define __KERNELTYPES_H__
00023
00024 //-----
00025 #if !defined(K_ADDR)
00026     #define K_ADDR        uint32_t
00027 #endif
00028 #if !defined(K_WORD)
00029     #define K_WORD        uint32_t
00030 #endif
00031
00032 //-----
00036 typedef void (*panic_func_t)( uint16_t u16PanicCode_ );
00037
00038 //-----
00043 typedef enum
00044 {
00045     EVENT_FLAG_ALL,
00046     EVENT_FLAG_ANY,
00047     EVENT_FLAG_ALL_CLEAR,
00048     EVENT_FLAG_ANY_CLEAR,
00049 //---
00050     EVENT_FLAG_MODES,
00051     EVENT_FLAG_PENDING_UNBLOCK
00052 } EventFlagOperation_t;
00053
00054 #endif

```

18.65 /home/mo/mark3-source/embedded/kernel/public/ksemaphore.h File Reference

[Semaphore](#) Blocking Object class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "threadlist.h"
```

Classes

- class [Semaphore](#)

Binary & Counting semaphores, based on [BlockingObject](#) base class.

18.65.1 Detailed Description

[Semaphore](#) Blocking Object class declarations.

Definition in file [ksemaphore.h](#).

18.66 ksemaphore.h

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00022 #ifndef __KSEMAPHORE_H__
00023 #define __KSEMAPHORE_H__
00024
00025 #include "kerneltypes.h"
00026 #include "mark3cfg.h"
00027
00028 #include "blocking.h"
00029 #include "threadlist.h"
00030
00031 #if KERNEL_USE_SEMAPHORE
00032
00033 //-----
00037 class Semaphore : public BlockingObject
00038 {
00039 public:
00061 void Init(uint16_t u16InitVal_, uint16_t u16MaxVal_);
00062
00077 bool Post();
00078
00086 void Pend();
00087
00099 uint16_t GetCount();
00100
00101 #if KERNEL_USE_TIMEOUTS
00102
00113 bool Pend( uint32_t u32WaitTimeMS_);
00114
00125 void WakeMe(Thread *pClChosenOne_);
00126 #endif
00127
00128 private:
00129
00135 uint8_t WakeNext();
00136
00137 #if KERNEL_USE_TIMEOUTS
00138
```



```

00007      |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00043 #ifndef __LL_H__
00044 #define __LL_H__
00045
00046 #include "kerneltypes.h"
00047
00048 //-----
00049 #ifndef NULL
00050 #define NULL      (0)
00051 #endif
00052
00053 //-----
00059 class LinkList;
00060 class DoubleLinkList;
00061 class CircularLinkList;
00062
00063 //-----
00068 class LinkListNode
00069 {
00070 protected:
00071
00072     LinkListNode *next;
00073     LinkListNode *prev;
00074
00075     LinkListNode() { }
00076
00082     void ClearNode();
00083
00084 public:
00092     LinkListNode *GetNext(void) { return next; }
00093
00101     LinkListNode *GetPrev(void) { return prev; }
00102
00103     friend class LinkList;
00104     friend class DoubleLinkList;
00105     friend class CircularLinkList;
00106     friend class ThreadList;
00107 };
00108
00109 //-----
00113 class LinkList
00114 {
00115 protected:
00116     LinkListNode *m_pstHead;
00117     LinkListNode *m_pstTail;
00118
00119 public:
00125     void Init() { m_pstHead = NULL; m_pstTail = NULL; }
00126
00134     virtual void Add(LinkListNode *node_) = 0;
00135
00143     virtual void Remove(LinkListNode *node_) = 0;
00144
00152     LinkListNode *GetHead() { return m_pstHead; }
00153
00161     LinkListNode *GetTail() { return m_pstTail; }
00162 };
00163
00164 //-----
00168 class DoubleLinkList : public LinkList
00169 {
00170 public:
00176     DoubleLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00177
00185     virtual void Add(LinkListNode *node_);
00186
00194     virtual void Remove(LinkListNode *node_);
00195 };
00196
00197 //-----
00201 class CircularLinkList : public LinkList
00202 {
00203 public:
00204     CircularLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00205
00213     virtual void Add(LinkListNode *node_);
00214
00222     virtual void Remove(LinkListNode *node_);
00223
00230     void PivotForward();
00231

```

```

00238     void PivotBackward();
00239
00249     void InsertNodeBefore(LinkListNode *node_,
        LinkListNode *insert_);
00250 };
00251
00252 #endif

```

18.69 /home/mo/mark3-source/embedded/kernel/public/mailbox.h File Reference

Mailbox + Envelope IPC Mechanism.

```

#include "mark3cfg.h"
#include "kerneltypes.h"
#include "ksemaphore.h"

```

Classes

- class **Mailbox**

The **Mailbox** class implements an IPC mechnism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

18.69.1 Detailed Description

Mailbox + Envelope IPC Mechanism.

Definition in file [mailbox.h](#).

18.70 mailbox.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #ifndef __MAILBOX_H__
00022 #define __MAILBOX_H__
00023
00024 #include "mark3cfg.h"
00025 #include "kerneltypes.h"
00026 #include "ksemaphore.h"
00027
00028 #if KERNEL_USE_MAILBOX
00029
00035 class Mailbox
00036 {
00037 public:
00038
00049     void Init( void *pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_ );
00050
00051 #if KERNEL_USE_AUTO_ALLOC
00052
00065     static Mailbox* Init( uint16_t u16BufferSize_, uint16_t u16ElementSize_ );
00066
00067 #endif
00068
00082     bool Send( void *pvData_ );
00083
00097     bool SendTail( void *pvData_ );
00098

```



```

00099 #if KERNEL_USE_TIMEOUTS
00100
00114     bool Send( void *pvData_, uint32_t u32TimeoutMS_ );
00115
00130     bool SendTail( void *pvData_, uint32_t u32TimeoutMS_ );
00131 #endif
00132
00142     void Receive( void *pvData_ );
00143
00153     void ReceiveTail( void *pvData_ );
00154
00155 #if KERNEL_USE_TIMEOUTS
00156
00168     bool Receive( void *pvData_, uint32_t u32TimeoutMS_ );
00169
00182     bool ReceiveTail( void *pvData_, uint32_t u32TimeoutMS_ );
00183 #endif
00184
00185     uint16_t GetFreeSlots( void )
00186     {
00187         uint16_t rc;
00188         CS_ENTER();
00189         rc = m_ul6Free;
00190         CS_EXIT();
00191         return rc;
00192     }
00193
00194     bool IsFull( void )
00195     {
00196         return (GetFreeSlots() == 0);
00197     }
00198
00199     bool IsEmpty( void )
00200     {
00201         return (GetFreeSlots() == m_ul6Count);
00202     }
00203
00204 private:
00205
00214     void *GetHeadPointer(void)
00215     {
00216         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00217         uAddr += (K_ADDR)(m_ul6ElementSize) * (K_ADDR)(
m_ul6Head);
00218         return (void*)uAddr;
00219     }
00220
00229     void *GetTailPointer(void)
00230     {
00231         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00232         uAddr += (K_ADDR)(m_ul6ElementSize) * (K_ADDR)(
m_ul6Tail);
00233         return (void*)uAddr;
00234     }
00235
00245     void CopyData( const void *src_, const void *dst_, uint16_t len_ )
00246     {
00247         uint8_t *u8Src = (uint8_t*)src_;
00248         uint8_t *u8Dst = (uint8_t*)dst_;
00249         while (len_--)
00250         {
00251             *u8Dst++ = *u8Src++;
00252         }
00253     }
00254
00260     void MoveTailForward(void)
00261     {
00262         m_ul6Tail++;
00263         if (m_ul6Tail == m_ul6Count)
00264         {
00265             m_ul6Tail = 0;
00266         }
00267     }
00268
00274     void MoveHeadForward(void)
00275     {
00276         m_ul6Head++;
00277         if (m_ul6Head == m_ul6Count)
00278         {
00279             m_ul6Head = 0;
00280         }
00281     }
00282
00288     void MoveTailBackward(void)
00289     {
00290         if (m_ul6Tail == 0)
00291     
```

```

00292         m_ul6Tail = m_ul6Count;
00293     }
00294     m_ul6Tail--;
00295 }
00296
00302 void MoveHeadBackward(void)
00303 {
00304     if (m_ul6Head == 0)
00305     {
00306         m_ul6Head = m_ul6Count;
00307     }
00308     m_ul6Head--;
00309 }
00310
00311 #if KERNEL_USE_TIMEOUTS
00312
00322     bool Send_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ );
00323 #else
00324
00333     bool Send_i( const void *pvData_, bool bTail_ );
00334 #endif
00335
00336 #if KERNEL_USE_TIMEOUTS
00337
00347     bool Receive_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ );
00348 #else
00349
00357     void Receive_i( const void *pvData_, bool bTail_ );
00358 #endif
00359
00360     uint16_t m_ul6Head;
00361     uint16_t m_ul6Tail;
00362
00363     uint16_t m_ul6Count;
00364     volatile uint16_t m_ul6Free;
00365
00366     uint16_t m_ul6ElementSize;
00367     const void *m_pvBuffer;
00368
00369     Semaphore m_clRecvSem;
00370
00371 #if KERNEL_USE_TIMEOUTS
00372     Semaphore m_clSendSem;
00373 #endif
00374
00375 };
00376
00377 #endif
00378
00379 #endif
00380

```

18.71 /home/mo/mark3-source/embedded/kernel/public/manual.h File Reference

/brief Ascii-format documentation, used by doxygen to create various printable and viewable forms.

18.71.1 Detailed Description

/brief Ascii-format documentation, used by doxygen to create various printable and viewable forms.

Definition in file [manual.h](#).

18.72 manual.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.

```

```

00012 See license.txt for more information
00013 =====*/
03763
03775

```

18.73 /home/mo/mark3-source/embedded/kernel/public/mark3.h File Reference

Single include file given to users of the Mark3 [Kernel](#) API.

```

#include "mark3cfg.h"
#include "kerneltypes.h"
#include "threadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "kernelprofile.h"
#include "kernel.h"
#include "thread.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "mutex.h"
#include "eventflag.h"
#include "message.h"
#include "notify.h"
#include "mailbox.h"
#include "atomic.h"
#include "driver.h"
#include "kernelaware.h"
#include "profile.h"
#include "autoalloc.h"

```

18.73.1 Detailed Description

Single include file given to users of the Mark3 [Kernel](#) API.

Definition in file [mark3.h](#).

18.74 mark3.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00021 #ifndef __MARK3_H__
00022 #define __MARK3_H__
00023
00024 #include "mark3cfg.h"
00025 #include "kerneltypes.h"
00026
00027 #include "threadport.h"
00028 #include "kernelswi.h"
00029 #include "kerneltimer.h"
00030 #include "kernelprofile.h"
00031
00032 #include "kernel.h"
00033 #include "thread.h"

```

```

00034 #include "timerlist.h"
00035
00036 #include "ksemaphore.h"
00037 #include "mutex.h"
00038 #include "eventflag.h"
00039 #include "message.h"
00040 #include "notify.h"
00041 #include "mailbox.h"
00042
00043 #include "atomic.h"
00044 #include "driver.h"
00045
00046 #include "kernelaware.h"
00047
00048 #include "profile.h"
00049 #include "autoalloc.h"
00050
00051 #endif

```

18.75 /home/mo/mark3-source/embedded/kernel/public/mark3cfg.h File Reference

Mark3 [Kernel](#) Configuration.

Macros

- **#define [KERNEL_USE_TIMERS](#) (1)**
The following options is related to all kernel time-tracking.
- **#define [KERNEL_TIMERS_TICKLESS](#) (1)**
If you've opted to use the kernel timers module, you have an option as to which timer implementation to use: Tick-based or Tick-less.
- **#define [KERNEL_USE_TIMEOUTS](#) (1)**
By default, if you opt to enable kernel timers, you also get timeout- enabled versions of the blocking object APIs along with it.
- **#define [KERNEL_USE_QUANTUM](#) (1)**
Do you want to enable time quanta? This is useful when you want to have tasks in the same priority group share time in a controlled way.
- **#define [THREAD_QUANTUM_DEFAULT](#) (4)**
This value defines the default thread quantum when [KERNEL_USE_QUANTUM](#) is enabled.
- **#define [KERNEL_USE_NOTIFY](#) (1)**
This is a simple blocking object, where a thread (or threads) are guaranteed to block until an asynchronous event signals the object.
- **#define [KERNEL_USE_SEMAPHORE](#) (1)**
Do you want the ability to use counting/binary semaphores for thread synchronization? Enabling this features provides fully-blocking semaphores and enables all API functions declared in [semaphore.h](#).
- **#define [KERNEL_USE_MUTEX](#) (1)**
Do you want the ability to use mutual exclusion semaphores (mutex) for resource/block protection? Enabling this feature provides mutexes, with priority inheritance, as declared in [mutex.h](#).
- **#define [KERNEL_USE_EVENTFLAG](#) (1)**
Provides additional event-flag based blocking.
- **#define [KERNEL_USE_MESSAGE](#) (1)**
Enable inter-thread messaging using message queues.
- **#define [GLOBAL_MESSAGE_POOL_SIZE](#) (8)**
If Messages are enabled, define the size of the default kernel message pool.
- **#define [KERNEL_USE_MAILBOX](#) (1)**
Enable inter-thread messaging using mailboxes.
- **#define [KERNEL_USE_SLEEP](#) (1)**
Do you want to be able to set threads to sleep for a specified time? This enables the [Thread::Sleep\(\)](#) API.
- **#define [KERNEL_USE_DRIVER](#) (1)**

Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.

- `#define KERNEL_USE_THREADNAME (0)`
Provide [Thread](#) method to allow the user to set a name for each thread in the system.
- `#define KERNEL_USE_DYNAMIC_THREADS (1)`
Provide extra [Thread](#) methods to allow the application to create (and more importantly destroy) threads at runtime.
- `#define KERNEL_USE_PROFILER (1)`
Provides extra classes for profiling the performance of code.
- `#define KERNEL_USE_DEBUG (1)`
Provides extra logic for kernel debugging, and instruments the kernel with extra asserts, and kernel trace functionality.
- `#define KERNEL_ENABLE_LOGGING (0)`
Set this to 1 to enable very chatty kernel logging.
- `#define KERNEL_ENABLE_USER_LOGGING (1)`
This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently.
- `#define KERNEL_USE_ATOMIC (0)`
Provides support for atomic operations, including addition, subtraction, set, and test-and-set.
- `#define SAFE_UNLINK (0)`
"Safe unlinking" performs extra checks on data to make sure that there are no consistencies when performing operations on linked lists.
- `#define KERNEL_AWARE_SIMULATION (1)`
Include support for kernel-aware simulation.
- `#define KERNEL_USE_IDLE_FUNC (1)`
Enabling this feature removes the necessity for the user to dedicate a complete thread for idle functionality.
- `#define KERNEL_USE_AUTO_ALLOC (0)`
This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory).

18.75.1 Detailed Description

Mark3 [Kernel](#) Configuration.

This file is used to configure the kernel for your specific application in order to provide the optimal set of features for a given use case.

Since you only pay the price (code space/RAM) for the features you use, you can usually find a sweet spot between features and resource usage by picking and choosing features a-la-carte. This config file is written in an "interactive" way, in order to minimize confusion about what each option provides, and to make dependencies obvious.

Definition in file [mark3cfg.h](#).

18.75.2 Macro Definition Documentation

18.75.2.1 `#define GLOBAL_MESSAGE_POOL_SIZE (8)`

If Messages are enabled, define the size of the default kernel message pool.

Messages can be manually added to the message pool, but this mechanism is more convenient and automatic. All message queues share their message objects from this global pool to maximize efficiency and simplify data management.

Definition at line 150 of file [mark3cfg.h](#).

18.75.2.2 `#define KERNEL_AWARE_SIMULATION (1)`

Include support for kernel-aware simulation.

Enabling this feature adds advanced profiling, trace, and environment-aware debugging and diagnostic functionality when Mark3-based applications are run on the flavr AVR simulator.

Definition at line 254 of file [mark3cfg.h](#).

18.75.2.3 `#define KERNEL_ENABLE_LOGGING (0)`

Set this to 1 to enable very chatty kernel logging.

Since most important things in the kernel emit logs, a large log-buffer and fast output are required in order to keep up. This is a pretty advanced power-user type feature, so it's disabled by default.

Definition at line 218 of file [mark3cfg.h](#).

18.75.2.4 `#define KERNEL_ENABLE_USER_LOGGING (1)`

This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently.

This allows for user-code to benefit from the built-in kernel logging macros without having to account for the super-high-volume of logs generated by kernel code. 1 to enable logging outside of kernel code

Definition at line 227 of file [mark3cfg.h](#).

