Mark3 Realtime Kernel

Generated by Doxygen 1.8.6

Wed May 27 2015 21:18:26

Contents

1	The	Mark3 Realtime Kernel	1
2	Pref	ace	3
	2.1	Who should read this	3
	2.2	Why Mark3?	3
3	Can	you Afford an RTOS?	5
	3.1	Intro	5
	3.2	Memory overhead:	6
	3.3	Code Space Overhead:	7
	3.4	Runtime Overhead	7
4	Sup	erloops erloops	9
	4.1	Intro to Superloops	9
	4.2	The simplest loop	9
	4.3	Interrupt-Driven Super-loop	10
	4.4	Cooperative multi-tasking	11
	4.5	Hybrid cooperative/preemptive multi-tasking	12
	4.6	Problems with superloops	13
5	Mari	k3 Overview	15
	5.1	Intro	15
	5.2	Features	15
	5.3	Design Goals	16
6	Gett	ing Started	17
	6.1	Kernel Setup	17
	6.2	Threads	18
		6.2.1 Thread Setup	18
		6.2.2 Entry Functions	19
	6.3	Timers	19
	6.4	Semaphores	20
	6 5	Mutavas	01

iv CONTENTS

	6.6	Event Flags	21
	6.7	Messages	22
		6.7.1 Message Objects	22
		6.7.2 Global Message Pool	23
		6.7.3 Message Queues	23
		6.7.4 Messaging Example	23
	6.8	Sleep	24
	6.9	Round-Robin Quantum	24
7	Build	d System	25
	7.1	Source Layout	25
	7.2	Building the kernel	25
	7.3	Building on Windows	27
_			
8	Lice		29
	8.1	License	29
9	Profi	iling Results	31
	9.1	Date Performed	31
	9.2	Compiler Information	31
	9.3	Profiling Results	31
10	Code	e Size Profiling	33
		Information	33
	10.2	Compiler Version	
			33
		Profiling Results	33 33
	10.3	Profiling Results	33
11	10.3	Profiling Results	33 35
11	10.3	Profiling Results	33
	10.3 Hiera 11.1	Profiling Results	33 35
	10.3 Hiera 11.1 Class	Profiling Results	33 35 35
12	10.3 Hiera 11.1 Class 12.1	Profiling Results	33 35 35 37
12	10.3 Hiera 11.1 Class 12.1 File I	Profiling Results	33 35 35
12	10.3 Hiera 11.1 Class 12.1 File I	Profiling Results	33 35 35 37 37
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class	Profiling Results archical Index Class Hierarchy s Index Class List Index File List S Documentation	33 35 35 37 37
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class	Profiling Results	33 35 37 37 39
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class	Profiling Results archical Index Class Hierarchy s Index Class List Index File List s Documentation BlockingObject Class Reference 14.1.1 Detailed Description	33 35 35 37 37 39 39
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class	Profiling Results archical Index Class Hierarchy s Index Class List Index File List s Documentation BlockingObject Class Reference 14.1.1 Detailed Description 14.1.2 Member Function Documentation	33 35 37 37 39 39 43
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class	Profiling Results archical Index Class Hierarchy s Index Class List Index File List s Documentation BlockingObject Class Reference 14.1.1 Detailed Description 14.1.2 Member Function Documentation 14.1.2.1 Block	33 35 37 37 39 39 43 43
12 13	10.3 Hiera 11.1 Class 12.1 File I 13.1 Class 14.1	Profiling Results archical Index Class Hierarchy s Index Class List Index File List s Documentation BlockingObject Class Reference 14.1.1 Detailed Description 14.1.2 Member Function Documentation	33 35 35 37 39 39 43 43 43

CONTENTS

	14.2.1	Detailed Description	45
	14.2.2	Member Function Documentation	45
		14.2.2.1 Add	45
		14.2.2.2 Remove	45
14.3	DevNu	Il Class Reference	45
	14.3.1	Detailed Description	46
	14.3.2	Member Function Documentation	46
		14.3.2.1 Close	46
		14.3.2.2 Control	46
		14.3.2.3 Open	47
		14.3.2.4 Read	47
		14.3.2.5 Write	47
14.4	Double	LinkList Class Reference	48
	14.4.1	Detailed Description	48
	14.4.2	Member Function Documentation	48
		14.4.2.1 Add	48
		14.4.2.2 Remove	49
14.5	Driver (Class Reference	49
	14.5.1	Detailed Description	50
	14.5.2	Member Function Documentation	50
		14.5.2.1 Close	50
		14.5.2.2 Control	50
		14.5.2.3 GetPath	51
		14.5.2.4 Open	51
		14.5.2.5 Read	51
		14.5.2.6 SetName	51
		14.5.2.7 Write	51
14.6	DriverL	ist Class Reference	52
	14.6.1	Detailed Description	52
	14.6.2	Member Function Documentation	52
		14.6.2.1 Add	52
		14.6.2.2 FindByPath	53
		14.6.2.3 Init	53
		14.6.2.4 Remove	53
14.7	EventF	lag Class Reference	53
	14.7.1	Detailed Description	54
	14.7.2	Member Function Documentation	54
		14.7.2.1 Clear	54
		14.7.2.2 GetMask	54
		14.7.2.3 Set	55

vi CONTENTS

14.7.2.4 Wait	55
14.7.2.5 Wait	55
14.7.2.6 Wait_i	55
14.7.2.7 WakeMe	56
14.8 FakeThread_t Struct Reference	56
14.8.1 Detailed Description	57
14.9 GlobalMessagePool Class Reference	57
14.9.1 Detailed Description	57
14.9.2 Member Function Documentation	57
14.9.2.1 Pop	57
14.9.2.2 Push	58
14.10Kernel Class Reference	58
14.10.1 Detailed Description	59
14.10.2 Member Function Documentation	59
14.10.2.1 GetIdleThread	59
14.10.2.2 Init	59
14.10.2.3 IsPanic	59
14.10.2.4 IsStarted	59
14.10.2.5 Panic	59
14.10.2.6 SetIdleFunc	60
14.10.2.7 SetPanic	60
14.10.2.8 Start	60
14.11 Kernel Aware Class Reference	60
14.11.1 Detailed Description	61
14.11.2 Member Function Documentation	61
14.11.2.1 ExitSimulator	61
14.11.2.2 IsSimulatorAware	61
14.11.2.3 Print	62
14.11.2.4 ProfileInit	62
14.11.2.5 ProfileReport	62
14.11.2.6 ProfileStart	62
14.11.2.7 ProfileStop	62
14.11.2.8 Trace	62
14.11.2.9 Trace	63
14.11.2.10Trace	63
14.11.2.11Trace_i	63
14.12KernelAwareData_t Union Reference	64
14.12.1 Detailed Description	64
14.13KernelSWI Class Reference	64
14.13.1 Detailed Description	65

CONTENTS vii

14.13.2 Member Function Documentation	65
14.13.2.1 DI	65
14.13.2.2 RI	65
14.14KernelTimer Class Reference	65
14.14.1 Detailed Description	66
14.14.2 Member Function Documentation	66
14.14.2.1 GetOvertime	66
14.14.2.2 Read	66
14.14.2.3 RI	66
14.14.2.4 SetExpiry	67
14.14.2.5 SubtractExpiry	67
14.14.2.6 TimeToExpiry	67
14.15LinkList Class Reference	67
14.15.1 Detailed Description	68
14.15.2 Member Function Documentation	68
14.15.2.1 Add	68
14.15.2.2 GetHead	68
14.15.2.3 GetTail	69
14.15.2.4 Remove	69
14.16LinkListNode Class Reference	69
14.16.1 Detailed Description	70
14.16.2 Member Function Documentation	70
14.16.2.1 GetNext	70
14.16.2.2 GetPrev	70
14.17Message Class Reference	70
14.17.1 Detailed Description	71
14.17.2 Member Function Documentation	71
14.17.2.1 GetCode	71
14.17.2.2 GetData	71
14.17.2.3 SetCode	72
14.17.2.4 SetData	72
14.18MessageQueue Class Reference	72
14.18.1 Detailed Description	73
14.18.2 Member Function Documentation	73
14.18.2.1 GetCount	73
14.18.2.2 Receive	73
14.18.2.3 Receive	73
14.18.2.4 Receive_i	73
14.18.2.5 Send	74
14.19Mutex Class Reference	74

viii CONTENTS

14.19.1 Detailed Description	75
14.19.2 Member Function Documentation	75
14.19.2.1 Claim	75
14.19.2.2 Claim	75
14.19.2.3 Claim_i	75
14.19.2.4 Release	76
14.19.2.5 WakeMe	76
14.20 Profiler Class Reference	76
14.20.1 Detailed Description	77
14.20.2 Member Function Documentation	77
14.20.2.1 Init	77
14.21 ProfileTimer Class Reference	77
14.21.1 Detailed Description	78
14.21.2 Member Function Documentation	78
14.21.2.1 ComputeCurrentTicks	78
14.21.2.2 GetAverage	78
14.21.2.3 GetCurrent	78
14.21.2.4 Init	79
14.21.2.5 Start	79
14.22Quantum Class Reference	79
14.22.1 Detailed Description	79
14.22.2 Member Function Documentation	80
14.22.2.1 AddThread	80
14.22.2.2 ClearInTimer	80
14.22.2.3 RemoveThread	80
14.22.2.4 SetInTimer	80
14.22.2.5 SetTimer	80
14.22.2.6 UpdateTimer	80
14.23 Scheduler Class Reference	81
14.23.1 Detailed Description	81
14.23.2 Member Function Documentation	82
14.23.2.1 Add	82
14.23.2.2 GetCurrentThread	83
14.23.2.3 GetNextThread	83
14.23.2.4 GetStopList	83
14.23.2.5 GetThreadList	83
14.23.2.6 IsEnabled	83
14.23.2.7 QueueScheduler	84
14.23.2.8 Remove	84
14.23.2.9 Schedule	84

CONTENTS

14.23.2.10SetScheduler	84
14.24Semaphore Class Reference	84
14.24.1 Detailed Description	85
14.24.2 Member Function Documentation	85
14.24.2.1 GetCount	85
14.24.2.2 Init	86
14.24.2.3 Pend	87
14.24.2.4 Pend	87
14.24.2.5 Pend_i	87
14.24.2.6 Post	87
14.24.2.7 WakeMe	87
14.25Thread Class Reference	88
14.25.1 Detailed Description	90
14.25.2 Member Function Documentation	91
14.25.2.1 ContextSwitchSWI	91
14.25.2.2 Exit	91
14.25.2.3 GetCurPriority	91
14.25.2.4 GetCurrent	91
14.25.2.5 GetEventFlagMask	91
14.25.2.6 GetEventFlagMode	91
14.25.2.7 GetExpired	92
14.25.2.8 GetID	92
14.25.2.9 GetOwner	92
14.25.2.10GetPriority	92
14.25.2.11GetQuantum	92
14.25.2.12GetStackSlack	93
14.25.2.13GetState	93
14.25.2.14InheritPriority	93
14.25.2.15nit	93
14.25.2.16nitIdle	93
14.25.2.17SetCurrent	94
14.25.2.1&SetEventFlagMask	94
14.25.2.19SetEventFlagMode	94
14.25.2.20SetExpired	94
14.25.2.21SetID	94
14.25.2.2SetOwner	94
14.25.2.23SetPriority	95
14.25.2.24SetPriorityBase	95
14.25.2.25SetQuantum	95
14.25.2.26SetState	95

CONTENTS

14.25.2.27Sleep	95
14.25.2.28Stop	96
14.25.2.29USleep	96
14.25.2.30Yield	96
14.26ThreadList Class Reference	96
14.26.1 Detailed Description	97
14.26.2 Member Function Documentation	97
14.26.2.1 Add	97
14.26.2.2 Add	97
14.26.2.3 HighestWaiter	98
14.26.2.4 Remove	98
14.26.2.5 SetFlagPointer	98
14.26.2.6 SetPriority	98
14.27ThreadPort Class Reference	98
14.27.1 Detailed Description	99
14.27.2 Member Function Documentation	99
14.27.2.1 InitStack	99
14.28Timer Class Reference	99
14.28.1 Detailed Description	101
14.28.2 Member Function Documentation	101
14.28.2.1 SetCallback	101
14.28.2.2 SetData	101
14.28.2.3 SetFlags	101
14.28.2.4 SetIntervalMSeconds	101
14.28.2.5 SetIntervalSeconds	101
14.28.2.6 SetIntervalTicks	102
14.28.2.7 SetIntervalUSeconds	102
14.28.2.8 SetOwner	102
14.28.2.9 SetTolerance	102
14.28.2.10Start	102
14.28.2.11Start	103
14.28.2.12Stop	103
14.29TimerList Class Reference	103
14.29.1 Detailed Description	104
14.29.2 Member Function Documentation	104
14.29.2.1 Add	104
14.29.2.2 Init	104
14.29.2.3 Process	104
14.29.2.4 Remove	104
14.30TimerScheduler Class Reference	105

CONTENTS xi

	14.30.1 Detailed Description	105
	14.30.2 Member Function Documentation	105
	14.30.2.1 Add	105
	14.30.2.2 Init	106
	14.30.2.3 Process	106
	14.30.2.4 Remove	106
15	File Documentation	107
	15.1 /home/moslevin/Project/R1/kernel/atomic.cpp File Reference	107
	15.1.1 Detailed Description	107
	15.2 atomic.cpp	107
	15.3 /home/moslevin/Project/R1/kernel/blocking.cpp File Reference	109
	15.3.1 Detailed Description	109
	15.4 blocking.cpp	109
	15.5 /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp File Reference	110
	15.5.1 Detailed Description	110
	15.6 kernelprofile.cpp	110
	15.7 /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp File Reference	111
	15.7.1 Detailed Description	112
	15.8 kernelswi.cpp	112
	15.9 /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp File Reference	113
	15.9.1 Detailed Description	113
	15.10kerneltimer.cpp	113
	15.11/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h File Reference	115
	15.11.1 Detailed Description	115
	15.12kernelprofile.h	115
	15.13/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h File Reference	116
	15.13.1 Detailed Description	116
	15.14kernelswi.h	116
	15.15/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h File Reference	117
	15.15.1 Detailed Description	117
	15.16kerneltimer.h	117
	15.17/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/threadport.h File Reference	118
	15.17.1 Detailed Description	119
	15.17.2 Macro Definition Documentation	119
	15.17.2.1 CS_ENTER	119
	15.18threadport.h	119
	15.19/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/threadport.cpp File Reference	121
	15.19.1 Detailed Description	122
	15.20threadport.cpp	122

xii CONTENTS

15.21/home/moslevin/Project/R1/kernel/driver.cpp File Reference	24
15.21.1 Detailed Description	24
15.21.2 Function Documentation	24
15.21.2.1 DrvCmp	24
15.22driver.cpp	25
15.23/home/moslevin/Project/R1/kernel/eventflag.cpp File Reference	26
15.23.1 Detailed Description	26
15.23.2 Function Documentation	26
15.23.2.1 TimedEventFlag_Callback	27
15.24eventflag.cpp	27
15.25/home/moslevin/Project/R1/kernel/kernel.cpp File Reference	30
15.25.1 Detailed Description	31
15.26kernel.cpp	31
15.27/home/moslevin/Project/R1/kernel/kernelaware.cpp File Reference	32
15.27.1 Detailed Description	33
15.27.2 Variable Documentation	33
15.27.2.1 g_blsKernelAware	33
15.27.2.2 g_stKAData	33
15.28kernelaware.cpp	33
15.29/home/moslevin/Project/R1/kernel/ksemaphore.cpp File Reference	35
15.29.1 Detailed Description	35
15.29.2 Function Documentation	35
15.29.2.1 TimedSemaphore_Callback	35
15.30ksemaphore.cpp	36
15.31/home/moslevin/Project/R1/kernel/II.cpp File Reference	38
15.31.1 Detailed Description	39
15.32ll.cpp	39
15.33/home/moslevin/Project/R1/kernel/message.cpp File Reference	41
15.33.1 Detailed Description	41
15.34message.cpp	41
15.35/home/moslevin/Project/R1/kernel/mutex.cpp File Reference	43
15.35.1 Detailed Description	44
15.35.2 Function Documentation	44
15.35.2.1 TimedMutex_Calback	44
15.36mutex.cpp	44
15.37/home/moslevin/Project/R1/kernel/profile.cpp File Reference	47
15.37.1 Detailed Description	47
15.38profile.cpp	48
15.39/home/moslevin/Project/R1/kernel/public/atomic.h File Reference	49
15.39.1 Detailed Description	49

CONTENTS xiii

15.40atomic.h
15.41/home/moslevin/Project/R1/kernel/public/blocking.h File Reference
15.41.1 Detailed Description
15.42blocking.h
15.43/home/moslevin/Project/R1/kernel/public/debugtokens.h File Reference
15.43.1 Detailed Description
15.44debugtokens.h
15.45/home/moslevin/Project/R1/kernel/public/driver.h File Reference
15.45.1 Detailed Description
15.45.2 Intro
15.45.3 Driver Design
15.45.4 Driver API
15.46driver.h
15.47/home/moslevin/Project/R1/kernel/public/eventflag.h File Reference
15.47.1 Detailed Description
15.48eventflag.h
15.49/home/moslevin/Project/R1/kernel/public/kernel.h File Reference
15.49.1 Detailed Description
15.50kernel.h
15.51/home/moslevin/Project/R1/kernel/public/kernelaware.h File Reference
15.51.1 Detailed Description
15.51.2 Enumeration Type Documentation
15.51.2.1 KernelAwareCommand_t
15.52kernelaware.h
15.53/home/moslevin/Project/R1/kernel/public/kerneldebug.h File Reference
15.53.1 Detailed Description
15.54kerneldebug.h
15.55/home/moslevin/Project/R1/kernel/public/kerneltypes.h File Reference
15.55.1 Detailed Description
15.55.2 Enumeration Type Documentation
15.55.2.1 EventFlagOperation_t
15.56kerneltypes.h
15.57/home/moslevin/Project/R1/kernel/public/ksemaphore.h File Reference
15.57.1 Detailed Description
15.58ksemaphore.h
15.59/home/moslevin/Project/R1/kernel/public/II.h File Reference
15.59.1 Detailed Description
15.60ll.h
15.61/home/moslevin/Project/R1/kernel/public/manual.h File Reference
15.61.1 Detailed Description