18.75.2.5 `#define KERNEL_TIMERS_TICKLESS (1)`

If you've opted to use the kernel timers module, you have an option as to which timer implementation to use: Tick-based or Tick-less.

Tick-based timers provide a "traditional" RTOS timer implementation based on a fixed-frequency timer interrupt. While this provides very accurate, reliable timing, it also means that the CPU is being interrupted far more often than may be necessary (as not all timer ticks result in "real work" being done).

Tick-less timers still rely on a hardware timer interrupt, but uses a dynamic expiry interval to ensure that the interrupt is only called when the next timer expires. This increases the complexity of the timer interrupt handler, but reduces the number and frequency.

Note that the CPU port ([kerneltimer.cpp](#)) must be implemented for the particular timer variant desired.

Definition at line 62 of file [mark3cfg.h](#).

18.75.2.6 `#define KERNEL_USE_ATOMIC (0)`

Provides support for atomic operations, including addition, subtraction, set, and test-and-set.

Add/Sub/Set contain 8, 16, and 32-bit variants.

Definition at line 238 of file [mark3cfg.h](#).

18.75.2.7 `#define KERNEL_USE_AUTO_ALLOC (0)`

This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory).

Note that auto-alloc memory cannot be reclaimed.

Definition at line 271 of file [mark3cfg.h](#).

18.75.2.8 #define KERNEL_USE_DYNAMIC_THREADS (1)

Provide extra [Thread](#) methods to allow the application to create (and more importantly destroy) threads at runtime. useful for designs implementing worker threads, or threads that can be restarted after encountering error conditions. Definition at line [197](#) of file [mark3cfg.h](#).

18.75.2.9 #define KERNEL_USE_EVENTFLAG (1)

Provides additional event-flag based blocking.

This relies on an additional per-thread flag-mask to be allocated, which adds 2 bytes to the size of each thread object.

Definition at line [129](#) of file [mark3cfg.h](#).

18.75.2.10 #define KERNEL_USE_IDLE_FUNC (1)

Enabling this feature removes the necessity for the user to dedicate a complete thread for idle functionality.

This saves a full thread stack, but also requires a bit extra static data. This also adds a slight overhead to the context switch and scheduler, as a special case has to be taken into account.

Definition at line [263](#) of file [mark3cfg.h](#).

18.75.2.11 #define KERNEL_USE_MAILBOX (1)

Enable inter-thread messaging using mailboxes.

A mailbox manages a blob of data provided by the user, that is partitioned into fixed-size blocks called envelopes. The size of an envelope is set by the user when the mailbox is initialized. Any number of threads can read-from and write-to the mailbox. Envelopes can be sent-to or received-from the mailbox at the head or tail. In this way, mailboxes essentially act as a circular buffer that can be used as a blocking FIFO or LIFO queue.

Definition at line [163](#) of file [mark3cfg.h](#).

18.75.2.12 #define KERNEL_USE_MESSAGE (1)

Enable inter-thread messaging using message queues.

This is the preferred mechanism for IPC for serious multi-threaded communications; generally anywhere a semaphore or event-flag is insufficient.

Definition at line [137](#) of file [mark3cfg.h](#).

18.75.2.13 #define KERNEL_USE_PROFILER (1)

Provides extra classes for profiling the performance of code.

useful for debugging and development, but uses an additional hardware timer.

Definition at line [203](#) of file [mark3cfg.h](#).

18.75.2.14 #define KERNEL_USE_QUANTUM (1)

Do you want to enable time quanta? This is useful when you want to have tasks in the same priority group share time in a controlled way.

This allows equal tasks to use unequal amounts of the CPU, which is a great way to set up CPU budgets per thread in a round-robin scheduling system. If enabled, you can specify a number of ticks that serves as the default time period (quantum). Unless otherwise specified, every thread in a priority will get the default quantum.

Definition at line 92 of file [mark3cfg.h](#).

18.75.2.15 `#define KERNEL_USE_SEMAPHORE (1)`

Do you want the ability to use counting/binary semaphores for thread synchronization? Enabling this features provides fully-blocking semaphores and enables all API functions declared in semaphore.h.

If you have to pick one blocking mechanism, this is the one to choose.

Definition at line 115 of file [mark3cfg.h](#).

18.75.2.16 `#define KERNEL_USE_THREADNAME (0)`

Provide [Thread](#) method to allow the user to set a name for each thread in the system.

Adds a const char* pointer to the size of the thread object.

Definition at line 189 of file [mark3cfg.h](#).

18.75.2.17 `#define KERNEL_USE_TIMEOUTS (1)`

By default, if you opt to enable kernel timers, you also get timeout- enabled versions of the blocking object APIs along with it.

This support comes at a small cost to code size, but a slightly larger cost to realtime performance - as checking for the use of timers in the underlying internal code costs some cycles.

As a result, the option is given to the user here to manually disable these timeout-based APIs if desired by the user for performance and code-size reasons.

Definition at line 77 of file [mark3cfg.h](#).

18.75.2.18 `#define KERNEL_USE_TIMERS (1)`

The following options is related to all kernel time-tracking.

-timers provide a way for events to be periodically triggered in a lightweight manner. These can be periodic, or one-shot.

-Thread [Quantum](#) (usedd for round-robin scheduling) is dependent on this module, as is [Thread](#) Sleep functionality.

Definition at line 41 of file [mark3cfg.h](#).

18.75.2.19 `#define SAFE_UNLINK (0)`

"Safe unlinking" performs extra checks on data to make sure that there are no consistencies when performing operations on linked lists.

This goes beyond pointer checks, adding a layer of structural and metadata validation to help detect system corruption early.

Definition at line 246 of file [mark3cfg.h](#).

18.75.2.20 `#define THREAD_QUANTUM_DEFAULT (4)`

This value defines the default thread quantum when `KERNEL_USE_QUANTUM` is enabled.


```

00212
00218     #define KERNEL_ENABLE_LOGGING          (0)
00219
00227     #define KERNEL_ENABLE_USER_LOGGING      (1)
00228 #else
00229     #define KERNEL_ENABLE_LOGGING          (0)
00230     #define KERNEL_ENABLE_USER_LOGGING      (0)
00231 #endif
00232
00233
00238 #define KERNEL_USE_ATOMIC                    (0)
00239
00246 #define SAFE_UNLINK                          (0)
00247
00254 #define KERNEL_AWARE_SIMULATION              (1)
00255
00263 #define KERNEL_USE_IDLE_FUNC                 (1)
00264
00271 #define KERNEL_USE_AUTO_ALLOC                (0)
00272
00273 #if KERNEL_USE_AUTO_ALLOC
00274     #define AUTO_ALLOC_SIZE                  (512)
00275 #endif
00276
00277 #endif

```

18.77 /home/mo/mark3-source/embedded/kernel/public/message.h File Reference

Inter-thread communication via message-passing.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "ksemaphore.h"
#include "timerlist.h"

```

Classes

- class [Message](#)
Class to provide message-based IPC services in the kernel.
- class [GlobalMessagePool](#)
Implements a list of message objects shared between all threads.
- class [MessageQueue](#)
List of messages, used as the channel for sending and receiving messages between threads.

18.77.1 Detailed Description

Inter-thread communication via message-passing.

Embedded systems guru Jack Ganssle once said that without a robust form of interprocess communications (IPC), an RTOS is just a toy. Mark3 implements a form of IPC to provide safe and flexible messaging between threads.

using kernel-managed IPC offers significant benefits over other forms of data sharing (i.e. Global variables) in that it avoids synchronization issues and race conditions common to the practice. using IPC also enforces a more disciplined coding style that keeps threads decoupled from one another and minimizes global data, preventing careless and hard-to-debug errors.

18.77.2 using Messages, Queues, and the Global Message Pool

```

// Declare a message queue shared between two threads
MessageQueue my_queue;

int main()

```

```
{
    ...
    // Initialize the message queue
    my_queue.init();
    ...
}

void Thread1()
{
    // Example TX thread - sends a message every 10ms
    while(1)
    {
        // Grab a message from the global message pool
        Message *tx_message = GlobalMessagePool::Pop();

        // Set the message data/parameters
        tx_message->SetCode( 1234 );
        tx_message->SetData( NULL );

        // Send the message on the queue.
        my_queue.Send( tx_message );
        Thread::Sleep(10);
    }
}

void Thread2()
{
    while()
    {
        // Blocking receive - wait until we have messages to process
        Message *rx_message = my_queue.Recv();

        // Do something with the message data...

        // Return back into the pool when done
        GlobalMessagePool::Push(rx_message);
    }
}
```

Definition in file [message.h](#).

18.78 message.h

```

00001  /*
00002
00003
00004
00005
00006
00007
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  ===== */
00080  #ifndef __MESSAGE_H__
00081  #define __MESSAGE_H__
00082
00083  #include "kerneltypes.h"
00084  #include "mark3cfg.h"
00085
00086  #include "ll.h"
00087  #include "ksemaphore.h"
00088
00089  #if KERNEL_USE_MESSAGE
00090
00091  #if KERNEL_USE_TIMEOUTS
00092      #include "timerlist.h"
00093  #endif
00094
00095  //-----
00099  class Message : public LinkListNode
00100  {
00101  public:
00107      void Init() { ClearNode(); m_pvData = NULL; m_ul6Code = 0; }
00108
00116      void SetData( void *pvData_ ) { m_pvData = pvData_; }
00117
00125      void *GetData() { return m_pvData; }
00126
00134      void SetCode( uint16_t ul6Code_ ) { m_ul6Code = ul6Code_; }

```

```

00135
00143     uint16_t GetCode() { return m_ul6Code; }
00144 private:
00145
00147     void *m_pvData;
00148
00150     uint16_t m_ul6Code;
00151 };
00152
00153 //-----
00157 class GlobalMessagePool
00158 {
00159 public:
00165     static void Init();
00166
00176     static void Push( Message *pclMessage_ );
00177
00186     static Message *Pop();
00187
00188 private:
00190     static Message m_aclMessagePool[
        GLOBAL_MESSAGE_POOL_SIZE];
00191
00193     static DoubleLinkedList m_clList;
00194 };
00195
00196 //-----
00201 class MessageQueue
00202 {
00203 public:
00209     void Init();
00210
00219     Message *Receive();
00220
00221 #if KERNEL_USE_TIMEOUTS
00222
00236     Message *Receive( uint32_t u32TimeWaitMS_ );
00237 #endif
00238
00247     void Send( Message *pclSrc_ );
00248
00256     uint16_t GetCount();
00257 private:
00258
00259 #if KERNEL_USE_TIMEOUTS
00260
00269     Message *Receive_i( uint32_t u32TimeWaitMS_ );
00270 #else
00271
00278     Message *Receive_i( void );
00279 #endif
00280
00282     Semaphore m_clSemaphore;
00283
00285     DoubleLinkedList m_clLinkList;
00286 };
00287
00288 #endif //KERNEL_USE_MESSAGE
00289
00290 #endif

```

18.79 /home/mo/mark3-source/embedded/kernel/public/mutex.h File Reference

Mutual exclusion class declaration.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "timerlist.h"

```

Classes

- class [Mutex](#)

Mutual-exclusion locks, based on [BlockingObject](#).

18.79.1 Detailed Description

Mutual exclusion class declaration.

Resource locks are implemented using mutual exclusion semaphores (Mutex_t). Protected blocks can be placed around any resource that may only be accessed by one thread at a time. If additional threads attempt to access the protected resource, they will be placed in a wait queue until the resource becomes available. When the resource becomes available, the thread with the highest original priority claims the resource and is activated. Priority inheritance is included in the implementation to prevent priority inversion. Always ensure that you claim and release your mutex objects consistently, otherwise you may end up with a deadlock scenario that's hard to debug.

18.79.2 Initializing

Initializing a mutex object by calling:

```
clMutex.Init();
```

18.79.3 Resource protection example

```
clMutex.Claim();
...
<resource protected block>
...
clMutex.Release();
```

Definition in file [mutex.h](#).

18.80 mutex.h

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00050 #ifndef __MUTEX_H_
00051 #define __MUTEX_H_
00052
00053 #include "kerneltypes.h"
00054 #include "mark3cfg.h"
00055
00056 #include "blocking.h"
00057
00058 #if KERNEL_USE_MUTEX
00059
00060 #if KERNEL_USE_TIMEOUTS
00061 #include "timerlist.h"
00062 #endif
00063
00064 //-----
00068 class Mutex : public BlockingObject
00069 {
00070 public:
00077     void Init();
00078
00096     void Claim();
00097
00098 #if KERNEL_USE_TIMEOUTS
00099
00110     bool Claim(uint32_t u32WaitTimeMS_);
00111
00124     void WakeMe( Thread *pclOwner_ );
00125
00126 #endif
```



```
00027 #if KERNEL_USE_NOTIFY
00028
00033 class Notify : public BlockingObject
00034 {
00035 public:
00041     void Init(void);
00042
00042     void Signal(void);
00053
00063     void Wait( bool *pbFlag_ );
00064
00065 #if KERNEL_USE_TIMEOUTS
00066
00078     bool Wait( uint32_t u32WaitTimeMS_, bool *pbFlag_ );
00079 #endif
00090     void WakeMe(Thread *pclChosenOne_);
00091 };
00092
00093 #endif
00094
00095 #endif
```

18.83 /home/mo/mark3-source/embedded/kernel/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

18.83.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file [paniccodes.h](#).

18.84 paniccodes.h

```

00001 /*-----
00002
00003 |-----|-----|-----|-----|-----|-----|-----|
00004 | \ / | | | \ / | | | \ / | | | \ / | | | \ / | | | \ / |
00005 | / \ | | | / \ | | | / \ | | | / \ | | | / \ | | | / \ |
00006 |-----|-----|-----|-----|-----|-----|-----|
00007 |-----|-----|-----|-----|-----|-----|-----|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00020 #ifndef __PANIC_CODES_H
00021 #define __PANIC_CODES_H
00022
00023 #define PANIC_ASSERT_FAILED (1)
00024 #define PANIC_LIST_UNLINK_FAILED (2)
00025 #define PANIC_STACK_SLACK_VIOLATED (3)
00026 #define PANIC_AUTO_HEAP_EXHUSTED (4)
00027
00028 #endif // __PANIC_CODES_H
00029

```

18.85 /home/mo/mark3-source/embedded/kernel/public/profile.h File Reference

High-precision profiling timers.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

Classes

- class [ProfileTimer](#)

Profiling timer.

18.85.1 Detailed Description

High-precision profiling timers.

Enables the profiling and instrumentation of performance-critical code. Multiple timers can be used simultaneously to enable system-wide performance metrics to be computed in a lightweight manner.

Usage:

```
ProfileTimer clMyTimer;
int i;

clMyTimer.Init();

// Profile the same block of code ten times
for (i = 0; i < 10; i++)
{
    clMyTimer.Start();
    ...
    //Block of code to profile
    ...
    clMyTimer.Stop();
}

// Get the average execution time of all iterations
u32AverageTimer = clMyTimer.GetAverage();

// Get the execution time from the last iteration
u32LastTimer = clMyTimer.GetCurrent();
```

Definition in file [profile.h](#).

18.86 profile.h

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00053 #ifndef __PROFILE_H__
00054 #define __PROFILE_H__
00055
00056 #include "kerneltypes.h"
00057 #include "mark3cfg.h"
00058 #include "ll.h"
00059
00060 #if KERNEL_USE_PROFILER
00061
00070 class ProfileTimer
00071 {
00072
00073 public:
00080     void Init();
00081
00088     void Start();
00089
00096     void Stop();
00097
00105     uint32_t GetAverage();
00106
00115     uint32_t GetCurrent();
00116
```



```
00117 private:
00118
00129     uint32_t ComputeCurrentTicks(uint16_t u16Count_, uint32_t u32Epoch);
00130
00131     uint32_t m_u32Cumulative;
00132     uint32_t m_u32CurrentIteration;
00133     uint16_t m_u16Initial;
00134     uint32_t m_u32InitialEpoch;
00135     uint16_t m_u16Iterations;
00136     bool m_bActive;
00137 };
00138
00139 #endif // KERNEL_USE_PROFILE
00140
00141 #endif
```

18.87 /home/mo/mark3-source/embedded/kernel/public/quantum.h File Reference

Thread Quantum declarations for Round-Robin Scheduling.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "timer.h"
#include "timerlist.h"
#include "timerscheduler.h"
```

Classes

- class Quantum

Static-class used to implement [Thread](#) quantum functionality, which is a key part of round-robin scheduling.

18.87.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file [quantum.h](#).

18.88 quantum.h

```

00001 /*-----
00002
00003 |-----|-----|-----|-----|-----|
00004 | | | | | | | | | | | | | | | | | | | | | |
00005 | | | | | | | | | | | | | | | | | | | | | |
00006 | | | | | | | | | | | | | | | | | | | | | |
00007 |-----|-----|-----|-----|-----|
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00022 #ifndef __KQUANTUM_H__
00023 #define __KQUANTUM_H__
00024
00025 #include "kerneltypes.h"
00026 #include "mark3cfg.h"
00027
00028 #include "thread.h"
00029 #include "timer.h"
00030 #include "timerlist.h"
00031 #include "timerscheduler.h"
00032
00033 #if KERNEL_USE_QUANTUM
00034 class Timer;
00035

```

```

00041 class Quantum
00042 {
00043 public:
00052     static void UpdateTimer();
00053
00060     static void AddThread( Thread *pclThread_ );
00061
00067     static void RemoveThread();
00068
00077     static void SetInTimer(void) { m_bInTimer = true; }
00078
00084     static void ClearInTimer(void) { m_bInTimer = false; }
00085
00086 private:
00098     static void SetTimer( Thread *pclThread_ );
00099
00100     static Timer m_clQuantumTimer;
00101     static bool m_bActive;
00102     static bool m_bInTimer;
00103 };
00104
00105 #endif //KERNEL_USE_QUANTUM
00106
00107 #endif

```

18.89 /home/mo/mark3-source/embedded/kernel/public/scheduler.h File Reference

[Thread](#) scheduler function declarations.