XIV

15.62 manual.h					
15.63/home/moslevin/Project/R1/kernel/public/mark3.h File Reference					
15.63.1 Detailed Description					
15.64mark3.h					
15.65/home/moslevin/Project/R1/kernel/public/mark3cfg.h File Reference					
15.65.1 Detailed Description					
15.65.2 Macro Definition Documentation					
15.65.2.1 GLOBAL_MESSAGE_POOL_SIZE					
15.65.2.2 KERNEL_AWARE_SIMULATION					
15.65.2.3 KERNEL_TIMERS_TICKLESS					
15.65.2.4 KERNEL_USE_ATOMIC					
15.65.2.5 KERNEL_USE_DYNAMIC_THREADS					
15.65.2.6 KERNEL_USE_EVENTFLAG					
15.65.2.7 KERNEL_USE_IDLE_FUNC					
15.65.2.8 KERNEL_USE_MESSAGE					
15.65.2.9 KERNEL_USE_PROFILER					
15.65.2.10KERNEL_USE_QUANTUM					
15.65.2.11KERNEL_USE_SEMAPHORE					
15.65.2.12KERNEL_USE_THREADNAME					
15.65.2.13KERNEL_USE_TIMEOUTS					
15.65.2.14KERNEL_USE_TIMERS					
15.65.2.15SAFE_UNLINK					
15.65.2.16THREAD_QUANTUM_DEFAULT					
15.66mark3cfg.h					
15.67/home/moslevin/Project/R1/kernel/public/message.h File Reference					
15.67.1 Detailed Description					
15.67.2 Using Messages, Queues, and the Global Message Pool					
15.68message.h					
15.69/home/moslevin/Project/R1/kernel/public/mutex.h File Reference					
15.69.1 Detailed Description					
15.69.2 Initializing					
15.69.3 Resource protection example					
15.70 mutex.h					
15.71/home/moslevin/Project/R1/kernel/public/paniccodes.h File Reference					
15.71.1 Detailed Description					
15.72 paniccodes.h					
15.73/home/moslevin/Project/R1/kernel/public/profile.h File Reference					
15.73.1 Detailed Description					
15.74profile.h					
15.75/home/moslevin/Project/R1/kernel/public/quantum.h File Reference					

CONTENTS xv

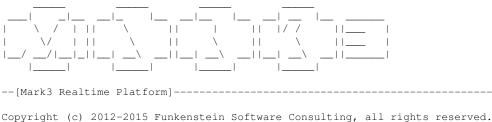
15.75.1 Detailed Description
15.76quantum.h
15.77/home/moslevin/Project/R1/kernel/public/scheduler.h File Reference
15.77.1 Detailed Description
15.78scheduler.h
15.79/home/moslevin/Project/R1/kernel/public/thread.h File Reference
15.79.1 Detailed Description
15.80thread.h
15.81/home/moslevin/Project/R1/kernel/public/threadlist.h File Reference
15.81.1 Detailed Description
15.82threadlist.h
15.83/home/moslevin/Project/R1/kernel/public/timer.h File Reference
15.83.1 Detailed Description
15.83.2 Macro Definition Documentation
15.83.2.1 TIMERLIST_FLAG_EXPIRED
15.83.3 Typedef Documentation
15.83.3.1 TimerCallback_t
15.84timer.h
15.85/home/moslevin/Project/R1/kernel/public/timerlist.h File Reference
15.85.1 Detailed Description
15.86timerlist.h
15.87/home/moslevin/Project/R1/kernel/public/timerscheduler.h File Reference
15.87.1 Detailed Description
15.88timerscheduler.h
15.89/home/moslevin/Project/R1/kernel/public/tracebuffer.h File Reference
15.89.1 Detailed Description
15.90tracebuffer.h
15.91/home/moslevin/Project/R1/kernel/public/writebuf16.h File Reference
15.91.1 Detailed Description
15.92writebuf16.h
15.93/home/moslevin/Project/R1/kernel/quantum.cpp File Reference
15.93.1 Detailed Description
15.93.2 Function Documentation
15.93.2.1 QuantumCallback
15.94quantum.cpp
15.95/home/moslevin/Project/R1/kernel/scheduler.cpp File Reference
15.95.1 Detailed Description
15.95.2 Variable Documentation
15.95.2.1 aucCLZ
15.96scheduler.cpp

xvi CONTENTS

1	15.97/home/moslevin/Project/R1/kernel/thread.cpp File Reference	198
	15.97.1 Detailed Description	199
1	15.98thread.cpp	199
1	15.99/home/moslevin/Project/R1/kernel/threadlist.cpp File Reference	204
	15.99.1 Detailed Description	204
1	15.10 6 hreadlist.cpp	204
1	15.10/lhome/moslevin/Project/R1/kernel/timer.cpp File Reference	206
	15.101.1Detailed Description	206
1	15.10 £ mer.cpp	206
1	15.102 home/moslevin/Project/R1/kernel/timerlist.cpp File Reference	207
	15.103. Detailed Description	208
1	15.10 ≰ merlist.cpp	208
1	15.105 nome/moslevin/Project/R1/kernel/tracebuffer.cpp File Reference	211
	15.105. Detailed Description	211
1	15.10 6 acebuffer.cpp	211
1	15.107/home/moslevin/Project/R1/kernel/writebuf16.cpp File Reference	212
	15.107. Detailed Description	212
1	15.10&ritebuf16.cpp	212
Inde	eX	215

Chapter 1

The Mark3 Realtime Kernel



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

The Mark3 Realtime Kernel is a completely free, open-source, real-time operating system aimed at bringing multi-tasking to microcontroller systems without MMUs.

It uses modern programming languages and concepts (it's written entirely in C_{++}) to minimize code duplication, and its object-oriented design enhances readibility. The API is simple - there are only six functions required to set up the kernel, initialize 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, specifically, Funkenstein Software's own flAVR AVR simulator

2	The Mark3 Realtime Kernel

Chapter 2

Preface

2.1 Who should read this

As the cover clearly states, this is a book about the Mark3 real-time kernel. I assume that if you're reading this book you have an interest in some, if not all, of the following subjects:

- · Embedded systems
- · Real-time systems
- · Operating system kernel design

And if you're interested in those topics, you're likely familiar with C and C++ and the more you know, the easier you'll find this book to read. And if C++ scares you, and you don't like embedded, real-time systems, you're probably looking for another book. If you're unfamiliar with RTOS fundamentals, I highly suggest searching through the vast amount of RTOS-related articles on the internet to familiarize yourself with the concepts.

2.2 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

4 Preface

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 (althrough 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 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, without incurring any extra overhead. Comparing the same configuration of Mark2 to Mark3, the code size is remarkably similar, and the execution performance is just as good. Not only that, but there are fewer lines of code. 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-passing, type casting, and other operations that are typically considered "unsafe" or "advaned topics" in C.

Chapter 3

Can you Afford an RTOS?

Of course, since you're reading the manual for an RTOS that I've been developing for the last few years, you can guess that the conclusion that I draw is a resounding "yes".

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.

- Sophisticated synchronization objects
- · The ability to efficiently block and wait
- · Enhanced responsiveness for high-priority tasks
- · Built in timers
- · Built in efficient memory management

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.

3.1 Intro

(Note - this article was written for the C-based Mark2 kernel, which is slightly different. While the general principles are the same, the numbers are not an 100% accurate reflection of the current costs of the Mark3 kernel.)

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. 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 I/O using various user-input devices and a serial graphics display. As a result, a number of Mark3 device drivers are also implemented.

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 WinAVR GCC compiler with -O2 level optimizations, an executable is produced with the following code/RAM utilization:

31600 Bytes Code Space 2014 Bytes RAM

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:

29904 Bytes Code Space 1648 Bytes RAM

3.2 Memory overhead:

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.

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) 64 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 declare the threads in preemptive mode.

With this taken into account, the true memory cost of a 2-thread system ends up being around 150 bytes of RA-M - which is less than 8% 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.

3.3 Code Space Overhead:

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:

3466 Bytes

The configuration tested in this comparison uses the thread/port module with timers, drivers, and semaphores, for a total kernel size of \sim 3.5KB, with the rest of the code space occupied by the application.

The custom cooperative-mode framework has a similar structure which is broken down by module as follows:

1850 Bytes

As can be seen from the compiler's output, the difference in code space between the two versions of the application is about 1.7kB - or about 5% 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.

3.4 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 high-priority thread's event flag
- · Acknowledging the event from the event loop

Using the cycle-accurate AVR simulator, we find the end-to-end event sequence time to be 20.4us for the cooperative mode scheduler and 44.2us for the preemptive, giving a difference of 23.8us.

With a fixed high-priority event frequency of 33Hz, we achieve a runtime overhead of 983.4us per second, or 0.0983% 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 neglible for a preemptive system. 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 10% of a microcontroller's code space/RAM for a preemptive kernel's overhead, you should only require at minimum a microcontroller with 16k 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 4

Superloops

4.1 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 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 liimted-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.

4.2 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.

While the execution timeline for this program is equally boring, for the sake of completeness it would look like this:

Despite its simplicity we can see the beginnings of some core OS concepts. Here, the while(1) statement can be logically seen as the he 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

10 Superloops

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.

4.3 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 miliseconds - 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-aplication 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.

4.4 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();
}
```

12 Superloops

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 Task 3. 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 ascooperative 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.

4.5 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)
                         (0x02)
#define HP_TASK_2
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
  _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();
```

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.

4.6 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.

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.

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.

No blocking calls

In a super-loop environment, there's no such thing as a blocking call or blocking objects. Tasks cannot stop midexecution 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.

Difficult to guarantee responsiveness

14 Superloops

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.

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 unwiedly for two levels of priority, adding more levels beyond this is 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 5

Mark3 Overview

5.1 Intro

The following section details the overall design of Mark3, the goals I've set out to achieve, the features that I've intended to provide, as well as an introduction to the programming concepts used to make it happen.

5.2 Features

Mark3 is a fully-featured real-time kernel, and is feature-competitive with other open-source and commercial RTOS's in the embedded arena.

The key features of this RTOS are:

- Flexible Scheduler
 - Unlimited number of threads with 8 priority levels
 - Unlimited threads per priority level
 - Round-robin scheduling for threads at each priority level
 - Time quantum scheduling for each thread in a given priority level
- · Configurable stacks for each Thread
- · Resource protection:
 - Integrated mutual-exclusion semaphores (Mutex)
 - Priority-inheritance on Mutex objects to prevent priority inversion
- · Synchronization Objects
 - Binary and counting Semaphore to coordinate thread execution
 - Event flags with 16-bit bitfields for complex thread synchronization
- · Efficient Timers
 - The RTOS is tickless, the OS only wakes up when a timer expires, not at a regular interval
 - One-shot and periodic timers with event callbacks
 - Timers are high-precision and long-counting (about 68000 seconds when used with a 16us resolution timer)
- Driver API
 - A hardware abstraction layer is provided to simplify driver development

16 Mark3 Overview

- · Robust Interprocess Communications
 - Threadsafe global Message pool and configurable message queues
- · Support for kernel-aware simulation
 - Provides advanced test and verification functionality, allowing for easy integration into continuousintegration systems
 - Provide accurate engineering data on key metrics like stack usage and realtime performance, with easyto-use APIs and little overhead

5.3 Design Goals

Lightweight

Mark3 can be configured to have an extremely low static memory footprint. Each thread is defined with its own stack, and each thread structure can be configured to take as little as 26 bytes of RAM. The complete Mark3 kernel with all features, setup code, a serial driver, and the Mark3 protocol libraries comes in at under 9K of code space and 1K of RAM on atmel AVR.

Modular

Each system feature can be enabled or disabled by modifying the kernel configuration header file. Include what you want, and ignore the rest to save code space and RAM.

Easily Portable

Mark3 should be portable to a variety of 8, 16 and 32 bit architectures without MMUs. Porting the OS to a new architecture is relatively straightforward, requiring only device-specific implementations for the lowest-level operations such as context switching and timer setup.

Easy To Use

Mark3 is small by design - which gives it the advantage that it's also easy to develop for. This manual, the code itself, and the Doxygen documentation in the code provide ample documentation to get you up to speed quickly. Because you get to see the source, there's nothing left to assumption.

Simple to Understand

Not only is the Mark3 API rigorously documented (hey - that's what this book is for!), but the architecture and naming conventions are intuitive - it's easy to figure out where code lives, and how it works. Individual modules are small due to the "one feature per file" rule used in development. This makes Mark3 an ideal platform for learning about aspects of RTOS design.

Chapter 6

Getting Started

6.1 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 K_UCHAR aucAppStack[STACK_SIZE_APP];
static K_UCHAR 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.

18 Getting Started

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

6.2 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.

6.2.1 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 K_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 aplies 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.

6.3 Timers 19

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 initallization is shown below:

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.

6.2.2 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:

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.

6.3 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:

20 Getting Started

6.4 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();
    }
}
```

6.5 Mutexes 21

6.5 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 inheritence 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 artificically 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();
    \ensuremath{//} Once the mutex is owned, no other thread can
    \ensuremath{//} 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();
```

6.6 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.

22 Getting Started

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)
        K_USHORT usWakeCondition;
        // Allow this thread to block on multiple flags
        usWakeCondition = clEventFlag.Wait(0x00FF, EVENT_FLAG_ANY);
        // Clear the event condition that caused the thread to wake (in this case,
        // usWakeCondtion will equal 0x20 when triggered from the interrupt above)
        clEventFlag.Clear(usWakeCondition);
        // <do something>
```

6.7 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.

6.7.1 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.

6.7 Messages 23

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.

6.7.2 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.

6.7.3 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 Message-Queue 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.

6.7.4 Messaging Example

```
// Message queue object shared between threads
MessageQueue clMsgQ;
// Function that initializes the shared message queue
void MsqQInit()
    clMsgQ.Init();
// Function called by one thread to send message data to
// another
void TxMessage()
    // Get a message, initialize its data
   Message *pclMesg = GlobalMessagePool::Pop();
    pclMesg->SetCode(0xAB);
   pclMesg->SetData((void*)some_data);
    // Send the data to the message queue
    clMsgQ.Send(pclMesg);
// Function called in the other thread to block until
// a message is received in the message queue.
void RxMessage()
    Message *pclMesg;
```

24 Getting Started

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

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

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

6.8 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;
}
```

6.9 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.

Build System

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.

7.1 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-configuration data for various platforms
              Documentation (including this)
   docs
   drivers
               Device driver code for various supported devices
              Example applications
   example
              Bitmap fonts converted from TTF, used by Mark3 graphics library
   fonts
   kernel
                Basic Mark3 Components (the focus of this manual)
       cpu
               CPU-specific porting code
               Scripts used to simplify build, documentation, and profiling
   scripts
                Utility code and services, extended system features
   services
   stage
                Staging directory, where the build system places artifacts
              Unit tests, written as C/C++ applications
   util
               .net-based utils: font conversion, terminal, programmer, and configuration
```

7.2 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.

26 Build System

```
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:

Once you've placed your code files in the right place, and configured the makefile appropriately, a fresh call to make headers, make library, then make binary will guarantee that your code is built.

Now, you can still copy-and-paste the required kernel, port, and drivers, directly into your application avoiding the whole process of using make from the command line. To do this, run "make source" from the root directory in svn, and copy the contents of /stage/src into your project. This should contain the source to the kernel, all drivers, and all services that are in the tree - along with the necessary header files.

7.3 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 toolcahin 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 7 x64, it should look something like this:

C: Files (x86) Toolchain GCC\Native\3.4.2.1002-gnu-toolchain

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 synatx 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.

28 **Build System**

License

8.1 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 INTERRUPTION) 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.

30 License

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.

9.1 Date Performed

Wed May 27 21:18:21 EDT 2015

9.2 Compiler Information

The kernel and test code used in these results were built using the following compiler: ./scripts/profile.sh: 55: ./scripts/profile.sh: /home/moslevin/atmel/bin/avr-gcc: not found

9.3 Profiling Results

- Semaphore Initialization: cycles (averaged over 0 iterations)
- Semaphore Post (uncontested): cycles (averaged over 0 iterations)
- Semaphore Pend (uncontested): cycles (averaged over 0 iterations)
- Semaphore Flyback Time (Contested Pend): cycles (averaged over 0 iterations)
- Mutex Init: cycles (averaged over 0 iterations)
- Mutex Claim: cycles (averaged over 0 iterations)
- Mutex Release: cycles (averaged over 0 iterations)
- Thread Initialize: cycles (averaged over 0 iterations)
- Thread Start: cycles (averaged over 0 iterations)
- Context Switch: cycles (averaged over 0 iterations)
- Thread Schedule: cycles (averaged over 0 iterations)

32 **Profiling Results**

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.

10.1 Information

Subversion Repository Information:

- · Repository Root: svn+ssh://m0slevin.code.sf.net/p/mark3/source
- · Revision: 211
- URL: svn+ssh://m0slevin.code.sf.net/p/mark3/source/branch/release/R1/embedded Relative URL: ^/branch/release/R1/embedded

Date Profiled: Mon May 18 22:24:26 EDT 2015

10.2 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.

10.3 Profiling Results

Mark3 Module Size Report:

- Synchronization Objects Base Class.....: 92 Bytes
- Device Driver Framework (including /dev/null)...: 226 Bytes
- Synchronization Object Event Flag.....: : 770 Bytes
- Fundamental Kernel Linked-List Classes......: : 496 Bytes

34 Code Size Profiling

Message-based IPC.....: : 426 Bytes

• Mutex (Synchronization Object).....: : 658 Bytes

• Performance-profiling timers.....: 546 Bytes

Round-Robin Scheduling Support.....: 264 Bytes

• Thread Scheduling.....: : 452 Bytes

• Semaphore (Synchronization Object).....: : 536 Bytes

• Thread Implementation.....: 1613 Bytes

• Fundamental Kernel Thread-list Data Structures.. : 210 Bytes

Mark3 Kernel Base Class.....: 110 Bytes

Software Timer Kernel Object.....: 378 Bytes

Software Timer Management.....:: 645 Bytes

• Runtime Kernel Trace Implementation.....: 0 Bytes

· Circular Logging Buffer Base Class.....: 0 Bytes

• Atmel AVR - Kernel Aware Simulation Support.....: 296 Bytes

• Atmel AVR - Basic Threading Support.....: : 598 Bytes

• Atmel AVR - Kernel Interrupt Implemenation...... : 56 Bytes

Atmel AVR - Kernel Timer Implementation......: 322 Bytes

· kernelprofile.cpp.o: 256 Bytes

Mark3 Kernel Size Summary:

· Kernel: 2973 Bytes

· Synchronization Objects: 2390 Bytes

Port: 1272 Bytes

· Features: 2059 Bytes

· Total Size: 8694 Bytes

Hierarchical Index

11.1 Class Hierarchy

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

BlockingObject	43
EventFlag	53
Mutex	74
Semaphore	84
DriverList	52
FakeThread_t	56
GlobalMessagePool	57
Kernel	58
KernelAware	60
KernelAwareData_t	64
KernelSWI	64
KernelTimer	65
LinkList	67
CircularLinkList	44
ThreadList	96
DoubleLinkList	48
TimerList	03
LinkListNode	69
Driver	49
DevNull	
Message	70
Thread	88
Timer	99
MessageQueue	72
Profiler	76
ProfileTimer	77
Quantum	79
Scheduler	81
ThreadPort	98
TimerScheduler 1	05

36 **Hierarchical Index**

Class Index

12.1 Class List

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

BlockingObject	
Class implementing thread-blocking primatives	43
CircularLinkList	
Circular-linked-list data type, inherited from the base LinkList type	44
DevNull	
This class implements the "default" driver (/dev/null)	45
DoubleLinkList	
Doubly-linked-list data type, inherited from the base LinkList type	48
Driver	
Base device-driver class used in hardware abstraction	49
DriverList	
List of Driver objects used to keep track of all device drivers in the system	52
EventFlag	
Blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread exe-	
cution based on events occurring within the system	53
FakeThread_t	
If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data	E/
structure to "simulate" the effect of having an idle thread in the system	56
GlobalMessagePool	57
Implements a list of message objects shared between all threads	5/
Class that encapsulates all of the kernel startup functions	58
KernelAware	50
The KernelAware class	60
KernelAwareData t	
This structure is used to communicate between the kernel and a kernel- aware host	64
KernelSWI	
Class providing the software-interrupt required for context-switching in the kernel	64
KernelTimer	
Hardware timer interface, used by all scheduling/timer subsystems	65
LinkList	
Abstract-data-type from which all other linked-lists are derived	67
LinkListNode	
Basic linked-list node data structure	69
Message	
Class to provide message-based IPC services in the kernel	70
MessageQueue	
List of messages, used as the channel for sending and receiving messages between threads .	72

38 Class Index

Mutex		
	Mutual-exclusion locks, based on BlockingObject	74
Profiler		
	System profiling timer interface	76
ProfileTi	mer	
	Profiling timer	77
Quantun	n	
	Static-class used to implement Thread quantum functionality, which is a key part of round-robin	
	scheduling	79
Schedul	er	
	Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping	81
Semaph	ore	
	Counting semaphore, based on BlockingObject base class	84
Thread		
	Object providing fundamental multitasking support in the kernel	88
ThreadL	ist	
	This class is used for building thread-management facilities, such as schedulers, and blocking	
	objects	96
ThreadP	Port	
	Class defining the architecture specific functions required by the kernel	98
Timer		
	Timer - an event-driven execution context based on a specified time interval	99
TimerLis	st ender the second of the	
	TimerList class - a doubly-linked-list of timer objects	103
TimerSc	heduler	
	"Static" Class used to interface a global TimerList with the rest of the kernel	105

File Index

13.1 File List

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

/home/moslevin/Project/R1/kernel/atomic.cpp
Basic Atomic Operations
/home/moslevin/Project/R1/kernel/blocking.cpp
Implementation of base class for blocking objects
/home/moslevin/Project/R1/kernel/driver.cpp
Device driver/hardware abstraction layer
/home/moslevin/Project/R1/kernel/eventflag.cpp
Event Flag Blocking Object/IPC-Object implementation
/home/moslevin/Project/R1/kernel/kernel.cpp
Kernel initialization and startup code
/home/moslevin/Project/R1/kernel/kernelaware.cpp
Kernel aware simulation support
/home/moslevin/Project/R1/kernel/ksemaphore.cpp
Semaphore Blocking-Object Implemenation
/home/moslevin/Project/R1/kernel/II.cpp
Core Linked-List implementation, from which all kernel objects are derived
/home/moslevin/Project/R1/kernel/message.cpp
Inter-thread communications via message passing
/home/moslevin/Project/R1/kernel/mutex.cpp
Mutual-exclusion object
/home/moslevin/Project/R1/kernel/profile.cpp
Code profiling utilities
/home/moslevin/Project/R1/kernel/quantum.cpp
Thread Quantum Implementation for Round-Robin Scheduling
/home/moslevin/Project/R1/kernel/scheduler.cpp
Strict-Priority + Round-Robin thread scheduler implementation
/home/moslevin/Project/R1/kernel/thread.cpp
Platform-Independent thread class Definition
/home/moslevin/Project/R1/kernel/threadlist.cpp
Thread linked-list definitions
/home/moslevin/Project/R1/kernel/timer.cpp
Timer implementations
/home/moslevin/Project/R1/kernel/timerlist.cpp
Implements timer list processing algorithms, responsible for all timer tick and expiry logic 208
/home/moslevin/Project/R1/kernel/tracebuffer.cpp
Kernel trace buffer class definition
/home/moslevin/Project/R1/kernel/writebuf16.cpp
16 bit circular buffer implementation with callbacks

40 File Index

/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp	
ATMega328p Profiling timer implementation	110
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp	
Kernel Software interrupt implementation for ATMega328p	112
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp	
Kernel Timer Implementation for ATMega328p	113
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/threadport.cpp	
ATMega328p Multithreading	122
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h	
Profiling timer hardware interface	115
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h	
Kernel Software interrupt declarations	116
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h	
Kernel Timer Class declaration	117
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/threadport.h	
ATMega328p Multithreading support	119
/home/moslevin/Project/R1/kernel/public/atomic.h	
Basic Atomic Operations	149
/home/moslevin/Project/R1/kernel/public/blocking.h	
Blocking object base class declarations	151
/home/moslevin/Project/R1/kernel/public/debugtokens.h	
Hex codes/translation tables used for efficient string tokenization	152
/home/moslevin/Project/R1/kernel/public/driver.h	
Driver abstraction framework	154
/home/moslevin/Project/R1/kernel/public/eventflag.h	
Event Flag Blocking Object/IPC-Object definition	156
/home/moslevin/Project/R1/kernel/public/kernel.h	100
Kernel initialization and startup class	157
/home/moslevin/Project/R1/kernel/public/kernelaware.h	137
Kernel aware simulation support	159
/home/moslevin/Project/R1/kernel/public/kerneldebug.h	159
Macros and functions used for assertions, kernel traces, etc	160
/home/moslevin/Project/R1/kernel/public/kerneltypes.h	100
Basic data type primatives used throughout the OS	163
/home/moslevin/Project/R1/kernel/public/ksemaphore.h	103
Semaphore Blocking Object class declarations	164
/home/moslevin/Project/R1/kernel/public/ll.h	104
· · · · · · · · · · · · · · · · · · ·	166
Core linked-list declarations, used by all kernel list types	100
/home/moslevin/Project/R1/kernel/public/manual.h	107
Ascii-format documentation, used by doxygen to create various printable and viewable forms .	167
/home/moslevin/Project/R1/kernel/public/mark3.h	100
Single include file given to users of the Mark3 Kernel API	168
/home/moslevin/Project/R1/kernel/public/mark3cfg.h	470
Mark3 Kernel Configuration	1/2
/home/moslevin/Project/R1/kernel/public/message.h	
Inter-thread communication via message-passing	175
/home/moslevin/Project/R1/kernel/public/mutex.h	
Mutual exclusion class declaration	177
/home/moslevin/Project/R1/kernel/public/paniccodes.h	
Defines the reason codes thrown when a kernel panic occurs	178
/home/moslevin/Project/R1/kernel/public/profile.h	
High-precision profiling timers	
/home/moslevin/Project/R1/kernel/public/ profiling_results.h	??
/home/moslevin/Project/R1/kernel/public/quantum.h	
Thread Quantum declarations for Round-Robin Scheduling	180
/home/moslevin/Project/R1/kernel/public/scheduler.h	
Thread scheduler function declarations	
/home/moslevin/Project/R1/kernel/public/ sizeprofile.h	??

13.1 File List 41

/home/moslevin/Project/R1/kernel/public/thread.h
Platform independent thread class declarations
/home/moslevin/Project/R1/kernel/public/threadlist.h
Thread linked-list declarations
/home/moslevin/Project/R1/kernel/public/timer.h
Timer object declarations
/home/moslevin/Project/R1/kernel/public/timerlist.h
Timer list declarations
/home/moslevin/Project/R1/kernel/public/timerscheduler.h
Timer scheduler declarations
/home/moslevin/Project/R1/kernel/public/tracebuffer.h
Kernel trace buffer class declaration
/home/moslevin/Project/R1/kernel/public/writebuf16.h
Thread-safe circular buffer implementation with 16-bit elements

42 File Index

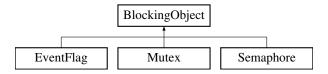
Class Documentation

14.1 BlockingObject Class Reference

Class implementing thread-blocking primatives.

#include <blocking.h>

Inheritance diagram for BlockingObject:



Protected Member Functions

- void Block (Thread *pclThread)
- void UnBlock (Thread *pclThread_)

Protected Attributes

ThreadList m_clBlockList

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

14.1.1 Detailed Description

Class implementing thread-blocking primatives.

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.

14.1.2 Member Function Documentation

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

44 Class Documentation

Parameters

pclThread_ Pointer to the thread object that will be blocked.

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.

Definition at line 36 of file blocking.cpp.

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

Parameters

pclThread_ Pointer to the thread to unblock.

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

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

Definition at line 52 of file blocking.cpp.

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

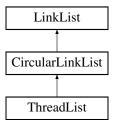
- /home/moslevin/Project/R1/kernel/public/blocking.h
- /home/moslevin/Project/R1/kernel/blocking.cpp

14.2 CircularLinkList Class Reference

Circular-linked-list data type, inherited from the base LinkList type.

#include <11.h>

Inheritance diagram for CircularLinkList:



Public Member Functions

virtual void Add (LinkListNode *node_)

Add the linked list node to this linked list.

virtual void Remove (LinkListNode *node)

Add the linked list node to this linked list.

void PivotForward ()

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

void PivotBackward ()

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

Additional Inherited Members

14.2.1 Detailed Description

Circular-linked-list data type, inherited from the base LinkList type.

Definition at line 196 of file II.h.

14.2.2 Member Function Documentation

14.2.2.1 void CircularLinkList::Add (LinkListNode * node_) [virtual]

Add the linked list node to this linked list.

Parameters

node_ Pointer to the node to add

Implements LinkList.

Reimplemented in ThreadList.

Definition at line 102 of file II.cpp.

14.2.2.2 void CircularLinkList::Remove (LinkListNode * node_) [virtual]

Add the linked list node to this linked list.

Parameters

node_ Pointer to the node to remove

Implements LinkList.

Reimplemented in ThreadList.

Definition at line 127 of file II.cpp.

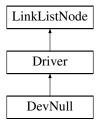
The documentation for this class was generated from the following files:

- · /home/moslevin/Project/R1/kernel/public/II.h
- /home/moslevin/Project/R1/kernel/II.cpp

14.3 DevNull Class Reference

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

Inheritance diagram for DevNull:



46 Class Documentation

Public Member Functions

· virtual void Init ()

Initialize a driver, must be called prior to use.

virtual K_UCHAR Open ()

Open a device driver prior to use.

virtual K_UCHAR Close ()

Close a previously-opened device driver.

virtual K_USHORT Read (K_USHORT usBytes_, K_UCHAR *pucData_)

Read a specified number of bytes from the device into a specific buffer.

virtual K_USHORT Write (K_USHORT usBytes_, K_UCHAR *pucData_)

Write a payload of data of a given length to the device.

 virtual K_USHORT Control (K_USHORT usEvent_, void *pvDataln_, K_USHORT usSizeIn_, void *pvData-Out_, K_USHORT usSizeOut_)

This is the main entry-point for device-specific io and control operations.

Additional Inherited Members

14.3.1 Detailed Description

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

Definition at line 40 of file driver.cpp.

14.3.2 Member Function Documentation

```
14.3.2.1 virtual K_UCHAR DevNull::Close() [inline], [virtual]
```

Close a previously-opened device driver.

Returns

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

Implements Driver.

Definition at line 45 of file driver.cpp.

```
14.3.2.2 virtual K_USHORT DevNull::Control ( K_USHORT usEvent_, void * pvDataln_, K_USHORT usSizeln_, void * pvDataOut_, K_USHORT usSizeOut_) [inline], [virtual]
```

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 function is analogous to the non-POSIX (yet still common) devctl() or ioctl().

Parameters

usEvent_	Code defining the io event (driver-specific)
pvDataIn_	Pointer to the intput data
usSizeIn_	Size of the input data (in bytes)

pvDataOut_	Pointer to the output data
usSizeOut_	Size of the output data (in bytes)

Returns

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

Implements Driver.

Definition at line 53 of file driver.cpp.

```
14.3.2.3 virtual K UCHAR DevNull::Open ( ) [inline], [virtual]
```

Open a device driver prior to use.

Returns

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

Implements Driver.

Definition at line 44 of file driver.cpp.

```
14.3.2.4 virtual K_USHORT DevNull::Read ( K_USHORT usBytes_, K_UCHAR * pucData_ ) [inline], [virtual]
```

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

usBytes_	Number of bytes to read (<= size of the buffer)
pucData_	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implements Driver.

Definition at line 47 of file driver.cpp.

```
14.3.2.5 virtual K_USHORT DevNull::Write ( K_USHORT usBytes_, K_UCHAR * pucData_ ) [inline], [virtual]
```

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

48 Class Documentation

usBytes_	Number of bytes to write (<= size of the buffer)
pucData_	Pointer to a data buffer containing the data to write

Returns

Number of bytes actually written

Implements Driver.

Definition at line 50 of file driver.cpp.

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

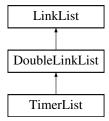
/home/moslevin/Project/R1/kernel/driver.cpp

14.4 DoubleLinkList Class Reference

Doubly-linked-list data type, inherited from the base LinkList type.

#include <ll.h>

Inheritance diagram for DoubleLinkList:



Public Member Functions

• DoubleLinkList ()

Default constructor - initializes the head/tail nodes to NULL.

virtual void Add (LinkListNode *node_)

Add the linked list node to this linked list.

virtual void Remove (LinkListNode *node)

Add the linked list node to this linked list.

Additional Inherited Members

14.4.1 Detailed Description

Doubly-linked-list data type, inherited from the base LinkList type.

Definition at line 165 of file II.h.

14.4.2 Member Function Documentation

14.4.2.1 void DoubleLinkList::Add (LinkListNode * node_) [virtual]

Add the linked list node to this linked list.

Parameters

node_ Pointer to the node to add

Implements LinkList.

Definition at line 41 of file II.cpp.

14.4.2.2 void DoubleLinkList::Remove (LinkListNode * node_) [virtual]

Add the linked list node to this linked list.

Parameters

node_ Pointer to the node to remove

Implements LinkList.

Definition at line 65 of file II.cpp.

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

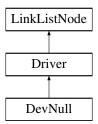
- /home/moslevin/Project/R1/kernel/public/II.h
- /home/moslevin/Project/R1/kernel/II.cpp

14.5 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

Initialize a driver, must be called prior to use.

virtual K_UCHAR Open ()=0

Open a device driver prior to use.

• virtual K_UCHAR Close ()=0

Close a previously-opened device driver.

• virtual K_USHORT Read (K_USHORT usBytes_, K_UCHAR *pucData_)=0

Read a specified number of bytes from the device into a specific buffer.

virtual K_USHORT Write (K_USHORT usBytes_, K_UCHAR *pucData_)=0

Write a payload of data of a given length to the device.

 virtual K_USHORT Control (K_USHORT usEvent_, void *pvDataln_, K_USHORT usSizeIn_, void *pvData-Out_, K_USHORT usSizeOut_)=0

This is the main entry-point for device-specific io and control operations.

50 Class Documentation

void SetName (const K_CHAR *pcName_)
 Set the path for the driver.

const K_CHAR * GetPath ()

Returns a string containing the device path.

Private Attributes

 const K_CHAR * m_pcPath
 string pointer that holds the driver path (name)

Additional Inherited Members

14.5.1 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.

14.5.2 Member Function Documentation

```
14.5.2.1 K_UCHAR Driver::Close() [pure virtual]
```

Close a previously-opened device driver.

Returns

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

Implemented in DevNull.

```
14.5.2.2 K_USHORT Driver::Control ( K_USHORT usEvent_, void * pvDataln_, K_USHORT usSizeln_, void * pvDataOut_, K_USHORT usSizeOut_) [pure virtual]
```

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 function is analogous to the non-POSIX (yet still common) devctl() or ioctl().

Parameters

usEvent_	Code defining the io event (driver-specific)
pvDataIn_	Pointer to the intput data
usSizeIn_	Size of the input data (in bytes)
pvDataOut_	Pointer to the output data
usSizeOut_	Size of the output data (in bytes)

Returns

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

Implemented in DevNull.

14.5 Driver Class Reference 51

```
14.5.2.3 const K_CHAR * Driver::GetPath() [inline]
```

Returns a string containing the device path.

Returns

pcName_ Return the string constant representing the device path

Definition at line 231 of file driver.h.

```
14.5.2.4 K_UCHAR Driver::Open() [pure virtual]
```

Open a device driver prior to use.

Returns

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

Implemented in DevNull.

```
14.5.2.5 K_USHORT Driver::Read ( K_USHORT usBytes_, K_UCHAR * pucData_ ) [pure virtual]
```

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

usBytes_	Number of bytes to read (<= size of the buffer)
pucData_	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implemented in DevNull.

```
14.5.2.6 void Driver::SetName ( const K_CHAR * pcName_ ) [inline]
```

Set the path for the driver.

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

Parameters

pcName_	String constant containing the device path

Definition at line 222 of file driver.h.

```
14.5.2.7 K_USHORT Driver::Write (K_USHORT usBytes_, K_UCHAR * pucData_) [pure virtual]
```

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.

52 Class Documentation

Parameters

usBytes_	Number of bytes to write (<= size of the buffer)
pucData_	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/moslevin/Project/R1/kernel/public/driver.h

14.6 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 ()

Initialize the list of drivers.

• static void Add (Driver *pclDriver_)

Add a Driver object to the managed global driver-list.

• static void Remove (Driver *pclDriver_)

Remove a driver from the global driver list.

• static Driver * FindByPath (const K_CHAR *m_pcPath)

Look-up a driver in the global driver-list based on its path.

Static Private Attributes

• static DoubleLinkList m_clDriverList

LinkedList object used to implementing the driver object management.

14.6.1 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 244 of file driver.h.

14.6.2 Member Function Documentation

```
14.6.2.1 DriverList::Add ( Driver * pclDriver_ ) [inline], [static]
```

Add a Driver object to the managed global driver-list.

Parameters

pclDriver_ pointer to the driver object to add to the global driver list.

Definition at line 264 of file driver.h.

```
14.6.2.2 Driver * DriverList::FindByPath ( const K_CHAR * m_pcPath ) [static]
```

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 107 of file driver.cpp.

```
14.6.2.3 void DriverList::Init( ) [static]
```

Initialize the list of drivers.

Must be called prior to using the device driver library.

Definition at line 98 of file driver.cpp.

```
14.6.2.4 void DriverList::Remove ( Driver * pclDriver_ ) [inline], [static]
```

Remove a driver from the global driver list.

Parameters

```
pclDriver_ Pointer to the driver object to remove from the global table
```

Definition at line 274 of file driver.h.

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

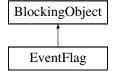
- /home/moslevin/Project/R1/kernel/public/driver.h
- /home/moslevin/Project/R1/kernel/driver.cpp

14.7 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.

K_USHORT Wait (K_USHORT usMask_, EventFlagOperation_t eMode_)

54 Class Documentation

Wait - Block a thread on the specific flags in this event flag group.

• K_USHORT Wait (K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG ulTimeMS_)

Wait - Block a thread on the specific flags in this event flag group.

void WakeMe (Thread *pclOwner_)

WakeMe.

· void Set (K USHORT usMask)

Set - Set additional flags in this object (logical OR).

• void Clear (K USHORT usMask)

ClearFlags - Clear a specific set of flags within this object, specific by bitmask.

K_USHORT GetMask ()

GetMask Returns the state of the 16-bit bitmask within this object.

Private Member Functions

K_USHORT Wait_i (K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG ulTimeMS_)
 Wait_i.

Private Attributes

· K USHORT m usSetMask

Event flags currently set in this object.

Additional Inherited Members

14.7.1 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.

Definition at line 46 of file eventflag.h.

14.7.2 Member Function Documentation

14.7.2.1 void EventFlag::Clear (K_USHORT usMask_)

ClearFlags - Clear a specific set of flags within this object, specific by bitmask.

Parameters

usMask_ - Bitmask of flags to clear

Definition at line 295 of file eventflag.cpp.

14.7.2.2 K_USHORT 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 304 of file eventflag.cpp.

```
14.7.2.3 void EventFlag::Set ( K_USHORT usMask_ )
```

Set - Set additional flags in this object (logical OR).

This API can potentially result in threads blocked on Wait() to be unblocked.

Parameters

```
usMask_ - Bitmask of flags to set.
```

Definition at line 176 of file eventflag.cpp.