```

#include "kerneltypes.h"
#include "thread.h"
#include "threadport.h"

```

Classes

- class [Scheduler](#)
Priority-based round-robin [Thread](#) scheduling, using ThreadLists for housekeeping.

Macros

- #define [NUM_PRIORITIES](#) (8)
Defines the maximum number of thread priorities supported in the scheduler.

Variables

- volatile [Thread](#) * [g_pclNext](#)
Pointer to the currently-chosen next-running thread.
- [Thread](#) * [g_pclCurrent](#)
Pointer to the currently-running thread.

18.89.1 Detailed Description

[Thread](#) scheduler function declarations.

This scheduler implements a very flexible type of scheduling, which has become the defacto industry standard when it comes to real-time operating systems. This scheduling mechanism is referred to as priority round- robin.

From the name, there are two concepts involved here:

1) Priority scheduling:

Threads are each assigned a priority, and the thread with the highest priority which is ready to run gets to execute.

2) Round-robin scheduling:

Where there are multiple ready threads at the highest-priority level, each thread in that group gets to share time, ensuring that progress is made.

The scheduler uses an array of [ThreadList](#) objects to provide the necessary housekeeping required to keep track of threads at the various priorities. As a result, the scheduler contains one [ThreadList](#) per priority, with an additional list to manage the storage of threads which are in the "stopped" state (either have been stopped, or have not been started yet).

Definition in file [scheduler.h](#).

18.90 scheduler.h

```

00001  /*=====
00002
00003  _____
00004  |   \   /   |   |   |   |   |   |   |   |   |   |
00005  |   \   /   |   |   |   |   |   |   |   |   |   |
00006  |   \   /   |   |   |   |   |   |   |   |   |   |
00007  |   \   /   |   |   |   |   |   |   |   |   |   |
00008  |   \   /   |   |   |   |   |   |   |   |   |   |
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  ===== */
00046  #ifndef __SCHEDULER_H__
00047  #define __SCHEDULER_H__
00048
00049  #include "kerneltypes.h"
00050  #include "thread.h"
00051  #include "threadport.h"
00052
00053  extern volatile Thread *g_pclNext;
00054  extern Thread *g_pclCurrent;
00055
00056  #define NUM_PRIORITIES (8)
00057  //-----
00058
00062  class Scheduler
00063  {
00064  public:
00070      static void Init();
00071
00079      static void Schedule();
00080
00088      static void Add(Thread *pclThread_);
00089
00098      static void Remove(Thread *pclThread_);
00099
00112      static bool SetScheduler(bool bEnable_);
00113
00121      static Thread *GetCurrentThread() { return g_pclCurrent; }
00122
00131      static volatile Thread *GetNextThread() { return g_pclNext; }
00132
00143      static ThreadList *GetThreadList(uint8_t u8Priority_) { return &
m_aclPriorities[u8Priority_]; }
00144
00153      static ThreadList *GetStopList() { return &m_clStopList; }
00154
00163      static uint8_t IsEnabled() { return m_bEnabled; }
00164
00171      static void QueueScheduler() { m_bQueuedSchedule = true; }
00172
00173  private:
00175      static bool m_bEnabled;
00176
00178      static bool m_bQueuedSchedule;
00179
00181      static ThreadList m_clStopList;
00182
00184      static ThreadList m_aclPriorities[NUM_PRIORITIES];
00185
00187      static uint8_t m_u8PriFlag;
00188  };
00189  #endif
00190

```

18.91 /home/mo/mark3-source/embedded/kernel/public/thread.h File Reference

Platform independent thread class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "scheduler.h"
#include "threadport.h"
#include "quantum.h"
#include "autoalloc.h"
```

Classes

- class [Thread](#)
Object providing fundamental multitasking support in the kernel.
- struct [FakeThread_t](#)
If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

Typedefs

- typedef void(* [ThreadEntry_t](#))(void *pvArg_)
Function pointer type used for thread entriypoint functions.

Enumerations

- enum [ThreadState_t](#)
Enumeration representing the different states a thread can exist in.

18.91.1 Detailed Description

Platform independent thread class declarations.

Threads are an atomic unit of execution, and each instance of the thread class represents an instance of a program running of the processor. The [Thread](#) is the fundmanetal user-facing object in the kernel - it is what makes multiprocessing possible from application code.

In Mark3, threads each have their own context - consisting of a stack, and all of the registers required to multiplex a processor between multiple threads.

The [Thread](#) class inherits directly from the [LinkListNode](#) class to facilitate efficient thread management using Double, or Double-Circular linked lists.

Definition in file [thread.h](#).

18.92 thread.h

```
00001  /*=====
00002
00003  00004  00005  00006  00007
```



```

00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00035 #ifndef __THREAD_H__
00036 #define __THREAD_H__
00037
00038 #include "kerneltypes.h"
00039 #include "mark3cfg.h"
00040
00041 #include "ll.h"
00042 #include "threadlist.h"
00043 #include "scheduler.h"
00044 #include "threadport.h"
00045 #include "quantum.h"
00046 #include "autoalloc.h"
00047
00048 //-----
00052 typedef void (*ThreadEntry_t)(void *pvArg_);
00053
00054 //-----
00058 typedef enum
00059 {
00060     THREAD_STATE_EXIT = 0,
00061     THREAD_STATE_READY,
00062     THREAD_STATE_BLOCKED,
00063     THREAD_STATE_STOP,
00064 } ThreadState_t;
00065
00066 //-----
00072 class Thread : public LinkListNode
00073 {
00074 public:
00090     void Init(K_WORD *pwStack_,
00091              uint16_t u16StackSize_,
00092              uint8_t u8Priority_,
00093              ThreadEntry_t pfEntryPoint_,
00094              void *pvArg_ );
00095
00096 #if KERNEL_USE_AUTO_ALLOC
00097
00115     static Thread* Init(uint16_t u16StackSize_,
00116                        uint8_t u8Priority_,
00117                        ThreadEntry_t pfEntryPoint_,
00118                        void *pvArg_);
00119 #endif
00120
00128     void Start();
00129
00130
00137     void Stop();
00138
00139 #if KERNEL_USE_THREADNAME
00140
00149     void SetName(const char *szName_) { m_szName = szName_; }
00150
00157     const char* GetName() { return m_szName; }
00158 #endif
00159
00168     ThreadList *GetOwner(void) { return m_pclOwner; }
00169
00177     ThreadList *GetCurrent(void) { return m_pclCurrent; }
00178
00187     uint8_t GetPriority(void) { return m_u8Priority; }
00188
00196     uint8_t GetCurPriority(void) { return m_u8CurPriority; }
00197
00198 #if KERNEL_USE_QUANTUM
00199
00206     void SetQuantum( uint16_t u16Quantum_ ) { m_u16Quantum = u16Quantum_; }
00207
00215     uint16_t GetQuantum(void) { return m_u16Quantum; }
00216 #endif
00217
00225     void SetCurrent( ThreadList *pclNewList_ ) {
00226         m_pclCurrent = pclNewList_; }
00226
00234     void SetOwner( ThreadList *pclNewList_ ) { m_pclOwner = pclNewList_; }
00235
00248     void SetPriority(uint8_t u8Priority_);
00249
00259     void InheritPriority(uint8_t u8Priority_);
00260

```

```

00261 #if KERNEL_USE_DYNAMIC_THREADS
00262
00273     void Exit();
00274 #endif
00275
00276 #if KERNEL_USE_SLEEP
00277
00285     static void Sleep(uint32_t u32TimeMs_);
00286
00295     static void USleep(uint32_t u32TimeUs_);
00296 #endif
00297
00305     static void Yield(void);
00306
00314     void SetID( uint8_t u8ID_ ) { m_u8ThreadID = u8ID_; }
00315
00323     uint8_t GetID() { return m_u8ThreadID; }
00324
00325
00338     uint16_t GetStackSlack();
00339
00340 #if KERNEL_USE_EVENTFLAG
00341
00348     uint16_t GetEventFlagMask() { return m_ul6FlagMask; }
00349
00354     void SetEventFlagMask(uint16_t ul6Mask_) { m_ul6FlagMask = ul6Mask_; }
00355
00361     void SetEventFlagMode(EventFlagOperation_t eMode_ ) {
00362         m_eFlagMode = eMode_; }
00363
00367     EventFlagOperation_t GetEventFlagMode() { return
00368         m_eFlagMode; }
00369 #endif
00370 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00371
00374     Timer *GetTimer();
00375 #endif
00376 #if KERNEL_USE_TIMEOUTS
00377
00385     void SetExpired( bool bExpired_ );
00386
00393     bool GetExpired();
00394 #endif
00395
00396 #if KERNEL_USE_IDLE_FUNC
00397
00402     void InitIdle();
00403 #endif
00404
00411     ThreadState_t GetState() { return
00412         m_eState; }
00413
00420     void SetState( ThreadState_t eState_ ) { m_eState = eState_; }
00421
00422     friend class ThreadPort;
00423
00424 private:
00432     static void ContextSwitchSWI(void);
00433
00439     void SetPriorityBase(uint8_t u8Priority_);
00440
00442     K_WORD *m_pwStackTop;
00443
00445     K_WORD *m_pwStack;
00446
00448     uint8_t m_u8ThreadID;
00449
00451     uint8_t m_u8Priority;
00452
00454     uint8_t m_u8CurPriority;
00455
00457     ThreadState_t m_eState;
00458
00459 #if KERNEL_USE_THREADNAME
00460     const char *m_szName;
00462 #endif
00463
00465     uint16_t m_ul6StackSize;
00466
00468     ThreadList *m_pclCurrent;
00469
00471     ThreadList *m_pclOwner;
00472
00474     ThreadEntry_t m_pfEntryPoint;
00475
00477     void *m_pvArg;

```

```

00478
00479 #if KERNEL_USE_QUANTUM
00480     uint16_t m_u16Quantum;
00482 #endif
00483
00484 #if KERNEL_USE_EVENTFLAG
00485     uint16_t m_u16FlagMask;
00487
00489     EventFlagOperation_t m_eFlagMode;
00490 #endif
00491
00492 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00493     Timer m_clTimer;
00495 #endif
00496 #if KERNEL_USE_TIMEOUTS
00497     bool m_bExpired;
00499 #endif
00500
00501 };
00502
00503 #if KERNEL_USE_IDLE_FUNC
00504 //-----
00516 typedef struct
00517 {
00518     LinkListNode *next;
00519     LinkListNode *prev;
00520
00522     K_WORD *m_pwStackTop;
00523
00525     K_WORD *m_pwStack;
00526
00528     uint8_t m_u8ThreadID;
00529
00531     uint8_t m_u8Priority;
00532
00534     uint8_t m_u8CurPriority;
00535
00537     ThreadState_t m_eState;
00538
00539 #if KERNEL_USE_THREADNAME
00540     const char *m_szName;
00542 #endif
00543
00544 } FakeThread_t;
00545 #endif
00546
00547 #endif

```

18.93 /home/mo/mark3-source/embedded/kernel/public/threadlist.h File Reference

[Thread](#) linked-list declarations.

```

#include "kerneltypes.h"
#include "ll.h"

```

Classes

- class [ThreadList](#)

This class is used for building thread-management facilities, such as schedulers, and blocking objects.

18.93.1 Detailed Description

[Thread](#) linked-list declarations.

Definition in file [threadlist.h](#).

18.94 threadlist.h

```

00001 /*=====

```

```

00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00022 #ifndef __THREADLIST_H__
00023 #define __THREADLIST_H__
00024
00025 #include "kerneltypes.h"
00026 #include "ll.h"
00027
00028 class Thread;
00029
00034 class ThreadList : public CircularLinkedList
00035 {
00036 public:
00042 ThreadList() { m_u8Priority = 0; m_pu8Flag = NULL; }
00043
00051 void SetPriority(uint8_t u8Priority_);
00052
00061 void SetFlagPointer(uint8_t *pu8Flag_);
00062
00070 void Add(LinkListNode *node_);
00071
00083 void Add(LinkListNode *node_, uint8_t *pu8Flag_, uint8_t u8Priority_);
00084
00093 void AddPriority(LinkListNode *node_);
00094
00102 void Remove(LinkListNode *node_);
00103
00111 Thread *HighestWaiter();
00112 private:
00113
00115 uint8_t m_u8Priority;
00116
00118 uint8_t *m_pu8Flag;
00119 };
00120
00121 #endif
00122

```

18.95 /home/mo/mark3-source/embedded/kernel/public/timer.h File Reference

Timer object declarations.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"

```

Classes

- class [Timer](#)
Timer - an event-driven execution context based on a specified time interval.

Macros

- #define [TIMERLIST_FLAG_ONE_SHOT](#) (0x01)
Timer is one-shot.
- #define [TIMERLIST_FLAG_ACTIVE](#) (0x02)
Timer is currently active.
- #define [TIMERLIST_FLAG_CALLBACK](#) (0x04)
Timer is pending a callback.

- #define **TIMERLIST_FLAG_EXPIRED** (0x08)
Timer is actually expired.
- #define **MAX_TIMER_TICKS** (0x7FFFFFFF)
Maximum value to set.
- #define **MIN_TICKS** (3)
The minimum tick value to set.

Typedefs

- `typedef void(* TimerCallback_t)(Thread *pclOwner_, void *pvData_)`
This type defines the callback function type for timer events.

18.95.1 Detailed Description

Timer object declarations.

Definition in file [timer.h](#).

18.95.2 Macro Definition Documentation

18.95.2.1 #define TIMERLIST_FLAG_EXPIRED (0x08)

Timer is actually expired.

Definition at line 36 of file timer.h.

18.95.3 Typedef Documentation

18.95.3.1 `typedef void(* TimerCallback_t)(Thread *pclOwner_, void *pvData_)`

This type defines the callback function type for timer events.

Since these are called from an interrupt context, they do not operate from within a thread or object context directly – as a result, the context must be manually passed into the calls.

pcOwner_ is a pointer to the thread that owns the timer pvData_ is a pointer to some data or object that needs to know about the timer's expiry from within the timer interrupt context.

Definition at line 91 of file timer.h.

18.96 timer.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===== */
00021 #ifndef __TIMER_H__
00022 #define __TIMER_H__
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026