14.7.2.4 K_USHORT EventFlag::Wait (K_USHORT usMask_, EventFlagOperation_t eMode_)

Wait - Block a thread on the specific flags in this event flag group.

Parameters

usMask_	- 16-bit bitmask to block on
eMode_	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask
	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

Definition at line 158 of file eventflag.cpp.

Wait - Block a thread on the specific flags in this event flag group.

Parameters

usMask_	- 16-bit bitmask to block on
eMode_	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask
	EVENT_FLAG_ALL: Thread will block on all of the bits in the mask
ulTimeMS_	- Time to block (in ms)

Returns

Bitmask condition that caused the thread to unblock, or 0 on error or timeout

Definition at line 169 of file eventflag.cpp.

Wait_i.

Interal abstraction used to manage both timed and untimed wait operations

56 Class Documentation

Parameters

usMask_	- 16-bit bitmask to block on
eMode_	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask
	EVENT_FLAG_ALL: Thread will block on all of the bits in the mask
ulTimeMS_	- 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 65 of file eventflag.cpp.

```
14.7.2.7 void EventFlag::WakeMe ( Thread * pclOwner_ )
```

WakeMe.

Wake the given thread, currently blocking on this object

Parameters

pclOwner_	Pointer to the owner thread to unblock.
-----------	---

Definition at line 57 of file eventflag.cpp.

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

- · /home/moslevin/Project/R1/kernel/public/eventflag.h
- /home/moslevin/Project/R1/kernel/eventflag.cpp

14.8 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.

· K UCHAR m ucThreadID

Thread ID.

• K_UCHAR m_ucPriority

Default priority of the thread.

K_UCHAR m_ucCurPriority

Current priority of the thread (priority inheritence)

• ThreadState_t m_eState

Enum indicating the thread's current state.

14.8.1 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 494 of file thread.h.

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

/home/moslevin/Project/R1/kernel/public/thread.h

14.9 GlobalMessagePool Class Reference

Implements a list of message objects shared between all threads.

```
#include <message.h>
```

Static Public Member Functions

• static void Init ()

Initialize the message queue prior to use.

static void Push (Message *pclMessage_)

Return a previously-claimed message object back to the global queue.

static Message * Pop ()

Pop a message from the global queue, returning it to the user to be populated before sending by a transmitter.

Static Private Attributes

• static Message m_aclMessagePool [GLOBAL_MESSAGE_POOL_SIZE]

Array of message objects that make up the message pool.

static DoubleLinkList m_clList

Linked list used to manage the Message objects.

14.9.1 Detailed Description

Implements a list of message objects shared between all threads.

Definition at line 157 of file message.h.

14.9.2 Member Function Documentation

```
14.9.2.1 Message * GlobalMessagePool::Pop( ) [static]
```

Pop a message from the global queue, returning it to the user to be populated before sending by a transmitter.

Returns

Pointer to a Message object

Definition at line 70 of file message.cpp.

```
14.9.2.2 void GlobalMessagePool::Push ( Message * pclMessage_ ) [static]
```

Return a previously-claimed message object back to the global queue.

Used once the message has been processed by a receiver.

Parameters

```
pclMessage_ Pointer to the Message object to return back to the global queue
```

Definition at line 58 of file message.cpp.

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

- /home/moslevin/Project/R1/kernel/public/message.h
- /home/moslevin/Project/R1/kernel/message.cpp

14.10 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 kernel; function never returns.

• static K_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 K BOOL IsPanic ()

IsPanic Returns whether or not the kernel is in a panic state.

static void Panic (K_USHORT usCause_)

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 K BOOL m blsStarted

true if kernel is running, false otherwise

static K_BOOL m_blsPanic

true if kernel is in panic state, false otherwise

· static panic func t m pfPanic

user-set panic function

static idle_func_t m_pfldle

user-set idle function

static FakeThread_t m_clldle

Idle thread object (note: not a real thread)

14.10.1 Detailed Description

Class that encapsulates all of the kernel startup functions.

Definition at line 48 of file kernel.h.

14.10.2 Member Function Documentation

```
14.10.2.1 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 124 of file kernel.h.

```
14.10.2.2 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.

Definition at line 52 of file kernel.cpp.

```
14.10.2.3 static K_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 95 of file kernel.h.

```
14.10.2.4 static K BOOL Kernel::IsStarted() [inline], [static]
```

IsStarted.

Returns

Whether or not the kernel has started - true = running, false = not started

Definition at line 80 of file kernel.h.

```
14.10.2.5 void Kernel::Panic ( K USHORT usCause_ ) [static]
```

Panic Cause the kernel to enter its panic state.

Parameters

usCause_ Reason for the kernel panic

Definition at line 95 of file kernel.cpp.

14.10.2.6 static void Kernel::SetIdleFunc(idle_func_t pfldle_) [inline], [static]

SetIdleFunc Set the function to be called when no active threads are available to be scheduled by the scheduler.

Parameters

```
pfldle_ Pointer to the idle function
```

Definition at line 109 of file kernel.h.

```
14.10.2.7 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

```
pfPanic_ Panic function pointer
```

Definition at line 89 of file kernel.h.

```
14.10.2.8 Kernel::Start(void) [static]
```

Start the kernel; function never returns.

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.

Definition at line 86 of file kernel.cpp.

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

- /home/moslevin/Project/R1/kernel/public/kernel.h
- /home/moslevin/Project/R1/kernel/kernel.cpp

14.11 KernelAware Class Reference

The KernelAware class.

```
#include <kernelaware.h>
```

Static Public Member Functions

static void ProfileInit (const K_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 K_CHAR *szStr_)

Print

static void Trace (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_)

Trace.

static void Trace (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_)

Trace.

 static void Trace (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_, K_USHORT usArg2_)

Trace

static K_BOOL IsSimulatorAware (void)

IsSimulatorAware.

Static Private Member Functions

static void Trace_i (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_, K_USHORT usArg2_, KernelAwareCommand_t eCmd_)
 Trace_i.

14.11.1 Detailed Description

The KernelAware class.

This class contains functions that are used to trigger kernel-aware functionality within a supported simulation environment (i.e. flAVR).

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.

14.11.2 Member Function Documentation

```
14.11.2.1 void KernelAware::ExitSimulator(void) [static]
```

ExitSimulator.

Instruct the kernel-aware simulator to terminate (destroying the virtual CPU).

Definition at line 104 of file kernelaware.cpp.

```
14.11.2.2 K_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 164 of file kernelaware.cpp.

```
14.11.2.3 void KernelAware::Print ( const K_CHAR * szStr_ ) [static]
```

Print.

Instruct the kernel-aware simulator to print a char string

Parameters

```
szStr
```

Definition at line 155 of file kernelaware.cpp.

```
14.11.2.4 void KernelAware::ProfileInit(const K_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

```
szStr_ String to use as a tag for the profiling session.
```

Definition at line 77 of file kernelaware.cpp.

```
14.11.2.5 void KernelAware::ProfileReport (void ) [static]
```

ProfileReport.

Instruct the kernel-aware simulator to print a report for its current profiling data.

Definition at line 98 of file kernelaware.cpp.

```
14.11.2.6 void KernelAware::ProfileStart(void) [static]
```

ProfileStart.

Instruct the kernel-aware simulator to begin counting cycles towards the current profiling counter.

Definition at line 86 of file kernelaware.cpp.

```
14.11.2.7 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 92 of file kernelaware.cpp.

```
14.11.2.8 void KernelAware::Trace ( K USHORT usFile_, K USHORT usLine_, K USHORT usCode_ ) [static]
```

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

usFile_	16-bit code representing the file
usLine_	16-bit code representing the line in the file
usCode_	16-bit data code, which indicates the line's format.

Definition at line 110 of file kernelaware.cpp.

14.11.2.9 void KernelAware::Trace (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

usFile_	16-bit code representing the file
usLine_	16-bit code representing the line in the file
usCode_	16-bit data code, which indicates the line's format
usArg1_	16-bit argument to the format string.

Definition at line 118 of file kernelaware.cpp.

14.11.2.10 void KernelAware::Trace (K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_, K_USHORT usArg2_) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

usFile_	16-bit code representing the file
usLine_	16-bit code representing the line in the file
usCode_	16-bit data code, which indicates the line's format
usArg1_	16-bit argument to the format string.
usArg2_	16-bit argument to the format string.

Definition at line 127 of file kernelaware.cpp.

14.11.2.11 void KernelAware::Trace_i(K_USHORT usFile_, K_USHORT usLine_, K_USHORT usCode_, K_USHORT usArg1_, K_USHORT usArg2_, KernelAwareCommand_t eCmd_) [static], [private]

Trace_i.

Private function by which the class's Trace() methods are reflected, which allows us to realize a modest code saving.

Parameters

usFile_	16-bit code representing the file
usLine_	16-bit code representing the line in the file
usCode_	16-bit data code, which indicates the line's format
usArg1_	16-bit argument to the format string.
usArg2_	16-bit argument to the format string.
eCmd_	Code indicating the number of arguments to emit.

Definition at line 137 of file kernelaware.cpp.

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

/home/moslevin/Project/R1/kernel/public/kernelaware.h

/home/moslevin/Project/R1/kernel/kernelaware.cpp

14.12 KernelAwareData t Union Reference

This structure is used to communicate between the kernel and a kernel- aware host.

Public Attributes

• volatile K USHORT ausBuffer [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 simluator

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.

14.12.1 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_ucKA-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 37 of file kernelaware.cpp.

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

/home/moslevin/Project/R1/kernel/kernelaware.cpp

14.13 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)

Configure the software interrupt - must be called before any other software interrupt functions are called.

static void Start (void)

Enable ("Start") the software interrupt functionality.

static void Stop (void)

Disable the software interrupt functionality.

• static void Clear (void)

Clear the software interrupt.

static void Trigger (void)

Call the software interrupt.

static K_UCHAR DI ()

Disable the SWI flag itself.

static void RI (K_BOOL bEnable_)

Restore the state of the SWI to the value specified.

14.13.1 Detailed Description

Class providing the software-interrupt required for context-switching in the kernel.

Definition at line 32 of file kernelswi.h.

14.13.2 Member Function Documentation

```
14.13.2.1 K_UCHAR KernelSWI::DI() [static]
```

Disable the SWI flag itself.

Returns

previous status of the SWI, prior to the DI call

Definition at line 50 of file kernelswi.cpp.

```
14.13.2.2 void KernelSWI::RI(K_BOOL bEnable_) [static]
```

Restore the state of the SWI to the value specified.

Parameters

```
bEnable_ true - enable the SWI, false - disable SWI
```

Definition at line 58 of file kernelswi.cpp.

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

- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h
- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp

14.14 KernelTimer Class Reference

Hardware timer interface, used by all scheduling/timer subsystems.

```
#include <kerneltimer.h>
```

Static Public Member Functions

• static void Config (void)

Initializes the kernel timer before use.

static void Start (void)

Starts the kernel time (must be configured first)

static void Stop (void)

Shut down the kernel timer, used when no timers are scheduled.

static K_UCHAR DI (void)

Disable the kernel timer's expiry interrupt.

static void RI (K_BOOL bEnable_)

Retstore the state of the kernel timer's expiry interrupt.

static void El (void)

Enable the kernel timer's expiry interrupt.

static K_ULONG SubtractExpiry (K_ULONG ulInterval_)

Subtract the specified number of ticks from the timer's expiry count register.

static K_ULONG TimeToExpiry (void)

Returns the number of ticks remaining before the next timer expiry.

static K_ULONG SetExpiry (K_ULONG ulInterval_)

Resets the kernel timer's expiry interval to the specified value.

static K_ULONG GetOvertime (void)

Return the number of ticks that have elapsed since the last expiry.

static void ClearExpiry (void)

Clear the hardware timer expiry register.

Static Private Member Functions

· static K USHORT Read (void)

Safely read the current value in the timer register.

14.14.1 Detailed Description

Hardware timer interface, used by all scheduling/timer subsystems.

Definition at line 33 of file kerneltimer.h.

14.14.2 Member Function Documentation

```
14.14.2.1 K_ULONG KernelTimer::GetOvertime(void) [static]
```

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.

```
14.14.2.2 K_USHORT KernelTimer::Read (void ) [static], [private]
```

Safely read the current value in the timer register.

Returns

Value held in the timer register

Definition at line 66 of file kerneltimer.cpp.

```
14.14.2.3 void KernelTimer::RI(K_BOOL bEnable_) [static]
```

Retstore the state of the kernel timer's expiry interrupt.

Parameters

bEnable_ 1 enable, 0 disable

Definition at line 169 of file kerneltimer.cpp.

14.14.2.4 K_ULONG KernelTimer::SetExpiry (K_ULONG ulInterval_) [static]

Resets the kernel timer's expiry interval to the specified value.

Parameters

ulInterval_ 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.

14.14.2.5 K_ULONG KernelTimer::SubtractExpiry (K_ULONG ulInterval_) [static]

Subtract the specified number of ticks from the timer's expiry count register.

Returns the new expiry value stored in the register.

Parameters

ulInterval_ 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.

14.14.2.6 K_ULONG KernelTimer::TimeToExpiry(void) [static]

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:

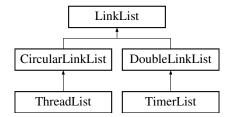
- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h
- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp

14.15 LinkList Class Reference

Abstract-data-type from which all other linked-lists are derived.

#include <ll.h>

Inheritance diagram for LinkList:



Public Member Functions

• void Init ()

Clear the linked list.

• virtual void Add (LinkListNode *node_)=0

Add the linked list node to this linked list.

virtual void Remove (LinkListNode *node)=0

Add the linked list node to this linked list.

LinkListNode * GetHead ()

Get the head node in the linked list.

LinkListNode * GetTail ()

Get the tail node of the linked list.

Protected Attributes

LinkListNode * m_pstHead

Pointer to the head node in the list.

LinkListNode * m_pstTail

Pointer to the tail node in the list.

14.15.1 Detailed Description

Abstract-data-type from which all other linked-lists are derived.

Definition at line 112 of file II.h.

14.15.2 Member Function Documentation

```
14.15.2.1 void LinkList::Add ( LinkListNode * node_ ) [pure virtual]
```

Add the linked list node to this linked list.

Parameters

```
node_ Pointer to the node to add
```

Implemented in CircularLinkList, DoubleLinkList, and ThreadList.

```
14.15.2.2 LinkListNode * LinkList::GetHead( ) [inline]
```

Get the head node in the linked list.

Returns

Pointer to the head node in the list

Definition at line 149 of file II.h.

14.15.2.3 LinkListNode * LinkList::GetTail() [inline]

Get the tail node of the linked list.

Returns

Pointer to the tail node in the list

Definition at line 158 of file II.h.

```
14.15.2.4 void LinkList::Remove ( LinkListNode * node_ ) [pure virtual]
```

Add the linked list node to this linked list.

Parameters

```
node_ Pointer to the node to remove
```

Implemented in CircularLinkList, DoubleLinkList, and ThreadList.

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

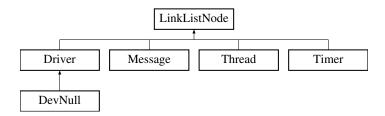
• /home/moslevin/Project/R1/kernel/public/II.h

14.16 LinkListNode Class Reference

Basic linked-list node data structure.

#include <11.h>

Inheritance diagram for LinkListNode:



Public Member Functions

LinkListNode * GetNext (void)

Returns a pointer to the next node in the list.

LinkListNode * GetPrev (void)

Returns a pointer to the previous node in the list.

Protected Member Functions

• void ClearNode ()

Initialize the linked list node, clearing its next and previous node.

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

14.16.1 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 II.h.

14.16.2 Member Function Documentation

```
14.16.2.1 LinkListNode * LinkListNode::GetNext(void) [inline]
```

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 II.h.

```
14.16.2.2 LinkListNode * LinkListNode::GetPrev(void) [inline]
```

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 II.h.

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

- · /home/moslevin/Project/R1/kernel/public/II.h
- /home/moslevin/Project/R1/kernel/II.cpp

14.17 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 ()

Initialize the data and code in the message.

void SetData (void *pvData)

Set the data pointer for the message before transmission.

void * GetData ()

Get the data pointer stored in the message upon receipt.

void SetCode (K_USHORT usCode_)

Set the code in the message before transmission.

K USHORT GetCode ()

Return the code set in the message upon receipt.

Private Attributes

void * m_pvData

Pointer to the message data.

· K USHORT m usCode

Message code, providing context for the message.

Additional Inherited Members

14.17.1 Detailed Description

Class to provide message-based IPC services in the kernel.

Definition at line 99 of file message.h.

14.17.2 Member Function Documentation

```
14.17.2.1 K_USHORT Message::GetCode( ) [inline]
```

Return the code set in the message upon receipt.

Returns

User code set in the object

Definition at line 143 of file message.h.

```
14.17.2.2 void * Message::GetData() [inline]
```

Get the data pointer stored in the message upon receipt.

Returns

Pointer to the data set in the message object

Definition at line 125 of file message.h.

```
14.17.2.3 Message::SetCode ( K_USHORT usCode_ ) [inline]
```

Set the code in the message before transmission.

Parameters

```
usCode_ Data code to set in the object
```

Definition at line 134 of file message.h.

```
14.17.2.4 void Message::SetData (void * pvData_) [inline]
```

Set the data pointer for the message before transmission.

Parameters

```
pvData_ Pointer to the data object to send in the message
```

Definition at line 116 of file message.h.

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

/home/moslevin/Project/R1/kernel/public/message.h

14.18 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 ()

Initialize the message queue prior to use.

• Message * Receive ()

Receive a message from the message queue.

Message * Receive (K_ULONG ulTimeWaitMS_)

Receive a message from the message queue.

void Send (Message *pclSrc_)

Send a message object into this message queue.

K_USHORT GetCount ()

Return the number of messages pending in the "receive" queue.

Private Member Functions

```
    Message * Receive_i (K_ULONG ulTimeWaitMS_)

Receive_i.
```

Private Attributes

• Semaphore m_clSemaphore

Counting semaphore used to manage thread blocking.

• DoubleLinkList m_clLinkList

List object used to store messages.

14.18.1 Detailed Description

List of messages, used as the channel for sending and receiving messages between threads.

Definition at line 201 of file message.h.

14.18.2 Member Function Documentation

```
14.18.2.1 K USHORT MessageQueue::GetCount ( )
```

Return the number of messages pending in the "receive" queue.

Returns

Count of pending messages in the queue.

Definition at line 156 of file message.cpp.

```
14.18.2.2 Message * MessageQueue::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

Definition at line 92 of file message.cpp.

```
14.18.2.3 Message * MessageQueue::Receive ( K_ULONG ulWaitTimeMS_ )
```

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

ulWaitTimeMS_	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 103 of file message.cpp.

```
14.18.2.4 Message * MessageQueue::Receive_i(K_ULONG ulTimeWaitMS_) [private]
```

Receive_i.

Internal function used to abstract timed and un-timed Receive calls.

Parameters

ulTimeWaitMS_	Time (in ms) to block, 0 for un-timed call.
---------------	---

Returns

Pointer to a message, or 0 on timeout.

Definition at line 111 of file message.cpp.

```
14.18.2.5 void MessageQueue::Send ( Message * pclSrc_ )
```

Send a message object into this message queue.

Will un-block the first waiting thread blocked on this queue if that occurs.

Parameters

```
pclSrc_ Pointer to the message object to add to the queue
```

Definition at line 140 of file message.cpp.

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

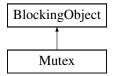
- · /home/moslevin/Project/R1/kernel/public/message.h
- /home/moslevin/Project/R1/kernel/message.cpp

14.19 Mutex Class Reference

Mutual-exclusion locks, based on BlockingObject.

```
#include <mutex.h>
```

Inheritance diagram for Mutex:



Public Member Functions

• void Init ()

Initialize a mutex object for use - must call this function before using the object.

• void Claim ()

Claim the mutex.

- K_BOOL Claim (K_ULONG ulWaitTimeMS_)
- void WakeMe (Thread *pclOwner)

Wake a thread blocked on the mutex.

• void Release ()

Release the mutex.

Private Member Functions

K_UCHAR WakeNext ()

Wake the next thread waiting on the Mutex.

K_BOOL Claim_i (K_ULONG ulWaitTimeMS_)
 Claim_i.

Private Attributes

• K_UCHAR m_ucRecurse

The recursive lock-count when a mutex is claimed multiple times by the same owner.

K_UCHAR m_bReady

State of the mutex - true = ready, false = claimed.

• K_UCHAR m_ucMaxPri

Maximum priority of thread in queue, used for priority inheritence.

Thread * m_pclOwner

Pointer to the thread that owns the mutex (when claimed)

Additional Inherited Members

14.19.1 Detailed Description

Mutual-exclusion locks, based on BlockingObject.

Definition at line 68 of file mutex.h.

14.19.2 Member Function Documentation

```
14.19.2.1 void Mutex::Claim (void)
```

Claim the mutex.

When the mutex is claimed, no other thread can claim a region protected by the object.

Definition at line 209 of file mutex.cpp.

```
14.19.2.2 K_BOOL Mutex::Claim ( K_ULONG ulWaitTimeMS_ )
```

Parameters

```
ulWaitTimeMS_
```

Returns

true - mutex was claimed within the time period specified false - mutex operation timed-out before the claim operation.

Definition at line 220 of file mutex.cpp.

```
14.19.2.3 K BOOL Mutex::Claim_i ( K ULONG ulWaitTimeMS_ ) [private]
```

Claim_i.

Abstracts out timed/non-timed mutex claim operations.

Parameters

ulWaitTimeMS Time in MS to wait, 0 for infinite

Returns

true on successful claim, false otherwise

Definition at line 107 of file mutex.cpp.

```
14.19.2.4 void Mutex::Release ( )
```

Release the mutex.

When the mutex is released, another object can enter the mutex-protected region.

Definition at line 227 of file mutex.cpp.

```
14.19.2.5 void Mutex::WakeMe ( Thread * pclOwner_ )
```

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 65 of file mutex.cpp.

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

- /home/moslevin/Project/R1/kernel/public/mutex.h
- /home/moslevin/Project/R1/kernel/mutex.cpp

14.20 Profiler Class Reference

System profiling timer interface.

```
#include <kernelprofile.h>
```

Static Public Member Functions

· static void Init ()

Initialize the global system profiler.

• static void Start ()

Start the global profiling timer service.

static void Stop ()

Stop the global profiling timer service.

static K_USHORT Read ()

Read the current tick count in the timer.

• static void Process ()

Process the profiling counters from ISR.

static K_ULONG GetEpoch ()

Return the current timer epoch.

14.20.1 Detailed Description

System profiling timer interface.

Definition at line 37 of file kernelprofile.h.

14.20.2 Member Function Documentation

```
14.20.2.1 void Profiler::Init (void ) [static]
```

Initialize the global system profiler.

Must be called prior to use.

Definition at line 32 of file kernelprofile.cpp.

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

- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h
- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp

14.21 ProfileTimer Class Reference

Profiling timer.

```
#include <profile.h>
```

Public Member Functions

• void Init ()

Initialize the profiling timer prior to use.

• void Start ()

Start a profiling session, if the timer is not already active.

void Stop ()

Stop the current profiling session, adding to the cumulative time for this timer, and the total iteration count.

• K ULONG GetAverage ()

Get the average time associated with this operation.

K_ULONG GetCurrent ()

Return the current tick count held by the profiler.

Private Member Functions

K_ULONG ComputeCurrentTicks (K_USHORT usCount_, K_ULONG ulEpoch_)

Figure out how many ticks have elapsed in this iteration.

Private Attributes

• K_ULONG m_ulCumulative

Cumulative tick-count for this timer.

K_ULONG m_ulCurrentIteration

Tick-count for the current iteration.

• K_USHORT m_usInitial

Initial count.

• K_ULONG m_ullnitialEpoch

Initial Epoch.

• K_USHORT m_usIterations

Number of iterations executed for this profiling timer.

K_UCHAR m_bActive

Wheter or not the timer is active or stopped.

14.21.1 Detailed Description

Profiling timer.

This class is used to perform high-performance profiling of code to see how K_LONG certain operations take. Useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

Definition at line 69 of file profile.h.

14.21.2 Member Function Documentation

```
14.21.2.1 K_ULONG ProfileTimer::ComputeCurrentTicks ( K_USHORT usCount_, K_ULONG ulEpoch_ )

[private]
```

Figure out how many ticks have elapsed in this iteration.

Parameters

usCount_	Current timer count
ulEpoch_	Current timer epoch

Returns

Current tick count

Definition at line 106 of file profile.cpp.

```
14.21.2.2 K ULONG ProfileTimer::GetAverage ( )
```

Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 79 of file profile.cpp.

14.21.2.3 K_ULONG ProfileTimer::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 89 of file profile.cpp.

```
14.21.2.4 void ProfileTimer::Init ( void )
```

Initialize the profiling timer prior to use.

Can also be used to reset a timer that's been used previously.

Definition at line 37 of file profile.cpp.

```
14.21.2.5 void ProfileTimer::Start (void)
```

Start a profiling session, if the timer is not already active.

Has no effect if the timer is already active.

Definition at line 46 of file profile.cpp.

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

- /home/moslevin/Project/R1/kernel/public/profile.h
- /home/moslevin/Project/R1/kernel/profile.cpp

14.22 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 ()

This function is called to update the thread quantum timer whenever something in the scheduler has changed.

static void AddThread (Thread *pclThread)

Add the thread to the quantum timer.

• static void RemoveThread ()

Remove the thread from the quantum timer.

static void SetInTimer (void)

SetInTimer.

• static void ClearInTimer (void)

ClearInTimer.

Static Private Member Functions

static void SetTimer (Thread *pclThread_)

Set up the quantum timer in the timer scheduler.

14.22.1 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.

14.22.2 Member Function Documentation

```
14.22.2.1 void Quantum::AddThread ( Thread * pclThread_ ) [static]
```

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 82 of file quantum.cpp.

```
14.22.2.2 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.

```
14.22.2.3 void Quantum::RemoveThread ( void ) [static]
```

Remove the thread from the quantum timer.

This will cancel the timer.

Definition at line 111 of file quantum.cpp.

```
14.22.2.4 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.

```
14.22.2.5 void Quantum::SetTimer( Thread * pclThread_ ) [static], [private]
```

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

```
pclThread_ Pointer to the thread to set the Quantum timer on
```

Definition at line 72 of file quantum.cpp.

```
14.22.2.6 void Quantum::UpdateTimer(void) [static]
```

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 if may be ignored on expiry. Definition at line 124 of file quantum.cpp.

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

- /home/moslevin/Project/R1/kernel/public/quantum.h
- /home/moslevin/Project/R1/kernel/quantum.cpp

14.23 Scheduler Class Reference

Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping.

```
#include <scheduler.h>
```

Static Public Member Functions

· static void Init ()

Intiailize the scheduler, must be called before use.

• static void Schedule ()

Run the scheduler, determines the next thread to run based on the current state of the threads.

static void Add (Thread *pclThread_)

Add a thread to the scheduler at its current priority level.

• static void Remove (Thread *pclThread_)

Remove a thread from the scheduler at its current priority level.

static K BOOL SetScheduler (K BOOL bEnable)

Set the active state of the scheduler.

static Thread * GetCurrentThread ()

Return the pointer to the currently-running thread.

static volatile Thread * GetNextThread ()

Return the pointer to the thread that should run next, according to the last run of the scheduler.

• static ThreadList * GetThreadList (K_UCHAR ucPriority_)

Return the pointer to the active list of threads that are at the given priority level in the scheduler.

static ThreadList * GetStopList ()

Return the pointer to the list of threads that are in the scheduler's stopped state.

static K_UCHAR IsEnabled ()

Return the current state of the scheduler - whether or not scheddling is enabled or disabled.

• static void QueueScheduler ()

QueueScheduler.

Static Private Attributes

static K_BOOL m_bEnabled

Scheduler's state - enabled or disabled.

• static K_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 K_UCHAR m_ucPriFlag

Bitmap flag for each.

14.23.1 Detailed Description

Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping.

Definition at line 62 of file scheduler.h.

14.23.2 Member Function Documentation

14.23.2.1 void Scheduler::Add (Thread * pclThread_) [static]

Add a thread to the scheduler at its current priority level.

Parameters

pclThread_ Pointer to the thread to add to the scheduler

Definition at line 108 of file scheduler.cpp.

14.23.2.2 static Thread* Scheduler::GetCurrentThread() [inline], [static]

Return the pointer to the currently-running thread.

Returns

Pointer to the currently-running thread

Definition at line 119 of file scheduler.h.

14.23.2.3 static volatile Thread* Scheduler::GetNextThread() [inline], [static]

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 127 of file scheduler.h.

14.23.2.4 static ThreadList* Scheduler::GetStopList() [inline], [static]

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 145 of file scheduler.h.

14.23.2.5 static ThreadList* Scheduler::GetThreadList(K_UCHAR ucPriority_) [inline], [static]

Return the pointer to the active list of threads that are at the given priority level in the scheduler.

Parameters

ucPriority_ Priority level of

Returns

Pointer to the ThreadList for the given priority level

Definition at line 137 of file scheduler.h.

14.23.2.6 K_UCHAR Scheduler::lsEnabled() [inline], [static]

Return the current state of the scheduler - whether or not scheddling is enabled or disabled.

Returns

true - scheduler enabled, false - disabled

Definition at line 155 of file scheduler.h.

```
14.23.2.7 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 163 of file scheduler.h.

```
14.23.2.8 void Scheduler::Remove ( Thread * pclThread_ ) [static]
```

Remove a thread from the scheduler at its current priority level.

Parameters

```
pclThread_ Pointer to the thread to be removed from the scheduler
```

Definition at line 114 of file scheduler.cpp.

```
14.23.2.9 Scheduler::Schedule( ) [static]
```

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 71 of file scheduler.cpp.

```
14.23.2.10 void Scheduler::SetScheduler( K_BOOL bEnable_) [static]
```

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

```
bEnable_ true to enable, false to disable the scheduler
```

Definition at line 120 of file scheduler.cpp.

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

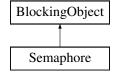
- /home/moslevin/Project/R1/kernel/public/scheduler.h
- /home/moslevin/Project/R1/kernel/scheduler.cpp

14.24 Semaphore Class Reference

Counting semaphore, based on BlockingObject base class.

```
#include <ksemaphore.h>
```

Inheritance diagram for Semaphore:



Public Member Functions

void Init (K_USHORT usInitVal_, K_USHORT usMaxVal_)

Initialize a semaphore before use.

• K BOOL Post ()

Increment the semaphore count.

• void Pend ()

Decrement the semaphore count.

K_USHORT GetCount ()

Return the current semaphore counter.

K_BOOL Pend (K_ULONG ulWaitTimeMS_)

Decrement the semaphore count.

void WakeMe (Thread *pclChosenOne_)

Wake a thread blocked on the semaphore.

Private Member Functions

• K_UCHAR WakeNext ()

Wake the next thread waiting on the semaphore.

K_BOOL Pend_i (K_ULONG ulWaitTimeMS_)

Pend i.

Private Attributes

• K_USHORT m_usValue

Current count held by the semaphore.

• K USHORT m usMaxValue

Maximum count that can be held by this semaphore.

Additional Inherited Members

14.24.1 Detailed Description

Counting semaphore, based on BlockingObject base class.

Definition at line 37 of file ksemaphore.h.

14.24.2 Member Function Documentation

14.24.2.1 K_USHORT Semaphore::GetCount ()

Return the current semaphore counter.

This can be used 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 234 of file ksemaphore.cpp.

14.24.2.2 void Semaphore::Init (K_USHORT usInitVal_, K_USHORT usMaxVal_)

Initialize a semaphore before use.

Must be called before post/pend operations.

Parameters

usInitVal_	Initial value held by the semaphore
usMaxVal_	Maximum value for the semaphore

Definition at line 95 of file ksemaphore.cpp.

```
14.24.2.3 void Semaphore::Pend ( )
```

Decrement the semaphore count.

If the count is zero, the thread will block until the semaphore is pended.

Definition at line 216 of file ksemaphore.cpp.

```
14.24.2.4 K BOOL Semaphore::Pend ( K ULONG ulWaitTimeMS_ )
```

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 227 of file ksemaphore.cpp.

```
14.24.2.5 K_BOOL Semaphore::Pend_i( K_ULONG u/WaitTimeMS_ ) [private]
```

Pend i.

Internal function used to abstract timed and untimed semaphore pend operations.

Parameters

ulWaitTimeMS_	Time in MS to wait
---------------	--------------------

Returns

true on success, false on failure.

Definition at line 160 of file ksemaphore.cpp.

```
14.24.2.6 void Semaphore::Post ( )
```

Increment the semaphore count.

Returns

true if the semaphore was posted, false if the count is already maxed out.

Definition at line 107 of file ksemaphore.cpp.

```
14.24.2.7 void Semaphore::WakeMe ( Thread * pclChosenOne_ )
```

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 68 of file ksemaphore.cpp.

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

- /home/moslevin/Project/R1/kernel/public/ksemaphore.h
- /home/moslevin/Project/R1/kernel/ksemaphore.cpp

14.25 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 *paucStack_, K_USHORT usStackSize_, K_UCHAR ucPriority_, ThreadEntry_t pfEntry-Point_, void *pvArg_)

Initialize a thread prior to its use.

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.

• void Stop ()

Stop a thread that's actively scheduled without destroying its stacks.

ThreadList * GetOwner (void)

Return the ThreadList where the thread belongs when it's in the active/ready state in the scheduler.

ThreadList * GetCurrent (void)

Return the ThreadList where the thread is currently located.

• K_UCHAR GetPriority (void)

Return the priority of the current thread.

K_UCHAR GetCurPriority (void)

Return the priority of the current thread.

void SetQuantum (K_USHORT usQuantum_)

Set the thread's round-robin execution quantum.

K_USHORT GetQuantum (void)

Get the thread's round-robin execution quantum.

void SetCurrent (ThreadList *pclNewList_)

Set the thread's current to the specified thread list.

void SetOwner (ThreadList *pclNewList_)

Set the thread's owner to the specified thread list.

void SetPriority (K_UCHAR ucPriority_)

Set the priority of the Thread (running or otherwise) to a different level.

• void InheritPriority (K_UCHAR ucPriority_)

Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions.

• void Exit ()

Remove the thread from being scheduled again.

void SetID (K_UCHAR ucID_)

Set an 8-bit ID to uniquely identify this thread.

• K_UCHAR GetID ()

Return the 8-bit ID corresponding to this thread.

K_USHORT GetStackSlack ()

Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack.

K_USHORT GetEventFlagMask ()

GetEventFlagMask returns the thread's current event-flag mask, which is used in conjunction with the EventFlag blocking object type.

void SetEventFlagMask (K_USHORT usMask_)

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 (K_BOOL bExpired_)

SetExpired.

• K_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 (K ULONG ulTimeMs)

Put the thread to sleep for the specified time (in milliseconds).

static void USleep (K_ULONG ulTimeUs_)

Put the thread to sleep for the specified time (in microseconds).

static void Yield (void)

Yield the thread - this forces the system to call the scheduler and determine what thread should run next.

Private Member Functions

• void SetPriorityBase (K_UCHAR ucPriority_)

Static Private Member Functions

static void ContextSwitchSWI (void)

This code is used to trigger the context switch interrupt.

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.

· K UCHAR m ucThreadID

Thread ID.

• K_UCHAR m_ucPriority

Default priority of the thread.

· K UCHAR m ucCurPriority

Current priority of the thread (priority inheritence)

• ThreadState t m eState

Enum indicating the thread's current state.

• K_USHORT m_usStackSize

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 entrypoint.

K_USHORT m_usQuantum

Thread quantum (in milliseconds)

K_USHORT m_usFlagMask

Event-flag mask.

EventFlagOperation_t m_eFlagMode

Event-flag mode.

• Timer m_clTimer

Timer used for blocking-object timeouts.

• K_BOOL m_bExpired

Indicate whether or not a blocking-object timeout has occurred.

Friends

· class ThreadPort

Additional Inherited Members

14.25.1 Detailed Description

Object providing fundamental multitasking support in the kernel.

Definition at line 71 of file thread.h.

14.25.2 Member Function Documentation

```
14.25.2.1 void Thread::ContextSwitchSWI(void) [static], [private]
```

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 395 of file thread.cpp.

```
14.25.2.2 void Thread::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.

Definition at line 174 of file thread.cpp.

```
14.25.2.3 K_UCHAR Thread::GetCurPriority (void ) [inline]
```

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 174 of file thread.h.

```
14.25.2.4 ThreadList * Thread::GetCurrent(void) [inline]
```

Return the ThreadList where the thread is currently located.

Returns

Pointer to the thread's current list

Definition at line 155 of file thread.h.

```
14.25.2.5 K_USHORT 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 327 of file thread.h.

```
14.25.2.6 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 346 of file thread.h.

```
14.25.2.7 K 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 413 of file thread.cpp.

```
14.25.2.8 K_UCHAR Thread::GetID() [inline]
```

Return the 8-bit ID corresponding to this thread.

Returns

Thread's 8-bit ID, set by the user

Definition at line 302 of file thread.h.

```
14.25.2.9 ThreadList * Thread::GetOwner(void) [inline]
```

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 146 of file thread.h.

```
14.25.2.10 K_UCHAR Thread::GetPriority (void ) [inline]
```

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 165 of file thread.h.

```
14.25.2.11 K_USHORT Thread::GetQuantum ( void ) [inline]
```

Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 193 of file thread.h.

```
14.25.2.12 K_USHORT Thread::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 284 of file thread.cpp.

```
14.25.2.13 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

Definition at line 390 of file thread.h.

```
14.25.2.14 void Thread::InheritPriority ( K_UCHAR ucPriority_ )
```

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

ucPriority_	New Priority to boost to.
-------------	---------------------------

Definition at line 388 of file thread.cpp.

```
14.25.2.15 void Thread::Init ( K_WORD * paucStack_, K_USHORT usStackSize_, K_UCHAR ucPriority_, ThreadEntry_t pfEntryPoint_, void * pvArg_ )
```

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

paucStack_	Pointer to the stack to use for the thread
usStackSize_	Size of the stack (in bytes)
ucPriority_	Priority of the thread (0 = idle, 7 = max)
pfEntryPoint_	This is the function that gets called when the thread is started
pvArg_	Pointer to the argument passed into the thread's entrypoint function.

Definition at line 41 of file thread.cpp.