```

```

00027 #include "ll.h"
00028
00029 #if KERNEL_USE_TIMERS
00030 class Thread;
00031
00032 //-----
00033 #define TIMERLIST_FLAG_ONE_SHOT          (0x01)
00034 #define TIMERLIST_FLAG_ACTIVE           (0x02)
00035 #define TIMERLIST_FLAG_CALLBACK         (0x04)
00036 #define TIMERLIST_FLAG_EXPIRED          (0x08)
00037
00038 //-----
00039 #define MAX_TIMER_TICKS                  (0x7FFFFFFF)
00040
00041 //-----
00042 #if KERNEL_TIMERS_TICKLESS
00043
00044 //-----
00045 /*
00046     Ugly macros to support a wide resolution of delays.
00047     Given a 16-bit timer @ 16MHz & 256 cycle prescaler, this gives u16...
00048     Max time, SECONDS_TO_TICKS: 68719s
00049     Max time, MSECONDS_TO_TICKS: 6871.9s
00050     Max time, useCONDS_TO_TICKS: 6.8719s
00051
00052     ...With a 16us tick resolution.
00053
00054     Depending on the system frequency and timer resolution, you may want to
00055     customize these values to suit your system more appropriately.
00056 */
00057 //-----
00058 #define SECONDS_TO_TICKS(x)              (((uint32_t)x) * TIMER_FREQ))
00059 #define MSECONDS_TO_TICKS(x)            (((((uint32_t)x) * (TIMER_FREQ/100)) + 5) / 10))
00060 #define useCONDS_TO_TICKS(x)            (((((uint32_t)x) * TIMER_FREQ) + 50000) / 100000))
00061
00062 //-----
00063 #define MIN_TICKS                        (3)
00064 //-----
00065
00066 #else
00067
00068 //-----
00069 // add time because we don't know how far in an epoch we are when a call is made.
00070 #define SECONDS_TO_TICKS(x)              (((uint32_t)(x) * 1000) + 1)
00071 #define MSECONDS_TO_TICKS(x)            ((uint32_t)(x + 1))
00072 #define useCONDS_TO_TICKS(x)            (((uint32_t)(x + 999)) / 1000)
00073
00074 //-----
00075 #define MIN_TICKS                        (1)
00076 //-----
00077
00078 #endif // KERNEL_TIMERS_TICKLESS
00079
00080 //-----
00091 typedef void (*TimerCallback_t)(Thread *pclOwner_, void *pvData_);
00092
00093 //-----
00094 class TimerList;
00095 class TimerScheduler;
00096 class Quantum;
00102 class Timer : public LinkListNode
00103 {
00104 public:
00110     Timer() { }
00111
00117     void Init() { ClearNode(); m_u32Interval = 0;
m_u32TimerTolerance = 0; m_u32TimeLeft = 0;
m_u8Flags = 0; }
00118
00130     void Start( bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback_t pfCallback_,
void *pvData_ );
00131
00145     void Start( bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
TimerCallback_t pfCallback_, void *pvData_ );
00146
00153     void Stop();
00154
00164     void SetFlags (uint8_t u8Flags_) { m_u8Flags = u8Flags_; }
00165
00173     void SetCallback( TimerCallback_t pfCallback_){
m_pfCallback = pfCallback_; }
00174
00182     void SetData( void *pvData_ ){ m_pvData = pvData_; }
00183
00192     void SetOwner( Thread *pclOwner_){ m_pclOwner = pclOwner_; }
00193
00201     void SetIntervalTicks(uint32_t u32Ticks_);

```

```

00202
00210     void SetIntervalSeconds(uint32_t u32Seconds_);
00211
00217     uint32_t GetInterval()    { return m_u32Interval; }
00218
00226     void SetIntervalMSeconds(uint32_t u32MSeconds_);
00227
00235     void SetIntervalUSeconds(uint32_t u32USeconds_);
00236
00245     void SetTolerance(uint32_t u32Ticks_);
00246
00247 private:
00248
00249     friend class TimerList;
00250
00252     uint8_t m_u8Flags;
00253
00255     TimerCallback_t m_pfCallback;
00256
00258     uint32_t m_u32Interval;
00259
00261     uint32_t m_u32TimeLeft;
00262
00264     uint32_t m_u32TimerTolerance;
00265
00267     Thread *m_pclOwner;
00268
00270     void *m_pvData;
00271 };
00272
00273 #endif // KERNEL_USE_TIMERS
00274
00275 #endif

```

18.97 /home/mo/mark3-source/embedded/kernel/public/timerlist.h File Reference

[Timer](#) list declarations.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timer.h"

```

Classes

- class [TimerList](#)

[TimerList](#) class - a doubly-linked-list of timer objects.

18.97.1 Detailed Description

[Timer](#) list declarations.

These classes implements a linked list of timer objects attached to the global kernel timer scheduler.

Definition in file [timerlist.h](#).

18.98 timerlist.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.

```

```

00012 See license.txt for more information
00013 ===== */
00024 #ifndef __TIMERLIST_H__
00025 #define __TIMERLIST_H__
00026
00027 #include "kerneltypes.h"
00028 #include "mark3cfg.h"
00029
00030 #include "timer.h"
00031 #if KERNEL_USE_TIMERS
00032
00033 //-----
00037 class TimerList : public DoubleLinkedList
00038 {
00039 public:
00046     void Init();
00047
00055     void Add(Timer *pclListNode_);
00056
00064     void Remove(Timer *pclListNode_);
00065
00072     void Process();
00073
00074 private:
00076     uint32_t m_u32NextWakeup;
00077
00079     bool m_bTimerActive;
00080 };
00081
00082 #endif // KERNEL_USE_TIMERS
00083
00084 #endif

```

18.99 /home/mo/mark3-source/embedded/kernel/public/timerscheduler.h File Reference

Timer scheduler declarations.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "timer.h"
#include "timerlist.h"

```

Classes

- class **TimerScheduler**

*"Static" Class used to interface a global **TimerList** with the rest of the kernel.*

18.99.1 Detailed Description

Timer scheduler declarations.

Definition in file [timerscheduler.h](#).

18.100 timerscheduler.h

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.

```

18.101 /home/mo/mark3-source/embedded/kernel/public/tracebuffer.h File Reference

```
#include "kerneltypes.h"
#include "mark3cfg.h"
```

Global kernel trace-buffer. used to instrument the kernel with lightweight encoded print statements. If something goes wrong, the tracebuffer can be examined for debugging purposes. Also, subsets of kernel trace information can be extracted and analyzed to provide information about runtime performance, thread-scheduling, and other nifty things in real-time.

18.102 tracebuffer.h

Generated on Mon Jan 11 2016 21:42:35 for Mark3 Realtime Kernel by Doxygen

```

00025 #define __TRACEBUFFER_H__
00026
00027 #include "kerneltypes.h"
00028 #include "mark3cfg.h"
00029
00030 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00031
00032 #define TRACE_BUFFER_SIZE (160)
00033
00034 typedef void (*TraceBufferCallback_t)(uint16_t *pul6Source_, uint16_t ul6Len_, bool bPingPong_);
00035
00036 class TraceBuffer
00037 {
00038 public:
00039     static void Init();
00040
00041     static uint16_t Increment(void)
00042     { return m_ul6SyncNumber++; }
00043
00044     static void Write( uint16_t *pul6Data_, uint16_t ul6Size_ );
00045
00046     static void SetCallback( TraceBufferCallback_t pfCallback_ )
00047     { m_pfCallback = pfCallback_; }
00048 private:
00049     static TraceBufferCallback_t m_pfCallback;
00050     static uint16_t m_ul6SyncNumber;
00051     static uint16_t m_ul6Index;
00052     static uint16_t m_aul6Buffer[ (TRACE_BUFFER_SIZE / sizeof( uint16_t )) ];
00053 };
00054
00055 #endif //KERNEL_USE_DEBUG
00056 #endif

```

18.103 /home/mo/mark3-source/embedded/kernel/quantum.cpp File Reference

[Thread Quantum](#) Implementation for Round-Robin Scheduling.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "timerlist.h"
#include "quantum.h"
#include "kernelaware.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

Functions

- static void [QuantumCallback](#) ([Thread](#) *pclThread_, void *pvData_)
QuantumCallback.

18.103.1 Detailed Description

[Thread Quantum](#) Implementation for Round-Robin Scheduling.

Definition in file [quantum.cpp](#).

18.103.2 Function Documentation

18.103.2.1 static void [QuantumCallback](#) ([Thread](#) * *pclThread_*, void * *pvData_*) [static]

[QuantumCallback.](#)

This is the timer callback that is invoked whenever a thread has exhausted its current execution quantum and a new thread must be chosen from within the same priority level.

Parameters

<i>pclThread_</i>	Pointer to the thread currently executing
<i>pvData_</i>	Unused in this context.

Definition at line 62 of file [quantum.cpp](#).

18.104 quantum.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "thread.h"
00026 #include "timerlist.h"
00027 #include "quantum.h"
00028 #include "kernelaware.h"
00029
00030 #define _CAN_HAS_DEBUG
00031 //--[Autogenerated - Do Not Modify]-----
00032 #include "dbg_file_list.h"
00033 #include "buffalogger.h"
00034 #if defined(DBG_FILE)
00035 # error "Debug logging file token already defined! Bailing."
00036 #else
00037 # define DBG_FILE _DBG__KERNEL_QUANTUM_CPP
00038 #endif
00039 //--[End Autogenerated content]-----
00040 #include "kerneldebug.h"
00041
00042 #if KERNEL_USE_QUANTUM
00043
00044 //-----
00045 static volatile bool bAddQuantumTimer; // Indicates that a timer add is pending
00046
00047 //-----
00048 Timer Quantum::m_clQuantumTimer; // The global timernodelist_t object
00049 bool Quantum::m_bActive;
00050 bool Quantum::m_bInTimer;
00051 //-----
00062 static void QuantumCallback(Thread *pclThread_, void *pvData_)
00063 {
00064     // Validate thread pointer, check that source/destination match (it's
00065     // in its real priority list). Also check that this thread was part of
00066     // the highest-running priority level.
00067     if (pclThread_->GetPriority() >= Scheduler::GetCurrentThread()->
GetPriority())
00068     {
00069         if (pclThread_->GetCurrent()->GetHead() != pclThread_->
GetCurrent()->GetTail() )
00070         {
00071             bAddQuantumTimer = true;
00072             pclThread_->GetCurrent()->PivotForward();
00073         }
00074     }
00075 }
00076
00077 //-----
00078 void Quantum::SetTimer(Thread *pclThread_)
00079 {
00080     m_clQuantumTimer.SetIntervalMSeconds(pclThread_->
GetQuantum());
00081     m_clQuantumTimer.SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00082     m_clQuantumTimer.SetData(NULL);
00083     m_clQuantumTimer.SetCallback((TimerCallback_t)
QuantumCallback);

```

```

00084     m_clQuantumTimer.SetOwner(pclThread_);
00085 }
00086
00087 //-----
00088 void Quantum::AddThread(Thread *pclThread_)
00089 {
00090     if (m_bActive
00091 #if KERNEL_USE_IDLE_FUNC
00092         || (pclThread_ == Kernel::GetIdleThread())
00093 #endif
00094     )
00095     {
00096         return;
00097     }
00098
00099     // If this is called from the timer callback, queue a timer add...
00100     if (m_bInTimer)
00101     {
00102         bAddQuantumTimer = true;
00103         return;
00104     }
00105
00106     // If this isn't the only thread in the list.
00107     if ( pclThread_>GetCurrent()->GetHead() !=
00108         pclThread_>GetCurrent()->GetTail() )
00109     {
00110         Quantum::SetTimer(pclThread_);
00111         TimerScheduler::Add(&m_clQuantumTimer);
00112         m_bActive = 1;
00113     }
00114 }
00115
00116 //-----
00117 void Quantum::RemoveThread(void)
00118 {
00119     if (!m_bActive)
00120     {
00121         return;
00122     }
00123
00124     // Cancel the current timer
00125     TimerScheduler::Remove(&m_clQuantumTimer);
00126     m_bActive = 0;
00127 }
00128
00129 //-----
00130 void Quantum::UpdateTimer(void)
00131 {
00132     // If we have to re-add the quantum timer (more than 2 threads at the
00133     // high-priority level...)
00134     if (bAddQuantumTimer)
00135     {
00136         // Trigger a thread yield - this will also re-schedule the
00137         // thread *and* reset the round-robin scheduler.
00138         Thread::Yield();
00139         bAddQuantumTimer = false;
00140     }
00141 }
00142
00143 #endif //KERNEL_USE_QUANTUM

```

18.105 /home/mo/mark3-source/embedded/kernel/scheduler.cpp File Reference

Strict-Priority + Round-Robin thread scheduler implementation.

```

#include "kerneltypes.h"
#include "ll.h"
#include "scheduler.h"
#include "thread.h"
#include "threadport.h"
#include "kernel.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```


Variables

- volatile `Thread * g_pclNext`
Pointer to the currently-chosen next-running thread.
- `Thread * g_pclCurrent`
Pointer to the currently-running thread.
- static const uint8_t `aucCLZ [16] = {255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3}`
This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

18.105.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file [scheduler.cpp](#).

18.105.2 Variable Documentation

18.105.2.1 `const uint8_t aucCLZ[16] = {255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3}` [static]

This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

It is used to efficiently perform a CLZ operation under the assumption that a native CLZ instruction is unavailable. This table is further optimized to provide a 0xFF result in the event that the index value is itself zero, allowing u16 to quickly identify whether or not subsequent 4-bit LUT operations are required to complete the scheduling process.

Definition at line 61 of file [scheduler.cpp](#).

18.106 scheduler.cpp

```

00001  /*=====
00002
00003  | | | | | | | | | | | | | | | | | |
00004  | | \ / | | | | \ / | | | | \ / | | | | | |
00005  | | \ / | | | | \ / | | | | \ / | | | | | |
00006  | | \ / | | | | \ / | | | | \ / | | | | | |
00007  | | | | | | | | | | | | | | | | | |
00008
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00022  #include "kerneltypes.h"
00023  #include "ll.h"
00024  #include "scheduler.h"
00025  #include "thread.h"
00026  #include "threadport.h"
00027  #include "kernel.h"
00028
00029  #define _CAN_HAS_DEBUG
00030  //--[Autogenerated - Do Not Modify]-----
00031  #include "dbg_file_list.h"
00032  #include "buffallogger.h"
00033  #if defined(DBG_FILE)
00034  # error "Debug logging file token already defined! Bailing."
00035  #else
00036  # define DBG_FILE _DBG__KERNEL_SCHEDULER_CPP
00037  #endif
00038  //--[End Autogenerated content]-----
00039
00040  #include "kerneldebug.h"
00041  volatile Thread *g_pclNext;
00042  Thread *g_pclCurrent;
00043
00044  //-----
00045  bool Scheduler::m_bEnabled;
00046  bool Scheduler::m_bQueuedSchedule;
00047
00048  ThreadList Scheduler::m_clStopList;

```

```

00049 ThreadList Scheduler::m_aclPriorities[
    NUM_PRIORITIES];
00050 uint8_t Scheduler::m_u8PriFlag;
00051
00052 //-----
00061 static const uint8_t aucCLZ[16] = {255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3};
00062
00063 //-----
00064 void Scheduler::Init()
00065 {
00066     m_u8PriFlag = 0;
00067     for (int i = 0; i < NUM_PRIORITIES; i++)
00068     {
00069         m_aclPriorities[i].SetPriority(i);
00070         m_aclPriorities[i].SetFlagPointer(&
m_u8PriFlag);
00071     }
00072     m_bQueuedSchedule = false;
00073 }
00074
00075 //-----
00076 void Scheduler::Schedule()
00077 {
00078     uint8_t u8Pri = 0;
00079
00080     // Figure out what priority level has ready tasks (8 priorities max)
00081     // To do this, we apply our current active-thread bitmap (m_u8PriFlag)
00082     // and perform a CLZ on the upper four bits. If no tasks are found
00083     // in the higher priority bits, search the lower priority bits. This
00084     // also assumes that we always have the idle thread ready-to-run in
00085     // priority level zero.
00086     u8Pri = aucCLZ[m_u8PriFlag >> 4];
00087     if (u8Pri == 0xFF)
00088     {
00089         u8Pri = aucCLZ[m_u8PriFlag & 0x0F];
00090     }
00091     else
00092     {
00093         u8Pri += 4;
00094     }
00095
00096 #if KERNEL_USE_IDLE_FUNC
00097     if (u8Pri == 0xFF)
00098     {
00099         // There aren't any active threads at all - set g_pclNext to IDLE
00100         g_pclNext = Kernel::GetIdleThread();
00101     }
00102     else
00103 #endif
00104     {
00105         // Get the thread node at this priority.
00106         g_pclNext = (Thread*)( m_aclPriorities[u8Pri].GetHead() );
00107     }
00108     KERNEL_TRACE_1( "Next Thread: %d\n", (uint16_t)((Thread*)g_pclNext)->GetID() );
00109
00110 }
00111
00112 //-----
00113 void Scheduler::Add(Thread *pclThread_)
00114 {
00115     m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00116 }
00117
00118 //-----
00119 void Scheduler::Remove(Thread *pclThread_)
00120 {
00121     m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00122 }
00123
00124 //-----
00125 bool Scheduler::SetScheduler(bool bEnable_)
00126 {
00127     bool bRet ;
00128     CS_ENTER();
00129     bRet = m_bEnabled;
00130     m_bEnabled = bEnable_;
00131     // If there was a queued scheduler event, dequeue and trigger an
00132     // immediate Yield
00133     if (m_bEnabled && m_bQueuedSchedule)
00134     {
00135         m_bQueuedSchedule = false;
00136         Thread::Yield();
00137     }
00138     CS_EXIT();
00139     return bRet;
00140 }

```

18.107 /home/mo/mark3-source/embedded/kernel/thread.cpp File Reference

Platform-Independent thread class Definition.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "scheduler.h"
#include "kernelswi.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "quantum.h"
#include "kernel.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Functions

- static void [ThreadSleepCallback](#) ([Thread](#) *pclOwner_, void *pvData_)
This callback is used to wake up a thread once the interval has expired.