```
14.25.2.16 void Thread::InitIdle (void)
```

InitIdle Initialize this Thread object as the Kernel's idle thread.

94 Class Documentation

There should only be one of these, maximum, in a given system.

Definition at line 418 of file thread.cpp.

```
14.25.2.17 void Thread::SetCurrent ( ThreadList * pclNewList_ ) [inline]
```

Set the thread's current to the specified thread list.

Parameters

```
pclNewList_ Pointer to the threadlist to apply thread ownership
```

Definition at line 203 of file thread.h.

```
14.25.2.18 void Thread::SetEventFlagMask ( K_USHORT usMask_ ) [inline]
```

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

```
usMask_
```

Definition at line 333 of file thread.h.

```
14.25.2.19 void Thread::SetEventFlagMode ( EventFlagOperation t eMode_ ) [inline]
```

SetEventFlagMode Sets the active event flag operation mode.

Parameters

```
eMode_ | Event flag operation mode, defines the logical operator to apply to the event flag.
```

Definition at line 340 of file thread.h.

```
14.25.2.20 void Thread::SetExpired ( K_BOOL bExpired_ )
```

SetExpired.

Set the status of the current blocking call on the thread.

Parameters

```
bExpired_ true - call expired, false - call did not expire
```

Definition at line 410 of file thread.cpp.

```
14.25.2.21 void Thread::SetID ( K_UCHAR uclD_ ) [inline]
```

Set an 8-bit ID to uniquely identify this thread.

Parameters

```
ucID_ 8-bit Thread ID, set by the user
```

Definition at line 293 of file thread.h.

```
14.25.2.22 void Thread::SetOwner ( ThreadList * pclNewList_ ) [inline]
```

Set the thread's owner to the specified thread list.

Parameters

pclNewList_ Pointer to the threadlist to apply thread ownership

Definition at line 212 of file thread.h.

14.25.2.23 void Thread::SetPriority (K_UCHAR ucPriority_)

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

```
ucPriority_ New priority of the thread
```

Definition at line 344 of file thread.cpp.

14.25.2.24 void Thread::SetPriorityBase (K_UCHAR ucPriority_) [private]

Parameters

ucPriority_

Definition at line 334 of file thread.cpp.

14.25.2.25 void Thread::SetQuantum (K_USHORT usQuantum_) [inline]

Set the thread's round-robin execution quantum.

Parameters

usQuantum Thread's execution quantum (in milliseconds)

Definition at line 184 of file thread.h.

14.25.2.26 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 indended for use by an end-user.

Parameters

eState_ New thread state to set.

Definition at line 399 of file thread.h.

14.25.2.27 void Thread::Sleep (K_ULONG ulTimeMs_) [static]

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

96 Class Documentation

ulTimeMs_ Time to sleep (in ms)

Definition at line 239 of file thread.cpp.

14.25.2.28 void Thread::Stop ()

Stop a thread that's actively scheduled without destroying its stacks.

Stopped threads can be restarted using the Start() API.

Definition at line 129 of file thread.cpp.

```
14.25.2.29 void Thread::USleep ( K_ULONG ulTimeUs_ ) [static]
```

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

```
ulTimeUs_ Time to sleep (in microseconds)
```

Definition at line 261 of file thread.cpp.

```
14.25.2.30 void Thread::Yield (void ) [static]
```

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 305 of file thread.cpp.

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

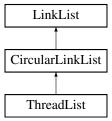
- /home/moslevin/Project/R1/kernel/public/thread.h
- /home/moslevin/Project/R1/kernel/thread.cpp

14.26 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 ()

Default constructor - zero-initializes the data.

void SetPriority (K_UCHAR ucPriority_)

Set the priority of this threadlist (if used for a scheduler).

void SetFlagPointer (K_UCHAR *pucFlag_)

Set the pointer to a bitmap to use for this threadlist.

void Add (LinkListNode *node_)

Add a thread to the threadlist.

void Add (LinkListNode *node_, K_UCHAR *pucFlag_, K_UCHAR ucPriority_)

Add a thread to the threadlist, specifying the flag and priority at the same time.

void Remove (LinkListNode *node_)

Remove the specified thread from the threadlist.

Thread * HighestWaiter ()

Return a pointer to the highest-priority thread in the thread-list.

Private Attributes

· K UCHAR m ucPriority

Priority of the threadlist.

K_UCHAR * m_pucFlag

Pointer to the bitmap/flag to set when used for scheduling.

Additional Inherited Members

14.26.1 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.

14.26.2 Member Function Documentation

```
14.26.2.1 void ThreadList::Add ( LinkListNode * node_ ) [virtual]
```

Add a thread to the threadlist.

Parameters

node_	Pointer to the thread (link list node) to add to the list

Reimplemented from CircularLinkList.

Definition at line 46 of file threadlist.cpp.

```
14.26.2.2 void ThreadList::Add ( LinkListNode * node_, K_UCHAR * pucFlag_, K_UCHAR ucPriority_ )
```

Add a thread to the threadlist, specifying the flag and priority at the same time.

Parameters

	node_	Pointer to the thread to add (link list node)
Ì	pucFlag_	Pointer to the bitmap flag to set (if used in a scheduler context), or NULL for non-scheduler.
Ì	ucPriority_	Priority of the threadlist

Definition at line 59 of file threadlist.cpp.

98 Class Documentation

```
14.26.2.3 Thread * ThreadList::HighestWaiter ( )
```

Return a pointer to the highest-priority thread in the thread-list.

Returns

Pointer to the highest-priority thread

Definition at line 84 of file threadlist.cpp.

```
14.26.2.4 void ThreadList::Remove ( LinkListNode * node_ ) [virtual]
```

Remove the specified thread from the threadlist.

Parameters

```
node_ Pointer to the thread to remove
```

Reimplemented from CircularLinkList.

Definition at line 68 of file threadlist.cpp.

```
14.26.2.5 void ThreadList::SetFlagPointer ( K_UCHAR * pucFlag_ )
```

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

```
pucFlag_ Pointer to the bitmap flag
```

Definition at line 40 of file threadlist.cpp.

```
14.26.2.6 void ThreadList::SetPriority ( K_UCHAR ucPriority_ )
```

Set the priority of this threadlist (if used for a scheduler).

Parameters

```
ucPriority_ Priority level of the thread list
```

Definition at line 34 of file threadlist.cpp.

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

- /home/moslevin/Project/R1/kernel/public/threadlist.h
- /home/moslevin/Project/R1/kernel/threadlist.cpp

14.27 ThreadPort Class Reference

Class defining the architecture specific functions required by the kernel.

```
#include <threadport.h>
```

Static Public Member Functions

static void StartThreads ()

Function to start the scheduler, initial threads, etc.

Static Private Member Functions

static void InitStack (Thread *pstThread_)

Initialize the thread's stack.

Friends

· class Thread

14.27.1 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.

14.27.2 Member Function Documentation

```
14.27.2.1 void ThreadPort::InitStack ( Thread * pstThread_ ) [static], [private]
```

Initialize the thread's stack.

Parameters

```
pstThread_ Pointer to the thread to initialize
```

Definition at line 39 of file threadport.cpp.

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

- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/threadport.h
- /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/threadport.cpp

14.28 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 ()

Default Constructor - zero-initializes all internal data.

• void Init ()

Re-initialize the Timer to default values.

100 Class Documentation

void Start (K_BOOL bRepeat_, K_ULONG ulIntervalMs_, TimerCallback_t pfCallback_, void *pvData_)
 Start a timer using default ownership, using repeats as an option, and millisecond resolution.

 void Start (K_BOOL bRepeat_, K_ULONG ulIntervalMs_, K_ULONG ulToleranceMs_, TimerCallback_t pf-Callback_, void *pvData_)

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

void Stop ()

Stop a timer already in progress.

void SetFlags (K_UCHAR ucFlags_)

Set the timer's flags based on the bits in the ucFlags_ argument.

void SetCallback (TimerCallback_t pfCallback_)

Define the callback function to be executed on expiry of the timer.

void SetData (void *pvData_)

Define a pointer to be sent to the timer callbcak on timer expiry.

void SetOwner (Thread *pclOwner)

Set the owner-thread of this timer object (all timers must be owned by a thread).

void SetIntervalTicks (K ULONG ulTicks)

Set the timer expiry in system-ticks (platform specific!)

• void SetIntervalSeconds (K_ULONG ulSeconds_)

! The next three cost us 330 bytes of flash on AVR...

void SetIntervalMSeconds (K_ULONG ulMSeconds_)

Set the timer expiry interval in milliseconds (platform agnostic)

void SetIntervalUSeconds (K ULONG ulUSeconds)

Set the timer expiry interval in microseconds (platform agnostic)

void SetTolerance (K_ULONG ulTicks_)

Set the timer's maximum tolerance in order to synchronize timer processing with other timers in the system.

Private Attributes

K_UCHAR m_ucFlags

Flags for the timer, defining if the timer is one-shot or repeated.

TimerCallback_t m_pfCallback

Pointer to the callback function.

· K ULONG m ulInterval

Interval of the timer in timer ticks.

K_ULONG m_ulTimeLeft

Time remaining on the timer.

K_ULONG m_ulTimerTolerance

Maximum tolerance (used for timer harmonization)

• Thread * m_pclOwner

Pointer to the owner thread.

void * m_pvData

Pointer to the callback data.

Friends

· class TimerList

14.28 Timer Class Reference 101

Additional Inherited Members

14.28.1 Detailed Description

Timer - an event-driven execution context based on a specified time interval.

This inherits from a LinkListNode for ease of management by a global TimerList object.

Definition at line 105 of file timer.h.

14.28.2 Member Function Documentation

```
14.28.2.1 void Timer::SetCallback ( TimerCallback_t pfCallback_ ) [inline]
```

Define the callback function to be executed on expiry of the timer.

Parameters

```
pfCallback_ Pointer to the callback function to call
```

Definition at line 166 of file timer.h.

```
14.28.2.2 void Timer::SetData (void * pvData_) [inline]
```

Define a pointer to be sent to the timer callbcak on timer expiry.

Parameters

```
pvData_ Pointer to data to pass as argument into the callback
```

Definition at line 175 of file timer.h.

```
14.28.2.3 void Timer::SetFlags ( K_UCHAR ucFlags_ ) [inline]
```

Set the timer's flags based on the bits in the ucFlags_ argument.

Parameters

ucFlags_	Flags to assign to the timer object. TIMERLIST_FLAG_ONE_SHOT for a one-shot timer, 0
	for a continuous timer.

Definition at line 157 of file timer.h.

14.28.2.4 void Timer::SetIntervalMSeconds (K_ULONG ulMSeconds_)

Set the timer expiry interval in milliseconds (platform agnostic)

Parameters

```
ulMSeconds_ Time in milliseconds
```

Definition at line 88 of file timer.cpp.

```
14.28.2.5 void Timer::SetIntervalSeconds ( K_ULONG ulSeconds_ )
```

! The next three cost us 330 bytes of flash on AVR...

Set the timer expiry interval in seconds (platform agnostic)

102 Class Documentation

Parameters

ulSeconds_ Time in seconds

Definition at line 82 of file timer.cpp.

14.28.2.6 void Timer::SetIntervalTicks (K_ULONG ulTicks_)

Set the timer expiry in system-ticks (platform specific!)

Parameters

```
ulTicks_ Time in ticks
```

Definition at line 74 of file timer.cpp.

14.28.2.7 void Timer::SetIntervalUSeconds (K_ULONG ulUSeconds_)

Set the timer expiry interval in microseconds (platform agnostic)

Parameters

ulUSeconds_	Time in microseconds
-------------	----------------------

Definition at line 94 of file timer.cpp.

14.28.2.8 void Timer::SetOwner (Thread * pclOwner_) [inline]

Set the owner-thread of this timer object (all timers must be owned by a thread).

Parameters

```
pclOwner_ Owner thread of this timer object
```

Definition at line 185 of file timer.h.

14.28.2.9 void Timer::SetTolerance (K_ULONG ulTicks_)

Set the timer's maximum tolerance in order to synchronize timer processing with other timers in the system.

Parameters

ulTicks_	Maximum tolerance in ticks

Definition at line 100 of file timer.cpp.

14.28.2.10 void Timer::Start (K_BOOL bRepeat_, K_ULONG ulIntervalMs_, TimerCallback_t pfCallback_, void * pvData_)

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

bRepeat_	0 - timer is one-shot. 1 - timer is repeating.
ulIntervalMs_	- Interval of the timer in miliseconds
pfCallback_	- Function to call on timer expiry

pvData_	- Data to pass into the callback function

Definition at line 42 of file timer.cpp.

```
14.28.2.11 void Timer::Start ( K_BOOL bRepeat_, K_ULONG ulIntervalMs_, K_ULONG ulToleranceMs_, TimerCallback_t pfCallback_, void * pvData_ )
```

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

bRepeat_	0 - timer is one-shot. 1 - timer is repeating.
ulIntervalMs_	- Interval of the timer in miliseconds
ulToleranceMs	- 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.
pfCallback_	- Function to call on timer expiry
pvData_	- Data to pass into the callback function

Definition at line 61 of file timer.cpp.

```
14.28.2.12 void Timer::Stop ( )
```

Stop a timer already in progress.

Has no effect on timers that have already been stopped.

Definition at line 68 of file timer.cpp.

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

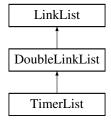
- /home/moslevin/Project/R1/kernel/public/timer.h
- /home/moslevin/Project/R1/kernel/timer.cpp

14.29 TimerList Class Reference

TimerList class - a doubly-linked-list of timer objects.

```
#include <timerlist.h>
```

 $Inheritance\ diagram\ for\ Timer List:$



Public Member Functions

void Init ()

Initialize the TimerList object.

void Add (Timer *pclListNode_)

Add a timer to the TimerList.

void Remove (Timer *pclListNode_)

104 Class Documentation

Remove a timer from the TimerList, cancelling its expiry.

· void Process ()

Process all timers in the timerlist as a result of the timer expiring.

Private Attributes

K_ULONG m_ulNextWakeup

The time (in system clock ticks) of the next wakeup event.

• K_UCHAR m_bTimerActive

Whether or not the timer is active.

Additional Inherited Members

14.29.1 Detailed Description

TimerList class - a doubly-linked-list of timer objects.

Definition at line 37 of file timerlist.h.

14.29.2 Member Function Documentation

```
14.29.2.1 void TimerList::Add ( Timer * pclListNode_ )
```

Add a timer to the TimerList.

Parameters

```
pclListNode_ Pointer to the Timer to Add
```

Definition at line 51 of file timerlist.cpp.

```
14.29.2.2 void TimerList::Init (void)
```

Initialize the TimerList object.

Must be called before using the object.

Definition at line 44 of file timerlist.cpp.

```
14.29.2.3 void TimerList::Process (void)
```

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 116 of file timerlist.cpp.

```
14.29.2.4 void TimerList::Remove ( Timer * pclListNode_ )
```

Remove a timer from the TimerList, cancelling its expiry.

Parameters

pclListNode_ Pointer to the Timer to remove

Definition at line 99 of file timerlist.cpp.

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

- /home/moslevin/Project/R1/kernel/public/timerlist.h
- /home/moslevin/Project/R1/kernel/timerlist.cpp

14.30 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 ()

Initialize the timer scheduler.

static void Add (Timer *pclListNode_)

Add a timer to the timer scheduler.

• static void Remove (Timer *pclListNode)

Remove a timer from the timer scheduler.

• static void Process ()

This function must be called on timer expiry (from the timer's ISR context).

Static Private Attributes

• static TimerList m clTimerList

TimerList object manipulated by the Timer Scheduler.

14.30.1 Detailed Description

"Static" Class used to interface a global TimerList with the rest of the kernel.

Definition at line 38 of file timerscheduler.h.

14.30.2 Member Function Documentation

```
14.30.2.1 void TimerScheduler::Add ( Timer * pclListNode_ ) [inline], [static]
```

Add a timer to the timer scheduler.

Adding a timer implicitly starts the timer as well.

Parameters

Pointer to the timer list node to add
F

Definition at line 57 of file timerscheduler.h.

106 Class Documentation

```
14.30.2.2 void TimerScheduler::Init(void) [inline], [static]
```

Initialize the timer scheduler.

Must be called before any timer, or timer-derived functions are used.

Definition at line 47 of file timerscheduler.h.

```
14.30.2.3 void TimerScheduler::Process (void ) [inline], [static]
```

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.

```
14.30.2.4 void TimerScheduler::Remove ( Timer * pclListNode_ ) [inline], [static]
```

Remove a timer from the timer scheduler.

May implicitly stop the timer if this is the only active timer scheduled.

Parameters

```
pclListNode_ 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/moslevin/Project/R1/kernel/public/timerscheduler.h
- /home/moslevin/Project/R1/kernel/timerlist.cpp

Chapter 15

File Documentation

15.1 /home/moslevin/Project/R1/kernel/atomic.cpp File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "atomic.h"
#include "threadport.h"
```

15.1.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.cpp.

15.2 atomic.cpp

```
00001 /
00002
00003
00004 |
00005 1
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
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "atomic.h"
00024 #include "threadport.h"
00025
00026 #if KERNEL_USE_ATOMIC
00029 K_UCHAR Atomic::Set( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00030 {
00031
         K UCHAR ucRet;
00032
        CS_ENTER();
       ucRet = *pucSource_;
00033
00034
        *pucSource_ = ucVal_;
00035
        CS_EXIT();
00036
         return ucRet;
00037 }
00038 //--
00039 K_USHORT Atomic::Set( K_USHORT *pusSource_, K_USHORT usVal_ )
```

```
K_USHORT usRet;
00041
00042
         CS_ENTER();
00043
         usRet = *pusSource_;
00044
         *pusSource_ = usVal_;
00045
         CS EXIT();
00046
         return usRet:
00047 }
00048 //---
00049 K_ULONG Atomic::Set( K_ULONG *pulSource_, K_ULONG ulVal_ )
00050 {
00051
         K ULONG ulRet:
         CS_ENTER();
00052
00053
         ulRet = *pulSource_;
         *pulSource_ = ulVal_;
00054
00055
         CS_EXIT();
00056
         return ulRet;
00057 }
00058
00059 //---
00060 K_UCHAR Atomic::Add( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00061 {
00062
         K_UCHAR ucRet;
00063
         CS_ENTER();
00064
         ucRet = *pucSource_;
00065
         *pucSource_ += ucVal_;
00066
         CS_EXIT();
00067
         return ucRet;
00068 }
00069
00070 //----
00071 K_USHORT Atomic::Add( K_USHORT *pusSource_, K_USHORT usVal_ )
00072 {
00073
         K_USHORT usRet;
00074
         CS_ENTER();
00075
         usRet = *pusSource_;
00076
         *pusSource_ += usVal_;
00077
         CS EXIT();
00078
         return usRet;
00079 }
08000
00081 //----
00082 K_ULONG Atomic::Add( K_ULONG *pulSource_, K_ULONG ulVal_ )
00083 {
00084
         K_ULONG ulRet;
00085
         CS_ENTER();
00086
         ulRet = *pulSource_;
00087
         *pulSource_ += ulVal_;
00088
         CS_EXIT();
00089
         return ulRet:
00090 }
00091
00092 //----
00093 K_UCHAR Atomic::Sub( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00094 {
00095
         K_UCHAR ucRet;
00096
         CS_ENTER();
00097
         ucRet = *pucSource_;
00098
         *pucSource_ -= ucVal_;
00099
         CS_EXIT();
00100
         return ucRet;
00101 }
00102
00103 //-
00104 K_USHORT Atomic::Sub( K_USHORT *pusSource_, K_USHORT usVal_ )
00105 {
00106
         K_USHORT usRet;
00107
         CS_ENTER();
         usRet = *pusSource_;
00108
00109
         *pusSource_ -= usVal_;
00110
         CS_EXIT();
00111
         return usRet;
00112 }
00113
00114 //-----
00115 K_ULONG Atomic::Sub( K_ULONG *pulSource_, K_ULONG ulVal_ )
00116 {
         K_ULONG ulRet;
00117
00118
         CS_ENTER();
         ulRet = *pulSource_;
00119
         *pulSource_ -= ulVal_;
00120
00121
         CS EXIT();
00122
         return ulRet;
00123 }
00124
00125 //---
00126 K_BOOL Atomic::TestAndSet( K_BOOL *pbLock_ )
00127 {
```

```
00128
          K_UCHAR ucRet;
00129
          CS_ENTER();
00130
          ucRet = *pbLock_;
00131
          if (!ucRet)
00132
00133
              *pbLock_ = 1;
00134
00135
          CS_EXIT();
00136
          return ucRet;
00137 }
00138
00139 #endif // KERNEL_USE_ATOMIC
```

15.3 /home/moslevin/Project/R1/kernel/blocking.cpp File Reference

Implementation of base class for blocking objects.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kerneldebug.h"
#include "blocking.h"
#include "thread.h"
```

Macros

#define __FILE_ID__ BLOCKING_CPP
 File ID used in kernel trace calls.