18.107.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file [thread.cpp](#).

18.108 thread.cpp

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "thread.h"
00026 #include "scheduler.h"
00027 #include "kernelswi.h"
00028 #include "timerlist.h"
00029 #include "ksemaphore.h"
00030 #include "quantum.h"
00031 #include "kernel.h"
00032
00033 #define _CAN_HAS_DEBUG
00034 //--[Autogenerated - Do Not Modify]-----
00035 #include "dbg_file_list.h"
00036 #include "buffalogger.h"
00037 #if defined(DBG_FILE)
00038 # error "Debug logging file token already defined! Bailing."
00039 #else
00040 # define DBG_FILE _DBG__KERNEL_THREAD_CPP
00041 #endif
00042 //--[End Autogenerated content]-----
00043
00044 #include "kerneldebug.h"
00045 //-----
```

```

00046 void Thread::Init( K_WORD *pwStack_,
00047                     uint16_t u16StackSize_,
00048                     uint8_t u8Priority_,
00049                     ThreadEntry_t pfEntryPoint_,
00050                     void *pvArg_ )
00051 {
00052     static uint8_t u8ThreadID = 0;
00053
00054     KERNEL_ASSERT( pwStack_ );
00055     KERNEL_ASSERT( pfEntryPoint_ );
00056
00057     ClearNode();
00058
00059     m_u8ThreadID = u8ThreadID++;
00060
00061     KERNEL_TRACE_1( "Stack Size: %d", u16StackSize_ );
00062     KERNEL_TRACE_1( "Thread Pri: %d", (uint8_t)u8Priority_ );
00063     KERNEL_TRACE_1( "Thread Id: %d", (uint16_t)m_u8ThreadID );
00064     KERNEL_TRACE_1( "Entrypoint: %x", (uint16_t)pfEntryPoint_ );
00065
00066     // Initialize the thread parameters to their initial values.
00067     m_pwStack = pwStack_;
00068     m_pwStackTop = TOP_OF_STACK(pwStack_, u16StackSize_);
00069
00070     m_u16StackSize = u16StackSize_;
00071
00072     #if KERNEL_USE_QUANTUM
00073         m_u16Quantum = THREAD_QUANTUM_DEFAULT;
00074     #endif
00075
00076     m_u8Priority = u8Priority_ ;
00077     m_u8CurPriority = m_u8Priority;
00078     m_pfEntryPoint = pfEntryPoint_;
00079     m_pvArg = pvArg_;
00080     m_eState = THREAD_STATE_STOP;
00081
00082     #if KERNEL_USE_THREADNAME
00083         m_szName = NULL;
00084     #endif
00085     #if KERNEL_USE_TIMERS
00086         m_clTimer.Init();
00087     #endif
00088
00089     // Call CPU-specific stack initialization
00090     ThreadPort::InitStack(this);
00091
00092     // Add to the global "stop" list.
00093     CS_ENTER();
00094     m_pclOwner = Scheduler::GetThreadList(
00095         m_u8Priority);
00096     m_pclCurrent = Scheduler::GetStopList();
00097     m_pclCurrent->Add(this);
00098     CS_EXIT();
00099 }
00100 #if KERNEL_USE_AUTO_ALLOC
00101 //-----
00102 Thread* Thread::Init(uint16_t u16StackSize_,
00103                      uint8_t u8Priority_,
00104                      ThreadEntry_t pfEntryPoint_,
00105                      void *pvArg_)
00106 {
00107     Thread *pclNew = (Thread*)AutoAlloc::Allocate(sizeof(Thread));
00108     K_WORD *pwStack = (K_WORD*)AutoAlloc::Allocate(u16StackSize_);
00109     pclNew->Init(pwStack, u16StackSize_, u8Priority_, pfEntryPoint_, pvArg_ );
00110     return pclNew;
00111 }
00112 #endif
00113 //-----
00115 void Thread::Start(void)
00116 {
00117     // Remove the thread from the scheduler's "stopped" list, and add it
00118     // to the scheduler's ready list at the proper priority.
00119     KERNEL_TRACE_1( "Starting Thread %d", (uint16_t)m_u8ThreadID );
00120
00121     CS_ENTER();
00122     Scheduler::GetStopList()->Remove(this);
00123     Scheduler::Add(this);
00124     m_pclOwner = Scheduler::GetThreadList(
00125         m_u8Priority);
00126     m_pclCurrent = m_pclOwner;
00127     m_eState = THREAD_STATE_READY;
00128     #if KERNEL_USE_QUANTUM
00129         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
00130             GetCurPriority())

```

```

00130     {
00131         // Deal with the thread Quantum
00132         Quantum::RemoveThread();
00133         Quantum::AddThread(this);
00134     }
00135 #endif
00136
00137     if (Kernel::IsStarted())
00138     {
00139         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
GetCurPriority())
00140         {
00141             Thread::Yield();
00142         }
00143     }
00144     CS_EXIT();
00145 }
00146
00147 //-----
00148 void Thread::Stop()
00149 {
00150     bool bReschedule = 0;
00151
00152     CS_ENTER();
00153
00154     // If a thread is attempting to stop itself, ensure we call the scheduler
00155     if (this == Scheduler::GetCurrentThread())
00156     {
00157         bReschedule = true;
00158     }
00159
00160     // Add this thread to the stop-list (removing it from active scheduling)
00161     // Remove the thread from scheduling
00162     if (m_eState == THREAD_STATE_READY)
00163     {
00164         Scheduler::Remove(this);
00165     }
00166     else if (m_eState == THREAD_STATE_BLOCKED)
00167     {
00168         m_pclCurrent->Remove(this);
00169     }
00170
00171     m_pclOwner = Scheduler::GetStopList();
00172     m_pclCurrent = m_pclOwner;
00173     m_pclOwner->Add(this);
00174     m_eState = THREAD_STATE_STOP;
00175
00176 #if KERNEL_USE_TIMERS
00177     // Just to be safe - attempt to remove the thread's timer
00178     // from the timer-scheduler (does no harm if it isn't
00179     // in the timer-list)
00180     TimerScheduler::Remove(&m_clTimer);
00181 #endif
00182
00183     CS_EXIT();
00184
00185     if (bReschedule)
00186     {
00187         Thread::Yield();
00188     }
00189 }
00190
00191 #if KERNEL_USE_DYNAMIC_THREADS
00192 //-----
00193 void Thread::Exit()
00194 {
00195     bool bReschedule = 0;
00196
00197     KERNEL_TRACE_1( "Exit Thread %d", m_u8ThreadID );
00198
00199     CS_ENTER();
00200
00201     // If this thread is the actively-running thread, make sure we run the
00202     // scheduler again.
00203     if (this == Scheduler::GetCurrentThread())
00204     {
00205         bReschedule = 1;
00206     }
00207
00208     // Remove the thread from scheduling
00209     if (m_eState == THREAD_STATE_READY)
00210     {
00211         Scheduler::Remove(this);
00212     }
00213     else if (m_eState == THREAD_STATE_BLOCKED)
00214     {
00215         m_pclCurrent->Remove(this);

```

```

00216     }
00217
00218     m_pclCurrent = 0;
00219     m_pclOwner = 0;
00220     m_eState = THREAD_STATE_EXIT;
00221
00222     // We've removed the thread from scheduling, but interrupts might
00223     // trigger checks against this thread's currently priority before
00224     // we get around to scheduling new threads. As a result, set the
00225     // priority to idle to ensure that we always wind up scheduling
00226     // new threads.
00227     m_u8CurPriority = 0;
00228     m_u8Priority = 0;
00229
00230 #if KERNEL_USE_TIMERS
00231     // Just to be safe - attempt to remove the thread's timer
00232     // from the timer-scheduler (does no harm if it isn't
00233     // in the timer-list)
00234     TimerScheduler::Remove(&m_clTimer);
00235 #endif
00236
00237     CS_EXIT();
00238
00239     if (bReschedule)
00240     {
00241         // Choose a new "next" thread if we must
00242         Thread::Yield();
00243     }
00244 }
00245 #endif
00246
00247 #if KERNEL_USE_SLEEP
00248 //-----
00250 static void ThreadSleepCallback( Thread *pclOwner_, void *pvData_ )
00251 {
00252     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00253     // Post the semaphore, which will wake the sleeping thread.
00254     pclSemaphore->Post();
00255 }
00256 //-----
00258 void Thread::Sleep(uint32_t u32TimeMs_)
00259 {
00260     Semaphore clSemaphore;
00261     Timer *pclTimer = g_pclCurrent->GetTimer();
00262
00263     // Create a semaphore that this thread will block on
00264     clSemaphore.Init(0, 1);
00265
00266     // Create a one-shot timer that will call a callback that posts the
00267     // semaphore, waking our thread.
00268     pclTimer->Init();
00269     pclTimer->SetIntervalMSeconds(u32TimeMs_);
00270     pclTimer->SetCallback(ThreadSleepCallback);
00271     pclTimer->SetData((void*)&clSemaphore);
00272     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00273
00274     // Add the new timer to the timer scheduler, and block the thread
00275     TimerScheduler::Add(pclTimer);
00276     clSemaphore.Pend();
00277 }
00278 //-----
00280 void Thread::USleep(uint32_t u32TimeUs_)
00281 {
00282     Semaphore clSemaphore;
00283     Timer *pclTimer = g_pclCurrent->GetTimer();
00284
00285     // Create a semaphore that this thread will block on
00286     clSemaphore.Init(0, 1);
00287
00288     // Create a one-shot timer that will call a callback that posts the
00289     // semaphore, waking our thread.
00290     pclTimer->Init();
00291     pclTimer->SetIntervalUSEconds(u32TimeUs_);
00292     pclTimer->SetCallback(ThreadSleepCallback);
00293     pclTimer->SetData((void*)&clSemaphore);
00294     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00295
00296     // Add the new timer to the timer scheduler, and block the thread
00297     TimerScheduler::Add(pclTimer);
00298     clSemaphore.Pend();
00299 }
00300 #endif // KERNEL_USE_SLEEP
00301
00302 //-----
00303 uint16_t Thread::GetStackSlack()

```

```

00304 {
00305     uint16_t u16Count = 0;
00306
00307     CS_ENTER();
00308
00310     for (u16Count = 0; u16Count < m_u16StackSize; u16Count++)
00311     {
00312         if (m_pwStack[u16Count] != 0xFF)
00313         {
00314             break;
00315         }
00316     }
00317
00318     CS_EXIT();
00319
00320     return u16Count;
00321 }
00322
00323 //-----
00324 void Thread::Yield()
00325 {
00326     CS_ENTER();
00327     // Run the scheduler
00328     if (Scheduler::IsEnabled())
00329     {
00330         Scheduler::Schedule();
00331
00332         // Only switch contexts if the new task is different than the old task
00333         if (Scheduler::GetCurrentThread() !=
00334             Scheduler::GetNextThread())
00335         {
00336             #if KERNEL_USE_QUANTUM
00337                 // new thread scheduled. Stop current quantum timer (if it exists),
00338                 // and restart it for the new thread (if required).
00339                 Quantum::RemoveThread();
00340                 Quantum::AddThread((Thread*)g_pclNext);
00341             #endif
00342             Thread::ContextSwitchSWI();
00343         }
00344     }
00345     else
00346     {
00347         Scheduler::QueueScheduler();
00348     }
00349     CS_EXIT();
00350 }
00351
00352 //-----
00353 void Thread::SetPriorityBase(uint8_t u8Priority_)
00354 {
00355     GetCurrent()->Remove(this);
00356
00357     SetCurrent(Scheduler::GetThreadList(
00358         m_u8Priority_));
00359     GetCurrent()->Add(this);
00360 }
00361
00362 //-----
00363 void Thread::SetPriority(uint8_t u8Priority_)
00364 {
00365     bool bSchedule = 0;
00366
00367     CS_ENTER();
00368     // If this is the currently running thread, it's a good idea to reschedule
00369     // Or, if the new priority is a higher priority than the current thread's.
00370     if ((g_pclCurrent == this) || (u8Priority_ > g_pclCurrent->
00371         GetPriority()))
00372     {
00373         bSchedule = 1;
00374     }
00375     Scheduler::Remove(this);
00376     CS_EXIT();
00377
00378     m_u8CurPriority = u8Priority_;
00379     m_u8Priority = u8Priority_;
00380
00381     CS_ENTER();
00382     Scheduler::Add(this);
00383     CS_EXIT();
00384
00385     if (bSchedule)
00386     {
00387         if (Scheduler::IsEnabled())
00388         {
00389             CS_ENTER();

```

```

00389         Scheduler::Schedule();
00390     #if KERNEL_USE_QUANTUM
00391         // new thread scheduled. Stop current quantum timer (if it exists),
00392         // and restart it for the new thread (if required).
00393         Quantum::RemoveThread();
00394         Quantum::AddThread((Thread*)g_pclNext);
00395     #endif
00396     CS_EXIT();
00397     Thread::ContextSwitchSWI();
00398     }
00399     else
00400     {
00401         Scheduler::QueueScheduler();
00402     }
00403 }
00404 }
00405
00406 //-----
00407 void Thread::InheritPriority(uint8_t u8Priority_)
00408 {
00409     SetOwner(Scheduler::GetThreadList(u8Priority_));
00410     m_u8CurPriority = u8Priority_;
00411 }
00412
00413 //-----
00414 void Thread::ContextSwitchSWI()
00415 {
00416     // Call the context switch interrupt if the scheduler is enabled.
00417     if (Scheduler::IsEnabled() == 1)
00418     {
00419         KERNEL_TRACE_1( "Context switch to Thread %d", (uint16_t)((
00420             Thread*)g_pclNext)->GetID() );
00421         KernelSWI::Trigger();
00422     }
00423 }
00424
00425 #if KERNEL_USE_TIMEOUTS
00426 //-----
00427 Timer *Thread::GetTimer() { return &
00428     m_clTimer; }
00429
00430 //-----
00431 void Thread::SetExpired( bool bExpired_ ) { m_bExpired = bExpired_; }
00432
00433 //-----
00434 bool Thread::GetExpired() { return
00435     m_bExpired; }
00436 #endif
00437
00438 #if KERNEL_USE_IDLE_FUNC
00439 //-----
00440 void Thread::InitIdle( void )
00441 {
00442     ClearNode();
00443     m_u8Priority = 0;
00444     m_u8CurPriority = 0;
00445     m_pfEntryPoint = 0;
00446     m_pvArg = 0;
00447     m_u8ThreadID = 255;
00448     m_eState = THREAD_STATE_READY;
00449     #if KERNEL_USE_THREADNAME
00450     m_szName = "IDLE";
00451     #endif
00452 }
00453 #endif

```

18.109 /home/mo/mark3-source/embedded/kernel/threadlist.cpp File Reference

Thread linked-list definitions.

```

#include "kerneltypes.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```


18.109.1 Detailed Description

[Thread](#) linked-list definitions.

Definition in file [threadlist.cpp](#).

18.110 threadlist.cpp

```

00001  /*=====
00002
00003  _____
00004  |   \   |   \   |   \   |   \   |   \   |   \   |
00005  |   \   |   \   |   \   |   \   |   \   |   \   |
00006  |   \   |   \   |   \   |   \   |   \   |   \   |
00007  |   \   |   \   |   \   |   \   |   \   |   \   |
00008  |   \   |   \   |   \   |   \   |   \   |   \   |
00009  --[Mark3 Realtime Platform]-----
00010
00011  Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  =====*/
00022  #include "kerneltypes.h"
00023  #include "ll.h"
00024  #include "threadlist.h"
00025  #include "thread.h"
00026
00027  #define _CAN_HAS_DEBUG
00028  //--[Autogenerated - Do Not Modify]-----
00029  #include "dbg_file_list.h"
00030  #include "buffalogger.h"
00031  #if defined(DBG_FILE)
00032  # error "Debug logging file token already defined!  Bailing."
00033  #else
00034  # define DBG_FILE _DBG___KERNEL_THREADLIST_CPP
00035  #endif
00036  //--[End Autogenerated content]-----
00037  #include "kerneldebug.h"
00038
00039  //-----
00040  void ThreadList::SetPriority(uint8_t u8Priority_)
00041  {
00042      m_u8Priority = u8Priority_;
00043  }
00044
00045  //-----
00046  void ThreadList::SetFlagPointer( uint8_t *pu8Flag_)
00047  {
00048      m_pu8Flag = pu8Flag_;
00049  }
00050
00051  //-----
00052  void ThreadList::Add(LinkListNode *node_) {
00053      CircularLinkList::Add(node_);
00054      CircularLinkList::PivotForward();
00055
00056      // We've specified a bitmap for this threadlist
00057      if (m_pu8Flag)
00058      {
00059          // Set the flag for this priority level
00060          *m_pu8Flag |= (1 << m_u8Priority);
00061      }
00062  }
00063
00064  //-----
00065  void ThreadList::AddPriority(LinkListNode *node_) {
00066      Thread *pclCurr = static_cast<Thread*>(GetHead());
00067      if (!pclCurr) {
00068          Add(node_);
00069          return;
00070      }
00071      uint8_t u8HeadPri = pclCurr->GetCurPriority();
00072
00073      Thread *pclTail = static_cast<Thread*>(GetTail());
00074      Thread *pclNode = static_cast<Thread*>(node_);
00075
00076      // Set the threadlist's priority level, flag pointer, and then add the
00077      // thread to the threadlist
00078      uint8_t u8Priority = pclNode->GetCurPriority();
00079      do
00080      {
00081          if (u8Priority > pclCurr->GetCurPriority())
00082          {

```

```

00083         break;
00084     }
00085     pclCurr = static_cast<Thread*>(pclCurr->GetNext());
00086 } while (pclCurr != pclTail);
00087
00088 // Insert pclNode before pclCurr in the linked list.
00089 InsertNodeBefore(pclNode, pclCurr);
00090
00091 // If the priority is greater than current head, reset
00092 // the head pointer.
00093 if (u8Priority > u8HeadPri) {
00094     m_pstHead = pclNode;
00095     m_pstTail = m_pstHead->prev;
00096 }
00097 else if (pclNode->GetNext() == m_pstHead)
00098 {
00099     m_pstTail = pclNode;
00100 }
00101 }
00102
00103 //-----
00104 void ThreadList::Add(LinkListNode *node_, uint8_t *pu8Flag_, uint8_t u8Priority_
) {
00105     // Set the threadlist's priority level, flag pointer, and then add the
00106     // thread to the threadlist
00107     SetPriority(u8Priority_);
00108     SetFlagPointer(pu8Flag_);
00109     Add(node_);
00110 }
00111
00112 //-----
00113 void ThreadList::Remove(LinkListNode *node_) {
00114     // Remove the thread from the list
00115     CircularLinkList::Remove(node_);
00116
00117     // If the list is empty...
00118     if (!m_pstHead)
00119     {
00120         // Clear the bit in the bitmap at this priority level
00121         if (m_pu8Flag)
00122         {
00123             *m_pu8Flag &= ~(1 << m_u8Priority);
00124         }
00125     }
00126 }
00127
00128 //-----
00129 Thread *ThreadList::HighestWaiter()
00130 {
00131     return static_cast<Thread*>(GetHead());
00132 }

```

18.111 /home/mo/mark3-source/embedded/kernel/timer.cpp File Reference

Timer implementations.

```

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timer.h"
#include "timerlist.h"
#include "timerscheduler.h"
#include "kerneltimer.h"
#include "threadport.h"
#include "quantum.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"

```

18.111.1 Detailed Description

Timer implementations.

Definition in file [timer.cpp](#).

18.112 timer.cpp

```

00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "timer.h"
00026 #include "timerlist.h"
00027 #include "timerscheduler.h"
00028 #include "kerneltimer.h"
00029 #include "threadport.h"
00030 #include "quantum.h"
00031
00032 #define _CAN_HAS_DEBUG
00033 //--[Autogenerated - Do Not Modify]-----
00034 #include "dbg_file_list.h"
00035 #include "buffalogger.h"
00036 #if defined(DBG_FILE)
00037 # error "Debug logging file token already defined! Bailing."
00038 #else
00039 # define DBG_FILE _DBG__KERNEL_TIMER_CPP
00040 #endif
00041 //--[End Autogenerated content]-----
00042
00043 #include "kerneldebug.h"
00044
00045 #if KERNEL_USE_TIMERS
00046
00047 //-----
00048 void Timer::Start( bool bRepeat_, uint32_t u32IntervalMs_,
00049 TimerCallback_t pfCallback_, void *pvData_ )
00049 {
00050     SetIntervalMSeconds(u32IntervalMs_);
00051     m_u32TimerTolerance = 0;
00052     m_pfCallback = pfCallback_;
00053     m_pvData = pvData_;
00054     if (!bRepeat_)
00055     {
00056         m_u8Flags = TIMERLIST_FLAG_ONE_SHOT;
00057     }
00058     else
00059     {
00060         m_u8Flags = 0;
00061     }
00062     m_pclOwner = Scheduler::GetCurrentThread();
00063     TimerScheduler::Add(this);
00064 }
00065
00066 //-----
00067 void Timer::Start( bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
00068 TimerCallback_t pfCallback_, void *pvData_ )
00068 {
00069     m_u32TimerTolerance = MSECONDS_TO_TICKS(u32ToleranceMs_);
00070     Start(bRepeat_, u32IntervalMs_, pfCallback_, pvData_);
00071 }
00072
00073 //-----
00074 void Timer::Stop()
00074 {
00075     TimerScheduler::Remove(this);
00076 }
00077
00078 //-----
00079
00080 void Timer::SetIntervalTicks( uint32_t u32Ticks_ )
00080 {
00081     m_u32Interval = u32Ticks_;
00082 }
00083
00084
00085 //-----

```



```

00031 #define _CAN_HAS_DEBUG
00032 //--[Autogenerated - Do Not Modify]-----
00033 #include "dbg_file_list.h"
00034 #include "buffalogger.h"
00035 #if defined(DBG_FILE)
00036 # error "Debug logging file token already defined!  Bailing."
00037 #else
00038 # define DBG_FILE _DBG__KERNEL_TIMERLIST_CPP
00039 #endif
00040 //--[End Autogenerated content]-----
00041
00042 #include "kerneldebug.h"
00043
00044 #if KERNEL_USE_TIMERS
00045 //-----
00046 TimerList TimerScheduler::m_clTimerList;
00047
00048 //-----
00049 void TimerList::Init(void)
00050 {
00051     m_bTimerActive = 0;
00052     m_u32NextWakeup = 0;
00053 }
00054
00055 //-----
00056 void TimerList::Add(Timer *pclListNode_)
00057 {
00058     #if KERNEL_TIMERS_TICKLESS
00059         bool bStart = 0;
00060         int32_t lDelta;
00061     #endif
00062
00063     CS_ENTER();
00064
00065     #if KERNEL_TIMERS_TICKLESS
00066         if (GetHead() == NULL)
00067         {
00068             bStart = 1;
00069         }
00070     #endif
00071
00072     pclListNode_>ClearNode();
00073     DoubleLinkedList::Add(pclListNode_);
00074
00075     // Set the initial timer value
00076     pclListNode_>m_u32TimeLeft = pclListNode_>m_u32Interval;
00077
00078     #if KERNEL_TIMERS_TICKLESS
00079         if (!bStart)
00080         {
00081             // If the new interval is less than the amount of time remaining...
00082             lDelta = KernelTimer::TimeToExpiry() - pclListNode_>
m_u32Interval;
00083
00084             if (lDelta > 0)
00085             {
00086                 // Set the new expiry time on the timer.
00087                 m_u32NextWakeup = KernelTimer::SubtractExpiry((
uint32_t)lDelta);
00088             }
00089         }
00090         else
00091         {
00092             m_u32NextWakeup = pclListNode_>m_u32Interval;
00093             KernelTimer::SetExpiry(m_u32NextWakeup);
00094             KernelTimer::Start();
00095         }
00096     #endif
00097
00098     // Set the timer as active.
00099     pclListNode_>m_u8Flags |= TIMERLIST_FLAG_ACTIVE;
00100     CS_EXIT();
00101 }
00102
00103 //-----
00104 void TimerList::Remove(Timer *pclLinkListNode_)
00105 {
00106     CS_ENTER();
00107
00108     DoubleLinkedList::Remove(pclLinkListNode_);
00109
00110     #if KERNEL_TIMERS_TICKLESS
00111         if (this->GetHead() == NULL)
00112         {
00113             KernelTimer::Stop();
00114         }
00115     #endif

```

```

00116
00117     CS_EXIT();
00118 }
00119
00120 //-----
00121 void TimerList::Process(void)
00122 {
00123     #if KERNEL_TIMERS_TICKLESS
00124         uint32_t u32NewExpiry;
00125         uint32_t u32Overtime;
00126         bool bContinue;
00127     #endif
00128
00129     Timer *pclNode;
00130     Timer *pclPrev;
00131
00132     #if KERNEL_USE_QUANTUM
00133         Quantum::SetInTimer();
00134     #endif
00135     #if KERNEL_TIMERS_TICKLESS
00136         // Clear the timer and its expiry time - keep it running though
00137         KernelTimer::ClearExpiry();
00138         do
00139         {
00140         #endif
00141             pclNode = static_cast<Timer*>(GetHead());
00142             pclPrev = NULL;
00143
00144             #if KERNEL_TIMERS_TICKLESS
00145                 bContinue = 0;
00146                 u32NewExpiry = MAX_TIMER_TICKS;
00147             #endif
00148
00149             // Subtract the elapsed time interval from each active timer.
00150             while (pclNode)
00151             {
00152                 // Active timers only...
00153                 if (pclNode->m_u8Flags & TIMERLIST_FLAG_ACTIVE)
00154                 {
00155                     // Did the timer expire?
00156                     #if KERNEL_TIMERS_TICKLESS
00157                         if (pclNode->m_u32TimeLeft <= m_u32NextWakeup)
00158                         #else
00159                         if (pclNode->m_u32TimeLeft-- < 0)
00160                         {
00161                             // Yes - set the "callback" flag - we'll execute the callbacks later
00162                             pclNode->m_u8Flags |= TIMERLIST_FLAG_CALLBACK;
00163
00164                             if (pclNode->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT)
00165                             {
00166                                 // If this was a one-shot timer, deactivate the timer.
00167                                 pclNode->m_u8Flags |= TIMERLIST_FLAG_EXPIRED;
00168                                 pclNode->m_u8Flags &= ~TIMERLIST_FLAG_ACTIVE;
00169                             }
00170                         }
00171                     #else
00172                         // Reset the interval timer.
00173                         // I think we're good though...
00174                         pclNode->m_u32TimeLeft = pclNode->
00175                             m_u32Interval;
00176                     #endif
00177
00178                     #if KERNEL_TIMERS_TICKLESS
00179                         // If the time remaining (plus the length of the tolerance interval)
00180                         // is less than the next expiry interval, set the next expiry interval.
00181                         uint32_t u32Tmp = pclNode->m_u32TimeLeft + pclNode->
00182                             m_u32TimerTolerance;
00183
00184                         if (u32Tmp < u32NewExpiry)
00185                         {
00186                             u32NewExpiry = u32Tmp;
00187                         }
00188                     #endif
00189                 }
00190             }
00191             #if KERNEL_TIMERS_TICKLESS
00192             else
00193             {
00194                 // Not expiring, but determine how int32_t to run the next timer interval for.
00195                 pclNode->m_u32TimeLeft -= m_u32NextWakeup;
00196                 if (pclNode->m_u32TimeLeft < u32NewExpiry)
00197                 {
00198                     u32NewExpiry = pclNode->m_u32TimeLeft;
00199                 }
00200             }

```

```

00201 #endif
00202     }
00203     pclNode = static_cast<Timer*>(pclNode->GetNext());
00204 }
00205
00206 // Process the expired timers callbacks.
00207 pclNode = static_cast<Timer*>(GetHead());
00208 while (pclNode)
00209 {
00210     pclPrev = NULL;
00211
00212     // If the timer expired, run the callbacks now.
00213     if (pclNode->m_u8Flags & TIMERLIST_FLAG_CALLBACK)
00214     {
00215         // Run the callback. these callbacks must be very fast...
00216         pclNode->m_pfCallback( pclNode->m_pclOwner, pclNode->
m_pvData );
00217         pclNode->m_u8Flags &= ~TIMERLIST_FLAG_CALLBACK;
00218
00219         // If this was a one-shot timer, let's remove it.
00220         if (pclNode->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT)
00221         {
00222             pclPrev = pclNode;
00223         }
00224     }
00225     pclNode = static_cast<Timer*>(pclNode->GetNext());
00226
00227     // Remove one-shot-timers
00228     if (pclPrev)
00229     {
00230         Remove(pclPrev);
00231     }
00232 }
00233
00234 #if KERNEL_TIMERS_TICKLESS
00235 // Check to see how much time has elapsed since the time we
00236 // acknowledged the interrupt...
00237 u32Overtime = KernelTimer::GetOvertime();
00238
00239 if( u32Overtime >= u32NewExpiry ) {
00240     m_u32NextWakeup = u32Overtime;
00241     bContinue = 1;
00242 }
00243
00244 // If it's taken longer to go through this loop than would take u16 to
00245 // the next expiry, re-run the timing loop
00246 } while (bContinue);
00247
00248 // This timer elapsed, but there's nothing more to do...
00249 // Turn the timer off.
00250 if (u32NewExpiry >= MAX_TIMER_TICKS)
00251 {
00252     KernelTimer::Stop();
00253 }
00254 else
00255 {
00256     // Update the timer with the new "Next Wakeup" value, plus whatever
00257     // overtime has accumulated since the last time we called this handler
00258     m_u32NextWakeup = KernelTimer::SetExpiry(u32NewExpiry +
u32Overtime);
00259 }
00260 #endif
00261 #if KERNEL_USE_QUANTUM
00262 Quantum::ClearInTimer();
00263 #endif
00264 #endif
00265 }
00266
00267
00268
00269 #endif //KERNEL_USE_TIMERS

```

18.115 /home/mo/mark3-source/embedded/kernel/tracebuffer.cpp File Reference

[Kernel](#) trace buffer class definition.

```
#include "kerneltypes.h"
#include "tracebuffer.h"
#include "mark3cfg.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

18.115.1 Detailed Description

[Kernel](#) trace buffer class definition.

Definition in file [tracebuffer.cpp](#).

18.116 tracebuffer.cpp

```
00001 /*=====
00002
00003
00004
00005
00006
00007
00008
00009 --[Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====*/
00019 #include "kerneltypes.h"
00020 #include "tracebuffer.h"
00021 #include "mark3cfg.h"
00022
00023 #define _CAN_HAS_DEBUG
00024 //--[Autogenerated - Do Not Modify]-----
00025 #include "dbg_file_list.h"
00026 #include "buffalogger.h"
00027 #if defined(DBG_FILE)
00028 # error "Debug logging file token already defined! Bailing."
00029 #else
00030 # define DBG_FILE _DBG__KERNEL_TRACEBUFFER_CPP
00031 #endif
00032
00033 #include "kerneldebug.h"
00034
00035 //--[End Autogenerated content]-----
00036
00037 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00038 //--
00039 TraceBufferCallback_t TraceBuffer::m_pfCallback;
00040 uint16_t TraceBuffer::m_ul6Index;
00041 uint16_t TraceBuffer::m_ul6SyncNumber;
00042 uint16_t TraceBuffer::m_aul6Buffer[ (TRACE_BUFFER_SIZE/sizeof(uint16_t)) ];
00043
00044 //--
00045 void TraceBuffer::Init()
00046 {
00047     m_ul6Index = 0;
00048     m_ul6SyncNumber = 0;
00049     m_pfCallback = 0;
00050 }
00051
00052 //--
00053 void TraceBuffer::Write( uint16_t *pul6Data_, uint16_t ul6Size_ )
00054 {
00055     // Pipe the data directly to the circular buffer
00056     uint16_t ul6Start;
00057
00058     // Update the circular buffer index in a critical section. The
00059     // rest of the operations can take place in any context.
00060     CS_ENTER();
00061     uint16_t ul6NextIndex;
00062     ul6Start = m_ul6Index;
00063     ul6NextIndex = m_ul6Index + ul6Size_;
00064     if (ul6NextIndex >= (sizeof(m_aul6Buffer) / sizeof(uint16_t)) )
00065     {
```



```
00066         ul6NextIndex -= (sizeof(m_aul6Buffer) / sizeof(uint16_t));
00067     }
00068     m_ul6Index = ul6NextIndex;
00069     CS_EXIT();
00070
00071     // Write the data into the circular buffer.
00072     uint16_t i;
00073     bool bCallback = false;
00074     bool bPingPong = false;
00075     for (i = 0; i < ul6Size_; i++)
00076     {
00077         m_aul6Buffer[ul6Start++] = pul6Data_[i];
00078         if (ul6Start >= (sizeof(m_aul6Buffer) / sizeof(uint16_t)) )
00079         {
00080             ul6Start = 0;
00081             bCallback = true;
00082         }
00083         else if (ul6Start == ((sizeof(m_aul6Buffer) / sizeof(uint16_t)) / 2))
00084         {
00085             bPingPong = true;
00086             bCallback = true;
00087         }
00088     }
00089
00090     // Done writing - see if there's a 50% or rollover callback
00091     if (bCallback && m_pfCallback) {
00092         uint16_t ul6Size = (sizeof(m_aul6Buffer) / sizeof(uint16_t)) / 2;
00093         if (bPingPong) {
00094             m_pfCallback(m_aul6Buffer, ul6Size, bPingPong);
00095         } else {
00096             m_pfCallback(m_aul6Buffer + ul6Size, ul6Size, bPingPong);
00097         }
00098     }
00099 }
00100
00101 #endif
00102
```


Example Documentation

This example demonstrates how low-overhead logging can be implemented using `buffalogger`.