15.3.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

15.4 blocking.cpp

```
00001
00002
00003
00004
00005
00006
00007
80000
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 "kerneldebug.h"
00024
00025 #include "blocking.h"
00026 #include "thread.h"
00027
00028 //----
00029 #if defined __FILE_ID__
00030
         #undef ___FILE_ID___
00031 #endif
00032 #define ___FILE_ID__
00033
00034 #if KERNEL_USE_SEMAPHORE || KERNEL_USE_MUTEX
00035 //---
00036 void BlockingObject::Block(Thread *pclThread)
00037 {
00038
           KERNEL_ASSERT( pclThread_ );
```

```
00039
          KERNEL_TRACE_1 ( STR_THREAD_BLOCK_1, (K_USHORT)pclThread_->
00040
00041
          // Remove the thread from its current thread list (the "owner" list)
00042
          // ... And add the thread to this object's block list
00043
          Scheduler::Remove(pclThread_);
          m_clBlockList.Add(pclThread_);
00045
00046
          // Set the "current" list location to the blocklist for this thread
00047
          pclThread ->SetCurrent(&m clBlockList);
          pclThread_->SetState(THREAD_STATE_BLOCKED);
00048
00049 }
00050
00051 //-
00052 void BlockingObject::UnBlock(Thread *pclThread_)
00053 {
          KERNEL_ASSERT( pclThread_ );
00054
          KERNEL_TRACE_1( STR_THREAD_UNBLOCK_1, (K_USHORT)pclThread_->
00055
00056
00057
           // Remove the thread from its current thread list (the "owner" list)
00058
          pclThread_->GetCurrent()->Remove(pclThread_);
00059
          // Put the thread back in its active owner's list. This is usually // the ready-queue at the thread's original priority.
00060
00061
          Scheduler::Add(pclThread_);
00062
00063
00064
          // Tag the thread's current list location to its owner
00065
          pclThread_->SetCurrent(pclThread_->GetOwner());
00066
          pclThread_->SetState(THREAD_STATE_READY);
00067 }
00068
00069 #endif
```

15.5 /home/moslevin/Project/R1/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>
```

15.5.1 Detailed Description

ATMega328p Profiling timer implementation.

Definition in file kernelprofile.cpp.

15.6 kernelprofile.cpp

```
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>
00028 #if KERNEL_USE_PROFILER
00029 K_ULONG Profiler::m_ulEpoch;
00030
00031 //----
00032 void Profiler::Init()
00033 {
00034
          TCCR0A = 0;
00035
          TCCROB = 0;
        TIFR0 = 0;
TIMSK0 = 0;
00036
00037
00038
         m_ulEpoch = 0;
00039 }
00040
00041 //---
00042 void Profiler::Start()
00043 {
          TIFR0 = 0:
00044
00045
          TCNT0 = 0;
      TCCR0B |= (1 << CS01);
00047
          TIMSKO \mid = (1 << TOIE0);
00048 }
00049
00050 //----
00051 void Profiler::Stop()
00052 {
00052
          TIFR0 = 0;
00054
          TCCR0B &= \sim (1 << CS01);
         TIMSK0 &= \sim (1 << TOIE0);
00055
00056 }
00057 //---
00058 K_USHORT Profiler::Read()
00059 {
00060
          K_USHORT usRet;
        CS_ENTER();

TCCROB &= ~(1 << CSO1);

usRet = TCNTO;

TCCROB |= (1 << CSO1);

CS_EXIT();
00061
00062
00063
00064
00065
00066
          return usRet;
00067 }
00068
00069 //---
00070 void Profiler::Process()
00071 {
00072
          CS_ENTER();
        m_ulEpoch++;
CS_EXIT();
00073
00074
00075 }
00076
00077 //--
00078 ISR(TIMERO_OVF_vect)
00079 {
08000
          Profiler::Process();
00081 }
00082
00083 #endif
```

15.7 /home/moslevin/Project/R1/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>
```

15.7.1 Detailed Description

Kernel Software interrupt implementation for ATMega328p.

Definition in file kernelswi.cpp.

15.8 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 {
          PORTD &= \sim 0 \times 04; // Clear INTO
00031
00032
          DDRD \mid = 0x04;
                           // Set PortD, bit 2 (INTO) As Output
00033
          00034 }
00035
00036 //--
00037 void KernelSWI::Start(void)
00038 {
00039
          EIFR &= \sim (1 << INTF0);
                                    // Clear any pending interrupts on INTO
00040
          EIMSK \mid = (1 << INT0);
                                  // Enable INTO interrupt (as K_LONG as I-bit is set)
00041 }
00042
00043 //-
00044 void KernelSWI::Stop(void)
00045 {
00046
          EIMSK &= ~(1 << INTO);  // Disable INTO interrupts</pre>
00047 }
00048
00049 //-
00050 K_UCHAR KernelSWI::DI()
00051 {
00052
          K\_BOOL bEnabled = ((EIMSK & (1 << INTO)) != 0);
00053
         EIMSK &= \sim (1 << INT0);
00054
         return bEnabled:
00055 }
00056
00057 //--
00058 void KernelSWI::RI(K_BOOL bEnable_)
00059 {
00060
          if (bEnable_)
00061
          {
00062
              EIMSK |= (1 << INTO);
00063
00064
00065
00066
              EIMSK &= \sim (1 << INT0);
00067
00068 }
00069
00070 //--
00071 void KernelSWI::Clear(void)
00072 {
00073
         FIFR \&= \sim (1 << INTF0);
                                    // Clear the interrupt flag for INTO
00074 }
00075
00076 //-
00077 void KernelSWI::Trigger(void)
00078 {
00079
          //if(Thread_IsSchedulerEnabled())
08000
00081
              PORTD &= \sim 0 \times 04;
00082
              PORTD |= 0x04;
```

```
00083 }
```

15.9 /home/moslevin/Project/R1/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>
```

15.9.1 Detailed Description

Kernel Timer Implementation for ATMega328p.

Definition in file kerneltimer.cpp.

15.10 kerneltimer.cpp

```
00001 /*========
00002
00003
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]-
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
00031
00032 //---
00033 void KernelTimer::Config(void)
00034 {
00035
          TCCR1B = TCCR1B_INIT;
00037
00038 //--
00039 void KernelTimer::Start(void)
00040 {
00041 #if !KERNEL_TIMERS_TICKLESS
          TCCRIB = ((1 << WGM12) | (1 << CS11) | (1 << CS10));
OCR1A = ((SYSTEM_FREQ / 1000) / 64);
00043
00044 #else
        TCCR1B |= (1 << CS12);
00045
00046 #endif
00047
00048
          TCNT1 = 0;
00049
          TIFR1 &= ~TIMER_IFR;
00050
          TIMSK1 |= TIMER_IMSK;
00051 }
00052
00053 //---
00054 void KernelTimer::Stop(void)
00055 {
```

```
00056 #if KERNEL_TIMERS_TICKLESS
      TIFR1 &= ~TIMER_IFR;
TIMSK1 &= ~TIMER_IMSK;
00058
         TCCR1B &= ~ (1 << CS12);
                                     // Disable count...
00059
         TCNT1 = 0;
00060
        OCR1A = 0;
00061
00062 #endif
00063 }
00064
00065 //----
00066 K_USHORT KernelTimer::Read(void)
00067 {
00068 #if KERNEL_TIMERS_TICKLESS
       volatile K_USHORT usRead1;
00069
00070
         volatile K_USHORT usRead2;
00071
00072
         do {
         usRead1 = TCNT1;
usRead2 = TCNT1;
00073
00075
         } while (usRead1 != usRead2);
00076
00077
         return usRead1;
00078 #else
00079
       return 0;
00080 #endif
00081 }
00082
00083 //----
00084 K_ULONG KernelTimer::SubtractExpiry(K_ULONG ulInterval_)
00085 {
00086 #if KERNEL_TIMERS_TICKLESS
00087 OCR1A -= (K_USHORT)ulInterval_;
00088
          return (K_ULONG) OCR1A;
00089 #else
00090
         return 0;
00091 #endif
00092 }
00094 //-
00095 K_ULONG KernelTimer::TimeToExpiry(void)
00096 (
00097 #if KERNEL_TIMERS_TICKLESS
         K_USHORT usRead = KernelTimer::Read();
00098
         K_USHORT usOCR1A = OCR1A;
00099
00100
        if (usRead >= usOCR1A)
00101
00102
        {
00103
              return 0;
         }
00104
00105
         else
00106
        {
00107
              return (K_ULONG) (usOCR1A - usRead);
00108
00109 #else
00110
        return 0:
00111 #endif
00112 }
00113
00114 //---
00115 K_ULONG KernelTimer::GetOvertime(void)
00116 {
00117
         return KernelTimer::Read();
00118 }
00119
00120 //---
00121 K_ULONG KernelTimer::SetExpiry(K_ULONG ulInterval_)
00122 {
00123 #if KERNEL_TIMERS_TICKLESS
00124 K_USHORT usSetInterval;
00125
          if (ulInterval_ > 65535)
00126
              usSetInterval = 65535;
00127
00128
00129
         else
00130
        {
00131
              usSetInterval = (K_USHORT)ulInterval_;
00132
00133
         OCR1A = usSetInterval;
00134
         return (K_ULONG)usSetInterval;
00135
00136 #else
00137
         return 0;
00138 #endif
00139 }
00140
00141 //----
00142 void KernelTimer::ClearExpiry(void)
```

```
00143 {
00144 #if KERNEL_TIMERS_TICKLESS
                                                     // Clear the compare value
00145
           OCR1A = 65535;
00146 #endif
00147 }
00148
00149 //----
00150 K_UCHAR KernelTimer::DI(void)
00151 {
00152 #if KERNEL_TIMERS_TICKLESS
00153 K_BOOL bEnabled = ((TIMSK1 & (TIMER_IMSK)) != 0);

00154 TIFR1 &= ~TIMER_IFF; // Clear interrupt flags

00155 TIMSK1 &= ~TIMER_IMSK; // Disable interrupt

00156 return bEnabled;
00157 #else
00158
           return 0;
00159 #endif
00160 }
00161
00162 //--
00163 void KernelTimer::EI(void)
00164 {
           KernelTimer::RI(0);
00165
00166 }
00167
00168 //---
00169 void KernelTimer::RI(K_BOOL bEnable_)
00170 {
00171 #if KERNEL_TIMERS_TICKLESS
00172 if (bEnable_)
00173 {
                TIMSK1 |= (1 << OCIE1A);  // Enable interrupt</pre>
00174
        }
else
{
00175
00176
00177
                TIMSK1 &= \sim (1 << OCIE1A);
00178
00179
00180 #endif
00181 }
```

/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h 15.11 File Reference

Profiling timer hardware interface.

```
#include "kerneltypes.h"
#include "mark3cfq.h"
#include "ll.h"
```

Classes

· class Profiler

System profiling timer interface.

15.11.1 Detailed Description

Profiling timer hardware interface.

Definition in file kernelprofile.h.

15.12 kernelprofile.h

```
00001 /*----
00002
00003
00004
```

```
80000
00009 -- [Mark3 Realtime Platform] ------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022 #include "11.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
00032
00033 //----
00037 class Profiler
00038 {
00039 public:
       static void Init();
00047
00053
        static void Start();
00054
00060
        static void Stop();
00061
00067
         static K_USHORT Read();
00068
00072
         static void Process();
00073
         static K_ULONG GetEpoch() { return m_ulEpoch; }
00077
00078 private:
08000
         static K_ULONG m_ulEpoch;
00081 };
00082
00083 #endif //KERNEL USE PROFILER
00084
00085 #endif
00086
```

15.13 /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h File Reference

Kernel Software interrupt declarations.

```
#include "kerneltypes.h"
```

Classes

· class KernelSWI

Class providing the software-interrupt required for context-switching in the kernel.

15.13.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

15.14 kernelswi.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 ====
00023 #include "kerneltypes.h"
00024 #ifndef __KERNELSWI_H_
00025 #define __KERNELSWI_H_
00026
00027 //---
00032 class KernelSWI
00033 {
00034 public:
00041
          static void Config(void);
00042
00048
          static void Start (void);
00049
00055
          static void Stop (void);
00056
00062
          static void Clear (void);
00063
00069
          static void Trigger (void);
00070
00078
          static K UCHAR DI();
00079
00087
          static void RI(K_BOOL bEnable_);
00088 };
00089
00090
00091 #endif // __KERNELSIW_H_
```

15.15 /home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h File Reference

Kernel Timer Class declaration.

```
#include "kerneltypes.h"
```

Classes

class KernelTimer

Hardware timer interface, used by all scheduling/timer subsystems.

15.15.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

15.16 kerneltimer.h



```
00012 See license.txt for more information
00021 #include "kerneltypes.h"
00022 #ifndef __KERNELTIMER_H_
00023 #define ___KERNELTIMER_H_
00024
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 K_UCHAR DI (void);
00063
        static void RI(K_BOOL bEnable_);
00071
00072
00078
        static void EI (void);
00079
00090
        static K_ULONG SubtractExpiry(K_ULONG ulInterval_);
00091
00100
        static K_ULONG TimeToExpiry(void);
00101
        static K_ULONG SetExpiry(K_ULONG ulInterval_);
00110
00111
00120
        static K_ULONG GetOvertime(void);
00121
00127
        static void ClearExpiry(void);
00128
00129 private:
00137
        static K_USHORT Read(void);
00138
00139 };
00140
00141 #endif //__KERNELTIMER_H_
```

15.17 /home/moslevin/Project/R1/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) (K_UCHAR*) (((K_USHORT)x) + (y-1))

15.18 threadport.h

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.

15.17.1 Detailed Description

ATMega328p Multithreading support.

Definition in file threadport.h.

15.17.2 Macro Definition Documentation

```
15.17.2.1 #define CS_ENTER( )
```

Value:

```
{ \
volatile K_UCHAR x; \
x = _SFR_IO8(SR_); \
ASM("cli");
```

These macros must be used in pairs!

Enter critical section (copy status register, disable interrupts)

Definition at line 142 of file threadport.h.

15.18 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);
```

```
0x3F
0x3E
0x3D
00033 #define SR_
00035 #define SPH_
00037 #define SPL_
00038
00039
00040 //----
00042 #define TOP_OF_STACK(x, y) (K_UCHAR*) ( ((K_USHORT)x) + (y-1) )
00043 #define PUSH_TO_STACK(x, y)
                                               *x = y; x--;
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_pstCurrent");
00086 ASM("lds r27, g_pstCurrent + 1"); \
00087 ASM("adiw r26, 4"); \
00088 ASM("in r0, 0x3D"); \
00089 ASM("in r0, 0x3D"); \
00090 ASM("in r0, 0x3E"); \
00091 ASM("in x+, r0");
00092
00093 //-
00095 #define Thread_RestoreContext() \
00096 ASM("lds r26, g_pstCurrent"); \
00097 ASM("lds r27, g_pstCurrent + 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 //----
00142 #define CS_ENTER()
00143 {
00144 volatile K_UCHAR x; \
00145 x = _SFR_IO8(SR_); \
00146 ASM("cli");
00149 #define CS_EXIT() \
00150 _SFR_IO8(SR_) = x;\
00151 }
00152
00153 //--
00155 #define ENABLE_INTS() ASM("sei");
00156 #define DISABLE_INTS() ASM("cli");
00156 #define DISABLE_INTS()
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_
```

15.19 /home/moslevin/Project/R1/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 SWI using INT0 - used to trigger a context switch.
- ISR (TIMER1_COMPA_vect)

Timer interrupt ISR - causes a tick, which may cause a context switch.

15.19.1 Detailed Description

ATMega328p Multithreading.

Definition in file threadport.cpp.