```

/*=====*/
//-----[Mark3 Realtime Platform]-----
Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
=====*/

#include "mark3.h"
#include "kerneldebug.h"
#include "drvUART.h"
#include "tracebuffer.h"
#include "ksemaphore.h"

/*=====*/

Example - Logging data via buffalogger/debug APIs.

=====*/

#define _CAN_HAS_DEBUG
//--[Autogenerated - Do Not Modify]-----
#include "dbg_file_list.h"
#include "buffalogger.h"
#if defined(DBG_FILE)
# error "Debug logging file token already defined! Bailing."
#else
# define DBG_FILE_DBG_EXAMPLES_AVR_BUFFALOGGER_MAIN_CPP
#endif
//--[End Autogenerated content]-----

//-----
// This block declares the thread data for the main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP_STACK_SIZE    (192/sizeof(K_WORD))
static Thread   clAppThread;
static K_WORD   awAppStack[APP_STACK_SIZE];
static void     AppMain(void *unused_);

#define IDLE_STACK_SIZE   (192/sizeof(K_WORD))
static Thread   clIdleThread;
static K_WORD   awIdleStack[APP_STACK_SIZE];
static void     IdleMain(void *unused_);

#define LOGGER_STACK_SIZE (192/sizeof(K_WORD))
static Thread   clLoggerThread;
static K_WORD   awLoggerStack[APP_STACK_SIZE];
static void     LoggerMain(void *unused_);
static volatile bool bPingPong;
static Semaphore clSem;
```

```

//-----
static ATmegaUART clUART;

//-----
#define UART_SIZE_TX      (32)
#define UART_SIZE_RX      (8)

static uint8_t aucTxBuffer[UART_SIZE_TX];
static uint8_t aucRxBuffer[UART_SIZE_RX];

static volatile uint16_t *pul6Log;
static volatile uint16_t ul6LogLen;

extern "C" {
void __cxa_pure_virtual(void) { }
}

void IdleMain(void* unused_)
{
    while(1)
    {

    }
}

//-----
void LoggerCallback(uint16_t *pul6Data_, uint16_t ul6Len_, bool bPingPong_)
{
    CS_ENTER();
    bPingPong = bPingPong_;
    pul6Log = pul6Data_;
    ul6LogLen = ul6Len_;
    CS_EXIT();

    clSem.Post();
}

//-----
void LoggerMain(void* unused_)
{
    while (1)
    {
        uint8_t* src;
        uint16_t len;

        clSem.Pend();

        CS_ENTER();
        src = (uint8_t*)pul6Log;
        len = ul6LogLen * sizeof(uint16_t);
        CS_EXIT();

        uint16_t written = 0;
        while (len != written)
        {
            written += clUART.Write(len - written, src + written);
        }
    }
}

//-----
int main(void)
{
    Kernel::Init();

    // Example assumes use of built-in idle.
    clAppThread.Init( awAppStack, APP_STACK_SIZE, 2, AppMain, 0);
    clAppThread.Start();

    clLoggerThread.Init( awLoggerStack, LOGGER_STACK_SIZE, 1, LoggerMain, 0);
    clLoggerThread.Start();

    clIdleThread.Init( awIdleStack, IDLE_STACK_SIZE, 0, IdleMain, 0);
    clIdleThread.Start();

    clUART.SetName("/dev/tty");
    clUART.Init();
    clUART.Open();

    DriverList::Add( &clUART );

    Kernel::Start();

    return 0;
}

```

```

//
void AppMain(void *unused_)
{
    {
        uint32_t u32Baud = 57600 * 4;
        clUART.Control( CMD_SET_BAUDRATE, &u32Baud, 0, 0, 0);
    }
    clUART.Control( CMD_SET_BUFFERS, (void*)aucRxBuffer, UART_SIZE_RX, (void*)aucTxBuffer, UART_SIZE_TX );

    clSem.Init(0,1);

    TraceBuffer::SetCallback( LoggerCallback );
    volatile uint16_t ul6Iteration = 0;
    while(1)
    {
        Thread::Sleep(100);
        USER_TRACE("Beginning of the main application loop!");

        Thread::Sleep(100);
        USER_TRACE_1(" Iteration: %d", ul6Iteration++);

        Thread::Sleep(100);
        USER_TRACE("End of the main application loop!");
    }
}

```

19.2 lab10_notifications/main.cpp

This examples demonstrates how to use notification objects as a thread synchronization mechanism.

[illegible]


```

//-----
#define APP_STACK_SIZE      (256/sizeof(K_WORD))
static Thread  clApp1Thread;
static K_WORD  awApp1Stack[APP_STACK_SIZE];
static void    App1Main(void *unused_);

//-----
static Thread  clApp2Thread;
static K_WORD  awApp2Stack[APP_STACK_SIZE];
static void    App2Main(void *unused_);

//-----
static Mailbox clMailbox;
static uint8_t au8MBData[100];

typedef struct
{
    uint8_t au8Buffer[10];
} MBType_t;

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    // Initialize the threads used in this example
    clApp1Thread.Init( awApp1Stack, APP_STACK_SIZE, 1, App1Main, 0);
    clApp1Thread.Start();

    clApp2Thread.Init( awApp2Stack, APP_STACK_SIZE, 2, App2Main, 0);
    clApp2Thread.Start();

    // Initialize the mailbox used in this example
    clMailbox.Init(au8MBData, 100, sizeof(MBType_t));

    Kernel::Start();

    return 0;
}

//-----
void App1Main(void *unused_)
{
    while (1)
    {
        MBType_t stMsg;

        // Wait until there is an envelope available in the shared mailbox, and
        // then log a trace message.
        clMailbox.Receive(&stMsg);
        KernelAware::Trace(0, __LINE__, stMsg.au8Buffer[0], stMsg.au8Buffer[9]);
    }
}

//-----
void App2Main(void *unused_)
{
    while (1)
    {
        MBType_t stMsg;

        // Place a bunch of envelopes in the mailbox, and then wait for a
        // while. Note that this thread has a higher priority than the other
        // thread, so it will keep pushing envelopes to the other thread until
        // it gets to the sleep, at which point the other thread will be allowed
        // to execute.

        KernelAware::Print("Messages Begin\n");

        for (uint8_t i = 0; i < 10; i++)
        {
            for (uint8_t j = 0; j < 10; j++)
            {
                stMsg.au8Buffer[j] = (i * 10) + j;
            }
            clMailbox.Send(&stMsg);
        }

        KernelAware::Print("Messages End\n");
        Thread::Sleep(2000);
    }
}

```



```
// is started are referred to as the "static threads" in the system, as they
// are the default working-set of threads that make up the application on
// kernel startup.

// Initialize the application thread to use a specified word-array as its stack.
// The thread will run at priority level "1", and start execution the
// "AppMain" function when it's started.
clAppThread.Init( awAppStack, APP_STACK_SIZE, 1, AppMain, 0);

// Initialize the idle thread to use a specific word-array as its stack.
// The thread will run at priority level "0", which is reserved for the idle
// priority thread. IdleMain will be run when the thread is started.
clIdleThread.Init( awIdleStack, IDLE_STACK_SIZE, 0, IdleMain, 0);

// Once the static threads have been added, the user must then ensure that the
// threads are ready to execute. By default, creating a thread is created
// in a STOPPED state. All threads must manually be started using the
// Start() API before they will be scheduled by the system. Here, we are
// starting the application and idle threads before starting the kernel - and
// that's OK. When the kernel is started, it will choose which thread to run
// first from the pool of ready threads.

clAppThread.Start();
clIdleThread.Start();

// All threads have been initialized and made ready. The kernel will now
// select the first thread to run, enable the hardware required to run the
// kernel (Timers, software interrupts, etc.), and then do whatever is
// necessary to maneuver control of thread execution to the kernel. At this
// point, execution will transition to the highest-priority ready thread.
// This function will not return.

Kernel::Start();

// As Kernel::Start() results in the operating system being executed, control
// will not be relinquished back to main(). The "return 0" is simply to
// avoid warnings.

return 0;
}

//-----
void AppMain(void *unused_)
{
    // This function is run from within the application thread. Here, we
    // simply print a friendly greeting and allow the thread to sleep for a
    // while before repeating the message. Note that while the thread is
    // sleeping, CPU execution will transition to the Idle thread.

    while(1)
    {
        KernelAware::Print("Hello World!\n");
        Thread::Sleep(1000);
    }
}

//-----
void IdleMain(void *unused_)
{
    while(1)
    {
        // Low priority task + power management routines go here.
        // The actions taken in this context must *not* cause the thread
        // to block, as the kernel requires that at least one thread is
        // schedulable at all times when not using an idle thread.

        // Note that if you have no special power-management code or idle
        // tasks, an empty while(1){} loop is sufficient to guarantee that
        // condition.
    }
}
}
```

19.5 lab2_idle_function/main.cpp

This example demonstrates how to use the idle function, instead of an idle thread to manage system inactivity.

[illegible]

```
--[Mark3 Realtime Platform]-----

Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
=====*/
#include "mark3.h"

/*=====

Lab Example 2: Initializing the Mark3 RTOS kernel with one thread.

The following example code presents a working example of how to initialize
the Mark3 RTOS kernel, configured to use an application thread and the special
Kernel-Idle function. This example is functionally identical to lab1, although
it uses less memory as a result of only requiring one thread. This example also
uses the flAVR kernel-aware module to print out messages when run through the
flAVR AVR Simulator.

Lessons covered in this example include:

- usage of the Kernel::SetIdleFunc() API
- Changing an idle thread into an idle function
- You can save a thread and a stack by using an idle function instead of a
  dedicated idle thread.

Takeaway:

The Kernel-Idle context allows you to run the Mark3 RTOS without running
a dedicated idle thread (where supported). This results in a lower overall
memory footprint for the application, as you can avoid having to declare
a thread object and stack for Idle functionality.

=====*/
extern "C" {
void __cxa_pure_virtual(void) { }
}

//-----
// This block declares the thread data for the main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP_STACK_SIZE      (320/sizeof(K_WORD))
static Thread   clAppThread;
static K_WORD   awAppStack[APP_STACK_SIZE];
static void     AppMain(void *unused_);

//-----
// This block declares the special function called from with the special
// Kernel-Idle context. We use the Kernel::SetIdleFunc() API to ensure that
// this function is called to provide our idle context.
static void     IdleMain(void);

//-----
int main(void)
{
    // See the annotations in lab1.
    Kernel::Init();

    // Initialize the main application thread, as in lab1. Note that even
    // though we're using an Idle function and not a dedicated thread, priority
    // level 0 is still reserved for idle functionality. Application threads
    // should never be scheduled at priority level 0 when the idle function is
    // used instead of an idle thread.
    clAppThread.Init( awAppStack, APP_STACK_SIZE, 1, AppMain, 0);
    clAppThread.Start();

    // This function is used to install our specified idle function to be called
    // whenever there are no ready threads in the system. Note that if no
    // Idle function is specified, a default will be used. Note that this default
    // function is essentially a null operation.
    Kernel::SetIdleFunc(IdleMain);

    Kernel::Start();

    return 0;
}

//-----
void AppMain(void *unused_)
{
    // Same as in lab1.
    while(1)
    {
        KernelAware::Print("Hello World!\n");
```

```

    }
    Thread::Sleep(1000);
}

//-----
void IdleMain(void)
{
    // Low priority task + power management routines go here.
    // The actions taken in this context must *not* cause a blocking call,
    // similar to the requirements for an idle thread.

    // Note that unlike an idle thread, the idle function must run to
    // completion. As this is also called from a nested interrupt context,
    // it's worthwhile keeping this function brief, limited to absolutely
    // necessary functionality, and with minimal stack use.
}

```

19.6 lab3_round_robin/main.cpp

This example demonstrates how to use round-robin thread scheduling with multiple threads of the same priority.

```

/*
=====
--[Mark3 Realtime Platform]-----
Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
=====*/
#include "mark3.h"
/*=====

Lab Example 3:  using round-robin scheduling to time-slice the CPU.

Lessons covered in this example include:
- Threads at the same priority get timesliced automatically
- The Thread::SetQuantum() API can be used to set the maximum amount of CPU
  time a thread can take before being swapped for another task at that
  priority level.

Takeaway:

- CPU Scheduling can be achieved using not just strict Thread priority, but
  also with round-robin time-slicing between threads at the same priority.
=====*/
extern "C" {
void __cxa_pure_virtual(void) { }
}

//-----
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE      (320/sizeof(K_WORD))
static Thread  clApp1Thread;
static K_WORD  awApp1Stack[APP1_STACK_SIZE];
static void    App1Main(void *unused_);

//-----
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE      (320/sizeof(K_WORD))
static Thread  clApp2Thread;
static K_WORD  awApp2Stack[APP2_STACK_SIZE];
static void    App2Main(void *unused_);

//-----
int main(void)
{
    // See the annotations in lab1.
    Kernel::Init();

    // In this exercise, we create two threads at the same priority level.

```


Lab Example 4: using binary semaphores

In this example, we implement two threads, synchronized using a semaphore to model the classic producer-consumer pattern. One thread does work, and then posts the semaphore indicating that the other thread can consume that work. The blocking thread just waits idly until there is data for it to consume.

Lessons covered in this example include:

- Use of a binary semaphore to implement the producer-consumer pattern
- Synchronization of threads (within a single priority, or otherwise) using a semaphore

Takeaway:

Semaphores can be used to control which threads execute at which time. This allows threads to work cooperatively to achieve a goal in the system.

```

=====*/
extern "C" {
void __cxa_pure_virtual(void) { }
}

//-----
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320/sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void *unused_);

//-----
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320/sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void *unused_);

//-----
// This is the semaphore that we'll use to synchronize two threads in this
// demo application
static Semaphore clMySem;

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    // In this example we create two threads to illustrate the use of a
    // binary semaphore as a synchronization method between two threads.

    // Thread 1 is a "consumer" thread -- It waits, blocked on the semaphore
    // until thread 2 is done doing some work. Once the semaphore is posted,
    // the thread is unblocked, and does some work.

    // Thread 2 is thus the "producer" thread -- It does work, and once that
    // work is done, the semaphore is posted to indicate that the other thread
    // can use the producer's work product.

    clApp1Thread.Init( awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init( awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    // Initialize a binary semaphore (maximum value of one, initial value of
    // zero).
    clMySem.Init(0,1);

    Kernel::Start();

    return 0;
}

//-----
void App1Main(void *unused_)
{
    while(1)
    {
        // Wait until the semaphore is posted from the other thread
        KernelAware::Print("Wait\n");
        clMySem.Pend();
    }
}

```



```

// This is the mutex that we'll use to synchronize two threads in this
// demo application.
static Mutex    clMyMutex;

// This counter variable is the "shared resource" in the example, protected
// by the mutex. Only one thread should be given access to the counter at
// any time.
static volatile uint32_t u32Counter = 0;

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init( awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init( awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    // Initialize the mutex used in this example.
    clMyMutex.Init();

    Kernel::Start();

    return 0;
}

//-----
void App1Main(void *unused_)
{
    while(1)
    {
        // Claim the mutex. This will prevent any other thread from claiming
        // this lock simultaneously. As a result, the other thread has to
        // wait until we're done before it can do its work. You will notice
        // that the Start/Done prints for the thread will come as a pair (i.e.
        // you won't see "Thread2: Start" then "Thread1: Start").

        clMyMutex.Claim();

        // Start our work (incrementing a counter). Notice that the Start and
        // Done prints wind up as a pair when simulated with flAVR.

        KernelAware::Print("Thread1: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000)
        {
            u32Counter++;
        }
        u32Counter = 0;
        KernelAware::Print("Thread1: Done\n");

        // Release the lock, allowing the other thread to do its thing.
        clMyMutex.Release();
    }
}

//-----
void App2Main(void *unused_)
{
    while(1)
    {
        // Claim the mutex. This will prevent any other thread from claiming
        // this lock simultaneously. As a result, the other thread has to
        // wait until we're done before it can do its work. You will notice
        // that the Start/Done prints for the thread will come as a pair (i.e.
        // you won't see "Thread2: Start" then "Thread1: Start").

        clMyMutex.Claim();

        // Start our work (incrementing a counter). Notice that the Start and
        // Done prints wind up as a pair when simulated with flAVR.

        KernelAware::Print("Thread2: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000)
        {
            u32Counter++;
        }
        u32Counter = 0;
        KernelAware::Print("Thread2: Done\n");

        // Release the lock, allowing the other thread to do its thing.
        clMyMutex.Release();
    }
}

```



```

Timer        clOneShot;    // One-shot timer object

Semaphore    clMySem;      // Semaphore used to wake this thread

// Initialize a binary semaphore (maximum value of one, initial value of
// zero).
clMySem.Init(0,1);

// Start a timer that triggers every 500ms that will call PeriodicCallback.
// This timer simulates an external stimulus or event that would require
// an action to be taken by this thread, but would be serviced by an
// interrupt or other high-priority context.

// PeriodicCallback will post the semaphore which wakes the thread
// up to perform an action. Here that action consists of a trivial message
// print.
clMyTimer.Start(true, 500, PeriodicCallback, (void*)&clMySem);

// Set up a one-shot timer to print a message after 2.5 seconds, asynchronously
// from the execution of this thread.
clOneShot.Start(false, 2500, OneShotCallback, 0);

while(1)
{
    // Wait until the semaphore is posted from the timer expiry
    clMySem.Pend();

    // Take some action after the timer posts the semaphore to wake this
    // thread.
    KernelAware::Print("Thread Triggered.\n");
}
}

```

19.10 lab7_events/main.cpp

This example demonstrates how to create and use event groups

[illegible]

```

static void    App1Main(void *unused_);

//-----
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE    (320/sizeof(K_WORD))
static Thread    clApp2Thread;
static K_WORD    awApp2Stack[APP2_STACK_SIZE];
static void    App2Main(void *unused_);

//-----
//
static EventFlag    clFlags;

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init( awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init( awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    clFlags.Init();

    Kernel::Start();

    return 0;
}

//-----
void App1Main(void *unused_)
{
    while(1)
    {
        uint16_t ul6Flags;

        // Block this thread until any of the event flags have been set by
        // some outside force (here, we use Thread 2). As an exercise to the
        // user, try playing around with the event mask to see the effect it
        // has on which events get processed. Different threads can block on
        // different bitmasks - this allows events with different real-time
        // priorities to be handled in different threads, while still using
        // the same event-flag object.

        // Also note that EVENT_FLAG_ANY indicates that the thread will be
        // unblocked whenever any of the flags in the mask are selected. If
        // you wanted to trigger an action that only takes place once multiple
        // bits are set, you could block the thread waiting for a specific
        // event bitmask with EVENT_FLAG_ALL specified.
        ul6Flags = clFlags.Wait(0xFFFF, EVENT_FLAG_ANY);

        // Print a message indicating which bit was set this time.
        switch (ul6Flags)
        {
            case 0x0001:
                KernelAware::Print("Event1\n");
                break;
            case 0x0002:
                KernelAware::Print("Event2\n");
                break;
            case 0x0004:
                KernelAware::Print("Event3\n");
                break;
            case 0x0008:
                KernelAware::Print("Event4\n");
                break;
            case 0x0010:
                KernelAware::Print("Event5\n");
                break;
            case 0x0020:
                KernelAware::Print("Event6\n");
                break;
            case 0x0040:
                KernelAware::Print("Event7\n");
                break;
            case 0x0080:
                KernelAware::Print("Event8\n");
                break;
            case 0x0100:
                KernelAware::Print("Event9\n");
                break;
            case 0x0200:

```

```

KernelAware::Print("Event10\n");
break;
case 0x0400:
    KernelAware::Print("Event11\n");
    break;
case 0x0800:
    KernelAware::Print("Event12\n");
    break;
case 0x1000:
    KernelAware::Print("Event13\n");
    break;
case 0x2000:
    KernelAware::Print("Event14\n");
    break;
case 0x4000:
    KernelAware::Print("Event15\n");
    break;
case 0x8000:
    KernelAware::Print("Event16\n");
    break;
default:
    break;
}

// Clear the event-flag that we just printed a message about. This
// will allow ul6 to acknowledge further events in that bit in the future.
clFlags.Clear(u16Flags);

}

}

//-----
void App2Main(void *unused_)
{
    uint16_t u16Flag = 1;
    while(1)
    {
        Thread::Sleep(100);

        // Event flags essentially map events to bits in a bitmap. Here we
        // set one bit each 100ms. In this loop, we cycle through bits 0-15
        // repeatedly. Note that this will wake the other thread, which is
        // blocked, waiting for *any* of the flags in the bitmap to be set.
        clFlags.Set(u16Flag);

        // Bitshift the flag value to the left. This will be the flag we set
        // the next time this thread runs through its loop.
        if (u16Flag != 0x8000)
        {
            u16Flag <<= 1;
        }
        else
        {
            u16Flag = 1;
        }
    }
}
}

```

19.11 lab8_messages/main.cpp

This example demonstrates how to pass data between threads using message passing.

```
/*  
 _   _   _   _   _   _   _   _  
| | | | | | | | | | | | | |  
|_| |_|_|_|_|_|_|_|_|_|_|_|  
  
--[Mark3 Realtime Platform]-----  
  
Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.  
See license.txt for more information  
=====*/  
#include "mark3.h"  
  
/*=====*/
```

Lab Example 8: using messages for IPC.

In this example, we present a typical asynchronous producer/consumer pattern

using Mark3's message-driven IPC.

Lessons covered in this example include:

- use of Message and MessageQueue objects to send data between threads
- use of GlobalMessagePool to allocate and free message objects

Takeaway:

Unlike cases presented in previous examples that relied on semaphores or event flags, messages carry substantial context, specified in its "code" and "data" members. This mechanism can be used to pass data between threads extremely efficiently, with a simple and flexible API. Any number of threads can write to/block on a single message queue, which give this method of IPC even more flexibility.

```

=====*/
extern "C" {
void __cxa_pure_virtual(void) { }
}

//-----
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE      (320/sizeof(K_WORD))
static Thread  clApp1Thread;
static K_WORD  awApp1Stack[APP1_STACK_SIZE];
static void    App1Main(void *unused_);

//-----
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE      (320/sizeof(K_WORD))
static Thread  clApp2Thread;
static K_WORD  awApp2Stack[APP2_STACK_SIZE];
static void    App2Main(void *unused_);

//-----
static MessageQueue clMsgQ;

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init( awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init( awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    clMsgQ.Init();

    Kernel::Start();

    return 0;
}

//-----
void App1Main(void *unused_)
{
    uint16_t ul6Data = 0;
    while(1)
    {
        // This thread grabs a message from the global message pool, sets a
        // code-value and the message data pointer, then sends the message to
        // a message queue object. Another thread (Thread2) is blocked, waiting
        // for a message to arrive in the queue.

        // Get the message object
        Message *pclMsg = GlobalMessagePool::Pop();

        // Set the message object's data (contrived in this example)
        pclMsg->SetCode(0x1337);
        ul6Data++;
        pclMsg->SetData(&ul6Data);

        // Send the message to the shared message queue
        clMsgQ.Send(pclMsg);

        // Wait before sending another message.
        Thread::Sleep(200);
    }
}

```

```
//-----
void App2Main(void *unused_)
{
    while(1)
    {
        // This thread waits until it receives a message on the shared global
        // message queue. When it gets the message, it prints out information
        // about the message's code and data, before returning the message object
        // back to the global message pool. In a more practical application,
        // the user would typically use the code to tell the receiving thread
        // what kind of message was sent, and what type of data to expect in the
        // data field.

        // Wait for a message to arrive on the specified queue. Note that once
        // this thread receives the message, it is "owned" by the thread, and
        // must be returned back to its source message pool when it is no longer
        // needed.
        Message *pclMsg = clMsgQ.Receive();

        // We received a message, now print out its information
        KernelAware::Print("Received Message\n");
        KernelAware::Trace(0, __LINE__, pclMsg->GetCode(), *((uint16_t*)pclMsg->
        GetData()) );

        // Done with the message, return it back to the global message queue.
        GlobalMessagePool::Push(pclMsg);
    }
}
```

19.12 lab9_dynamic_threads/main.cpp

This example demonstrates how to create and destroy threads dynamically at runtime.

```

/*
  |-----|-----|-----|-----|-----|
  | \ / | | | \ / | | | \ / | | | \ / | | |
  | / \ | | | / \ | | | / \ | | | / \ | | |
  |-----|-----|-----|-----|-----|
*/

--[Mark3 Realtime Platform]-----

Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
=====*/
#include "mark3.h"
/*=====

Lab Example 9:  Dynamic Threading

Lessons covered in this example include:
- Creating, pausing, and destorying dynamically-created threads at runtime

Takeaway:

In addition to being able to specify a static set of threads during system
initialization, Mark3 gives the user the ability to create and manipu32ate
threads at runtime.  These threads can act as "temporary workers" that can
be activated when needed, without impacting the responsiveness of the rest
of the application.

=====*/
extern "C" {
void __cxa_pure_virtual(void) { }
}

//-----
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE      (320/sizeof(K_WORD))
static Thread  clApp1Thread;
static K_WORD  awApp1Stack[APP1_STACK_SIZE];
static void    App1Main(void *unused_);

//-----
// This block declares the thread stack data for a thread that we'll create
// dynamically.
#define APP2_STACK_SIZE      (320/sizeof(K_WORD))

```

```

static K_WORD  awApp2Stack[APP2_STACK_SIZE];

//-----
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init( awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp1Thread.Start();

    Kernel::Start();

    return 0;
}

//-----
static void WorkerMain1(void *arg_)
{
    Semaphore *pclSem = (Semaphore*)arg_;
    uint32_t u32Count = 0;

    // Do some work. Post a semaphore to notify the other thread that the
    // work has been completed.
    while (u32Count < 1000000)
    {
        u32Count++;
    }

    KernelAware::Print( "Worker1 -- Done Work\n");
    pclSem->Post();

    // Work is completed, just spin now. Let another thread destroy u16.
    while(1) { }
}

//-----
static void WorkerMain2(void *arg_)
{
    uint32_t u32Count = 0;
    while (u32Count < 1000000)
    {
        u32Count++;
    }

    KernelAware::Print( "Worker2 -- Done Work\n");

    // A dynamic thread can self-terminate as well:
    Scheduler::GetCurrentThread()->Exit();
}

//-----
void App1Main(void *unused_)
{
    Thread clMyThread;
    Semaphore clMySem;

    clMySem.Init(0,1);
    while (1)
    {
        // Example 1 - create a worker thread at our current priority in order to
        // parallelize some work.
        clMyThread.Init( awApp2Stack, APP2_STACK_SIZE, 1, WorkerMain1, (void*)&clMySem );
        clMyThread.Start();

        // Do some work of our own in parallel, while the other thread works on its project.
        uint32_t u32Count = 0;
        while (u32Count < 100000)
        {
            u32Count++;
        }

        KernelAware::Print( "Thread -- Done Work\n" );

        // Wait for the other thread to finish its job.
        clMySem.Pend();

        // Once the thread has signalled u16, we can safely call "Exit" on the thread to
        // remove it from scheduling and recycle it later.
        clMyThread.Exit();

        // Spin the thread up again to do something else in parallel. This time, the thread
        // will run completely asynchronously to this thread.
        clMyThread.Init( awApp2Stack, APP2_STACK_SIZE, 1, WorkerMain2, 0 );
        clMyThread.Start();

        u32Count = 0;
        while (u32Count < 1000000)

```

```
{
    u32Count++;
}

KernelAware::Print( "Thread -- Done Work\n" );

// Check that we're sure the worker thread has terminated before we try running the
// test loop again.
while (clMyThread.GetState() != THREAD_STATE_EXIT) { }

KernelAware::Print ( "  Test Done\n");
Thread::Sleep(1000);
}
}
```

Index

- Add
 - Scheduler, [139](#)
- Claim
 - Mutex, [128](#), [129](#)
- Close
 - Driver, [89](#)
- Control
 - Driver, [89](#)
- Driver, [88](#)
 - Close, [89](#)
 - Control, [89](#)
 - Init, [91](#)
 - Open, [91](#)
 - Read, [91](#)
 - Write, [92](#)
- EVENT_FLAG_ALL
 - kerneltypes.h, [232](#)
- EVENT_FLAG_ALL_CLEAR
 - kerneltypes.h, [232](#)
- EVENT_FLAG_ANY
 - kerneltypes.h, [232](#)
- EVENT_FLAG_ANY_CLEAR
 - kerneltypes.h, [232](#)
- EVENT_FLAG_MODES
 - kerneltypes.h, [232](#)
- EVENT_FLAG_PENDING_UNBLOCK
 - kerneltypes.h, [232](#)
- Exit
 - Thread, [147](#)
- Init
 - Driver, [91](#)
 - Kernel, [102](#)
 - Mailbox, [118](#)
 - Message, [124](#)
 - Mutex, [129](#)
 - Notify, [131](#)
 - Profiler, [133](#)
 - Scheduler, [140](#)
 - Semaphore, [142](#)
 - Thread, [150](#)
 - Timer, [159](#)
- KA_COMMAND_EXIT_SIMULATOR
 - kernelaware.h, [226](#)
- KA_COMMAND_IDLE
 - kernelaware.h, [226](#)
- KA_COMMAND_PRINT
 - kernelaware.h, [226](#)
- kernelaware.h, [226](#)
 - KA_COMMAND_PROFILE_INIT
 - kernelaware.h, [226](#)
 - KA_COMMAND_PROFILE_REPORT
 - kernelaware.h, [226](#)
 - KA_COMMAND_PROFILE_START
 - kernelaware.h, [226](#)
 - KA_COMMAND_PROFILE_STOP
 - kernelaware.h, [226](#)
 - KA_COMMAND_TRACE_0
 - kernelaware.h, [226](#)
 - KA_COMMAND_TRACE_1
 - kernelaware.h, [226](#)
 - KA_COMMAND_TRACE_2
 - kernelaware.h, [226](#)
- Kernel, [101](#)
 - Init, [102](#)
 - Panic, [103](#)
 - Start, [103](#)
- kerneltypes.h
 - KA_COMMAND_EXIT_SIMULATOR, [226](#)
 - KA_COMMAND_IDLE, [226](#)
 - KA_COMMAND_PRINT, [226](#)
 - KA_COMMAND_PROFILE_INIT, [226](#)
 - KA_COMMAND_PROFILE_REPORT, [226](#)
 - KA_COMMAND_PROFILE_START, [226](#)
 - KA_COMMAND_PROFILE_STOP, [226](#)
 - KA_COMMAND_TRACE_0, [226](#)
 - KA_COMMAND_TRACE_1, [226](#)
 - KA_COMMAND_TRACE_2, [226](#)
- Mailbox, [116](#)
 - Init, [118](#)
 - Receive, [119](#)
 - Send, [120](#), [121](#)
- Message, [122](#)
 - Init, [124](#)
- Mutex, [127](#)
 - Claim, [128](#), [129](#)
 - Init, [129](#)
 - Release, [129](#)
- Notify, [130](#)

- Init, [131](#)
- Signal, [131](#)
- Wait, [131](#), [132](#)
- Open
 - Driver, [91](#)
- Panic
 - Kernel, [103](#)
- Pend
 - Semaphore, [142](#), [143](#)
- Post
 - Semaphore, [143](#)
- Process
 - Profiler, [133](#)
- Profiler, [132](#)
 - Init, [133](#)
 - Process, [133](#)
 - Read, [133](#)
 - Start, [133](#)
 - Stop, [133](#)
- Quantum, [136](#)
- Read
 - Driver, [91](#)
 - Profiler, [133](#)
- Receive
 - Mailbox, [119](#)
- Release
 - Mutex, [129](#)
- Remove
 - Scheduler, [140](#)
- Schedule
 - Scheduler, [140](#)
- Scheduler, [138](#)
 - Add, [139](#)
 - Init, [140](#)
 - Remove, [140](#)
 - Schedule, [140](#)
- Semaphore, [141](#)
 - Init, [142](#)
 - Pend, [142](#), [143](#)
 - Post, [143](#)
- Send
 - Mailbox, [120](#), [121](#)
- Signal
 - Notify, [131](#)
- Sleep
 - Thread, [152](#)
- Start
 - Kernel, [103](#)
 - Profiler, [133](#)
 - Thread, [152](#)
 - Timer, [161](#)
- Stop
 - Profiler, [133](#)
 - Thread, [153](#)
- Timer, [162](#)
- Thread, [144](#)
 - Exit, [147](#)
 - Init, [150](#)
 - Sleep, [152](#)
 - Start, [152](#)
 - Stop, [153](#)
 - Yield, [153](#)
- Timer, [157](#)
 - Init, [159](#)
 - Start, [161](#)
 - Stop, [162](#)
 - Timer, [159](#)
- Wait
 - Notify, [131](#), [132](#)
- Write
 - Driver, [92](#)
- Yield
 - Thread, [153](#)