15.20 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"
00027 #include "kernercimer.
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_pstCurrentThread;
00037
00039 void ThreadPort::InitStack(Thread *pclThread_)
00040 {
          // Initialize the stack for a Thread K\_USHORT usAddr;
00041
00042
00043
          K_UCHAR *pucStack;
00044
          K_USHORT i;
00045
00046
           // Get the address of the thread's entry function
00047
          usAddr = (K_USHORT) (pclThread_->m_pfEntryPoint);
00048
00049
          // Start by finding the bottom of the stack
          pucStack = (K_UCHAR*)pclThread_->m_pwStackTop;
00050
00051
00052
          // clear the stack, and initialize it to a known-default value (easier
          \ensuremath{//} to debug when things go sour with stack corruption or overflow)
00053
00054
          for (i = 0; i < pclThread_->m_usStackSize; i++)
00055
00056
              pclThread_->m_pwStack[i] = 0xFF;
00057
00058
00059
           // Our context starts with the entry function
00060
          {\tt PUSH\_TO\_STACK} \, ({\tt pucStack, \ (K\_UCHAR)} \, ({\tt usAddr \& 0x00FF})) \, ; \\
          PUSH_TO_STACK(pucStack, (K_UCHAR)((usAddr >> 8) & 0x00FF));
00061
00062
00063
00064
          PUSH_TO_STACK(pucStack, 0x00);
                                            // R0
00065
00066
           // Push status register and R1 (which is used as a constant zero)
          PUSH_TO_STACK(pucStack, 0x80); // SR
00067
          PUSH_TO_STACK(pucStack, 0x00);
00068
00069
00070
           // Push other registers
00071
           for (i = 2; i <= 23; i++) //R2-R23
00072
00073
              PUSH_TO_STACK (pucStack, i);
00074
00075
00076
           // Assume that the argument is the only stack variable
00077
          PUSH_TO_STACK(pucStack, (K_UCHAR)(((K_USHORT)(pclThread_->
      //R24
00078
00079
00080
          // Push the rest of the registers in the context
```

15.20 threadport.cpp 123

```
00081
         for (i = 26; i <=31; i++)</pre>
00082
00083
            PUSH_TO_STACK(pucStack, i);
00084
00085
00086
         // Set the top o' the stack.
         pclThread_->m_pwStackTop = (K_UCHAR*)pucStack;
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
       // If there's no next-thread-to-run...
if (g_pstNext == Kernel::GetIdleThread())
00096
00097
00098
00099
             g_pstCurrent = Kernel::GetIdleThread();
00100
00101
             // Disable the SWI, and re-enable interrupts -- enter nested interrupt
             // mode.
00102
             KernelSWI::DI();
00103
00104
00105
             K_UCHAR ucsr = _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 (q_pstNext == 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" );
                Kernel::IdleFunc():
00116
               ASM( "cli" );
00117
00118
00119
00120
             // Progress has been achieved -- an interrupt-triggered event has caused
             // the scheduler to run, and choose a new thread. Since we've already // saved the context of the thread we've hijacked to run idle, we can
00121
00122
             \ensuremath{//} proceed to disable the nested interrupt context and switch to the
00123
             // new thread.
00124
00125
00126
             \_SFR\_IO8(SR\_) = ucSR;
00127
             KernelSWI::RI( true );
00128
00129 #endif
00130
        g_pstCurrent = (Thread*)g_pstNext;
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
                                              // run the scheduler - determine the first
00141
         Scheduler::Schedule();
      thread to run
00142
00143
         Thread_Switch();
                                             // Set the next scheduled thread to the current thread
00144
00145
         KernelTimer::Start();
                                              \ensuremath{//} enable the kernel timer
                                              // enable the task switch SWI
00146
         KernelSWI::Start();
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 //-----
00158 //---
00159 ISR(INTO_vect) __attribute__ ( ( signal, naked ) );
00160 ISR(INTO_vect)
00161 {
                                   // Push the context (registers) of the current task
00162
         Thread_SaveContext();
                                    // Switch to the next task
00163
         Thread Switch();
                                   // Pop the context (registers) of the next task
00164
         Thread RestoreContext();
                                    // Return to the next task
00165
         ASM("reti");
00166 }
00167
00168 //-----
00173 //-----
00174 ISR(TIMER1_COMPA_vect)
```

15.21 /home/moslevin/Project/R1/kernel/driver.cpp File Reference

Device driver/hardware abstraction layer.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kerneldebug.h"
#include "driver.h"
```

Classes

class DevNull

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

Macros

```
    #define __FILE_ID__ DRIVER_CPP
    File ID used in kernel trace calls.
```

Functions

```
    static K_UCHAR DrvCmp (const K_CHAR *szStr1_, const K_CHAR *szStr2_)
    DrvCmp.
```

Variables

· static DevNull clDevNull

Default driver included to allow for run-time "stubbing".

15.21.1 Detailed Description

Device driver/hardware abstraction layer.

Definition in file driver.cpp.

15.21.2 Function Documentation

```
15.21.2.1 static K UCHAR DrvCmp (const K CHAR * szStr1_, const K CHAR * szStr2_) [static]
```

DrvCmp.

String comparison function used to compare input driver name against a known driver name in the existing driver

15.22 driver.cpp 125

Parameters

szStr1_	user-specified driver name
szStr2_	name of a driver, provided from the driver table

Returns

1 on match, 0 on no-match

Definition at line 75 of file driver.cpp.

15.22 driver.cpp

```
00001 /
00002
00003
00004
                    \Box
00005
00006
00007
80000
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 "kerneldebug.h"
00024 #include "driver.h"
00025
00026 //---
00027 #if defined __FILE_ID__
00028
          #undef __FILE_ID__
00029 #endif
00030 #define __FILE_ID__
                               DRIVER_CPP
00031
00032 //----
00033 #if KERNEL_USE_DRIVER
00034
00035 DoubleLinkList DriverList::m_clDriverList;
00036
00040 class DevNull : public Driver
00041 {
00042 public:
00043
          virtual void Init() { SetName("/dev/null"); };
          virtual K_UCHAR Open() { return 0; }
virtual K_UCHAR Close() { return 0; }
00044
00045
00046
00047
          virtual K_USHORT Read( K_USHORT usBytes_,
00048
          K_UCHAR *pucData_) { return 0; }
00049
00050
          virtual K_USHORT Write( K_USHORT usBytes_,
00051
          K_UCHAR *pucData_) { return 0; }
00052
00053
          virtual K_USHORT Control( K_USHORT usEvent_,
00054
              void *pvDataIn_,
               K_USHORT usSizeIn_,
00055
00056
               void *pvDataOut_,
00057
              K_USHORT usSizeOut_ ) { return 0; }
00058
00059 };
00060
00061 //----
00062 static DevNull clDevNull;
00063
00064 //--
00075 static K_UCHAR DrvCmp( const K_CHAR *szStr1_, const K_CHAR *szStr2_ )
00076 {
00077
          K_CHAR *szTmp1 = (K_CHAR*) szStr1_;
K_CHAR *szTmp2 = (K_CHAR*) szStr2_;
00078
00079
08000
          while (*szTmp1 && *szTmp2)
00081
00082
               if (*szTmp1++ != *szTmp2++)
00083
               {
00084
                   return 0:
00085
00086
          }
```

```
00088
          // Both terminate at the same length
00089
          if (!(*szTmp1) && !(*szTmp2))
00090
00091
              return 1;
00092
          }
00093
00094
          return 0;
00095 }
00096
00097 //----
00098 void DriverList::Init()
00099 {
00100
          // Ensure we always have at least one entry - a default in case no match
00101
          // is found (/dev/null)
00102
         clDevNull.Init();
00103
         Add(&clDevNull);
00104 }
00107 Driver *DriverList::FindByPath( const K_CHAR *m_pcPath )
00108 {
00109
          KERNEL_ASSERT( m_pcPath );
     Driver *pclTemp = static_cast<Driver*>(m_clDriverList.
GetHead());
00110
00111
00112
          // Iterate through the list of drivers until we find a match, or we
00113
          \ensuremath{//} exhaust our list of installed drivers
00114
          while (pclTemp)
00115
00116
              if (DrvCmp (m_pcPath, pclTemp->GetPath()))
00117
              {
00118
                  return pclTemp;
00119
             pclTemp = static_cast<Driver*>(pclTemp->GetNext());
00120
00121
00122
          // No matching driver found - return a pointer to our /dev/null driver
          return &clDevNull;
00124 }
00125
00126 #endif
```

15.23 /home/moslevin/Project/R1/kernel/eventflag.cpp File Reference

Event Flag Blocking Object/IPC-Object implementation.

```
#include "mark3cfg.h"
#include "blocking.h"
#include "kernel.h"
#include "thread.h"
#include "eventflag.h"
#include "kernelaware.h"
#include "timerlist.h"
```

Functions

void TimedEventFlag_Callback (Thread *pclOwner_, void *pvData_)
 TimedEventFlag_Callback.

15.23.1 Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

15.23.2 Function Documentation

15.24 eventflag.cpp 127

15.23.2.1 void TimedEventFlag_Callback (Thread * pclOwner_, void * pvData_)

TimedEventFlag_Callback.

This function is called whenever a timed event flag wait operation fails in the time provided. This function wakes the thread for which the timeout was requested on the blocking call, sets the thread's expiry flags, and reschedules if necessary.

Parameters

pclOwner_	Thread to wake
pvData_	Pointer to the event-flag object

Definition at line 42 of file eventflag.cpp.

15.24 eventflag.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 "mark3cfg.h"
00020 #include "blocking.h"
00021 #include "kernel.h"
00022 #include "thread.h"
00023 #include "eventflag.h"
00024 #include "kernelaware.h"
00025
00026 #if KERNEL_USE_EVENTFLAG
00027
00028 #if KERNEL_USE_TIMEOUTS
00029 #include "timerlist.h'
00030 //--
00042 void TimedEventFlag_Callback(Thread *pclOwner_, void *pvData_)
00043 {
00044
          EventFlag *pclEventFlag = static_cast<EventFlag*>(pvData_);
00045
00046
          pclEventFlag->WakeMe(pclOwner_);
00047
          pclOwner_->SetExpired(true);
00048
          pclOwner_->SetEventFlagMask(0);
00049
00050
          if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      ()->GetCurPriority())
00051
          {
00052
              Thread::Yield();
00053
00054 }
00055
00056 //-
00057 void EventFlag::WakeMe(Thread *pclChosenOne_)
00058 {
00059
          UnBlock(pclChosenOne_);
00060 }
00061 #endif
00062
00063 //-
00064 #if KERNEL_USE_TIMEOUTS
          K_USHORT EventFlag::Wait_i(K_USHORT usMask_,
      EventFlagOperation_t eMode_, K_ULONG ulTimeMS_)
00066 #else
         K_USHORT EventFlag::Wait_i(K_USHORT usMask_,
00067
      EventFlagOperation_t eMode_)
00068 #endif
00069 {
00070
          K_BOOL bThreadYield = false;
00071
          K_BOOL bMatch = false;
00072
00073 #if KERNEL_USE_TIMEOUTS
00074
          Timer clEventTimer;
00075
          K_BOOL bUseTimer = false;
```

```
00076 #endif
00077
00078
          // Ensure we're operating in a critical section while we determine
00079
          \ensuremath{//} whether or not we need to block the current thread on this object.
00080
          CS ENTER();
00081
          \ensuremath{//} Check to see whether or not the current mask matches any of the
00083
          // desired bits.
00084
          g_pstCurrent->SetEventFlagMask(usMask_);
00085
     if ((eMode_ == EVENT_FLAG_ALL) || (eMode_ ==
EVENT_FLAG_ALL_CLEAR))
00086
00087
00088
              // Check to see if the flags in their current state match all of
00089
              // the set flags in the event flag group, with this mask.
00090
              if ((m_usSetMask & usMask_) == usMask_)
00091
              {
00092
                  bMatch = true;
                  g_pstCurrent->SetEventFlagMask(usMask_);
00093
00094
              }
00095
00096
          else if ((eMode_ == EVENT_FLAG_ANY) || (eMode_ ==
     EVENT_FLAG_ANY_CLEAR))
00097
00098
              // Check to see if the existing flags match any of the set flags in
00099
              // the event flag group with this mask
              if (m_usSetMask & usMask_)
00100
00101
              {
00102
                  bMatch = true;
                  g_pstCurrent->SetEventFlagMask(m_usSetMask & usMask_);
00103
00104
              }
00105
         }
00106
00107
          // We're unable to match this pattern as-is, so we must block.
00108
          if (!bMatch)
00109
              // Reset the current thread's event flag mask & mode
00110
              g_pstCurrent->SetEventFlagMask(usMask_);
00111
00112
              g_pstCurrent->SetEventFlagMode(eMode_);
00113
00114 #if KERNEL_USE_TIMEOUTS
             if (ulTimeMS_)
00115
00116
              {
00117
                  g_pstCurrent->SetExpired(false);
00118
                  clEventTimer.Init();
00119
                  clEventTimer.Start(0, ulTimeMS_, TimedEventFlag_Callback, (void*)
     this);
00120
                  bUseTimer = true;
00121
              }
00122 #endif
00124
              // Add the thread to the object's block-list.
00125
              Block(g_pstCurrent);
00126
              // Trigger that
00127
00128
              bThreadYield = true;
00129
         }
00130
00131
          // If bThreadYield is set, it means that we've blocked the current thread,
00132
          \ensuremath{//} and must therefore rerun the scheduler to determine what thread to
          // switch to.
00133
00134
          if (bThreadYield)
00135
          {
00136
              // Switch threads immediately
00137
              Thread::Yield();
00138
          }
00139
00140
          // Exit the critical section and return back to normal execution
00141
          CS_EXIT();
00142
00147 #if KERNEL_USE_TIMEOUTS
00148
          if (bUseTimer && bThreadYield)
00149
00150
              clEventTimer.Stop();
00151
00152 #endif
00153
00154
          return g_pstCurrent->GetEventFlagMask();
00155 }
00156
00157 //
00158 K_USHORT EventFlag::Wait(K_USHORT usMask_,
      EventFlagOperation_t eMode_)
00159 (
00160 #if KERNEL_USE_TIMEOUTS
00161
        return Wait_i(usMask_, eMode_, 0);
00162 #else
```

15.24 eventflag.cpp 129

```
return Wait_i(usMask_, eMode_);
00164 #endif
00165 }
00166
00167 #if KERNEL USE TIMEOUTS
00168 //--
00169 K_USHORT EventFlag::Wait(K_USHORT usMask_,
      EventFlagOperation_t eMode_, K_ULONG ulTimeMS_)
00170 {
00171
           return Wait_i(usMask_, eMode_, ulTimeMS_);
00172 }
00173 #endif
00174
00175 //--
00176 void EventFlag::Set(K_USHORT usMask_)
00177 {
00178
           Thread *pclPrev;
00179
           Thread *pclCurrent;
           K_BOOL bReschedule = false;
00180
00181
           K_USHORT usNewMask;
00182
00183
           CS_ENTER();
00184
           // Walk through the whole block list, checking to see whether or not // the current flag set now matches any/all of the masks and modes of
00185
00186
           // the threads involved.
00187
00188
00189
           m_usSetMask |= usMask_;
00190
           usNewMask = m_usSetMask;
00191
           // Start at the head of the list, and iterate through until we hit the // "head" element in the list again. Ensure that we handle the case where
00192
00193
00194
           // we remove the first or last elements in the list, or if there's only
00195
           // one element in the list.
00196
           pclCurrent = static_cast<Thread*>(m_clBlockList.GetHead());
00197
00198
           // Do nothing when there are no objects blocking.
00199
           if (pclCurrent)
00200
           {
00201
                // First loop - process every thread in the block-list and check to
00202
                // see whether or not the current flags match the event-flag conditions // on the thread.
00203
00204
00205
                {
                    pclPrev = pclCurrent;
00206
00207
                    pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00208
00209
                    // Read the thread's event mask/mode
                    K_USHORT usThreadMask = pclPrev->GetEventFlagMask();
00210
                    EventFlagOperation_t eThreadMode = pclPrev->
00211
      GetEventFlagMode();
00212
00213
                     // For the "any" mode - unblock the blocked threads if one or more bits
                    // in the thread's bitmask match the object's bitmask
if ((EVENT_FLAG_ANY == eThreadMode) || (
00214
00215
      EVENT_FLAG_ANY_CLEAR == eThreadMode))
00216
                    {
00217
                         if (usThreadMask & m_usSetMask)
00218
                         {
                              pclPrev->SetEventFlagMode(
00219
      EVENT_FLAG_PENDING_UNBLOCK);
                             pclPrev->SetEventFlagMask(m_usSetMask & usThreadMask);
00220
00221
                             bReschedule = true;
00222
00223
                              // If the "clear" variant is set, then clear the bits in the mask
00224
                              \ensuremath{//} that caused the thread to unblock.
00225
                              if (EVENT_FLAG_ANY_CLEAR == eThreadMode)
00226
                              {
00227
                                  usNewMask &=~ (usThreadMask & usMask_);
00228
                             }
00229
                         }
00230
                    // For the "all" mode, every set bit in the thread's requested bitmask must
// match the object's flag mask.
else if ((EVENT_FLAG_ALL == eThreadMode) || (
00231
00232
00233
      EVENT_FLAG_ALL_CLEAR == eThreadMode))
00234
                    {
00235
                         if ((usThreadMask & m_usSetMask) == usThreadMask)
00236
                              pclPrev->SetEventFlagMode(
00237
      EVENT_FLAG_PENDING_UNBLOCK);
00238
                             pclPrev->SetEventFlagMask(usThreadMask);
00239
                              bReschedule = true;
00240
00241
                              // If the "clear" variant is set, then clear the bits in the mask
                              // that caused the thread to unblock.
if (EVENT_FLAG_ALL_CLEAR == eThreadMode)
00242
00243
```

```
{
00245
                               usNewMask &=~ (usThreadMask & usMask_);
00246
00247
                       }
00248
                  }
00249
              ^{\prime} // To keep looping, ensure that there's something in the list, and
00251
              // that the next item isn't the head of the list.
00252
              while (pclPrev != m_clBlockList.GetTail());
00253
00254
              // Second loop - go through and unblock all of the threads that
              // were tagged for unblocking.
00255
00256
              pclCurrent = static_cast<Thread*>(m_clBlockList.
      GetHead());
00257
              K_BOOL bIsTail = false;
00258
00259
              {
00260
                   pclPrev = pclCurrent;
00261
                  pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00262
00263
                   // Check to see if this is the condition to terminate the loop
00264
                   if (pclPrev == m_clBlockList.GetTail())
00265
00266
                      bIsTail = true:
00267
                  }
00268
00269
                   \ensuremath{//} If the first pass indicated that this thread should be
00270
                   \ensuremath{//} unblocked, then unblock the thread
00271
                   if (pclPrev->GetEventFlagMode() ==
      EVENT_FLAG_PENDING_UNBLOCK)
00272
                  {
00273
                       UnBlock (pclPrev);
00274
00275
00276
              while (!bIsTail);
00277
         }
00278
          // If we awoke any threads, re-run the scheduler
00280
          if (bReschedule)
00281
00282
              Thread::Yield();
          }
00283
00284
00285
          // Update the bitmask based on any "clear" operations performed along
          // the way
00286
00287
          m_usSetMask = usNewMask;
00288
          \ensuremath{//} Restore interrupts - will potentially cause a context switch if a
00289
          // thread is unblocked.
00290
00291
          CS_EXIT();
00292 }
00293
00294 //---
00295 void EventFlag::Clear(K_USHORT usMask_)
00296 {
00297
           // Just clear the bitfields in the local object.
00298
          CS_ENTER();
00299
          m_usSetMask &= ~usMask_;
00300
          CS_EXIT();
00301 }
00302
00303 //-
00304 K_USHORT EventFlag::GetMask()
00305 {
00306
           // Return the presently held event flag values in this object. Ensure
00307
          \ensuremath{//} we get this within a critical section to guarantee atomicity.
00308
          K USHORT usReturn;
          CS_ENTER();
00309
00310
          usReturn = m usSetMask:
00311
          CS_EXIT();
00312
          return usReturn;
00313 }
00314
00315 #endif // KERNEL USE EVENTFLAG
```

15.25 /home/moslevin/Project/R1/kernel/kernel.cpp File Reference

Kernel initialization and startup code.

15.26 kernel.cpp 131

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernel.h"
#include "scheduler.h"
#include "thread.h"
#include "threadport.h"
#include "timerlist.h"
#include "driver.h"
#include "driver.h"
#include "kernelprofile.h"
#include "tracebuffer.h"
#include "kerneldebug.h"
```

Macros

#define __FILE_ID__ KERNEL_CPP
 File ID used in kernel trace calls.

15.25.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

15.26 kernel.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
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"
00031 #include "profile.h"
00032 #include "kernelprofile.h"
00033 #include "tracebuffer.h"
00034 #include "kerneldebug.h'
00035
00036 K_BOOL Kernel::m_bIsStarted;
00037 K_BOOL Kernel::m_bIsPanic;
00038 panic_func_t Kernel::m_pfPanic;
00039
00040 #if KERNEL_USE_IDLE_FUNC
00041 idle_func_t Kernel::m_pfIdle;
00042 FakeThread_t Kernel::m_clIdle;
00043 #endif
00044
00045 //
00046 #if defined __FILE_ID_
00047 #undef __FILE_ID_
00048 #endif
```

```
00049 #define __FILE_ID__
                             KERNEL_CPP
00051 //----
00052 void Kernel::Init(void)
00053 {
00054
         m bIsStarted = false;
         m_bIsPanic = false;
00056
        m_pfPanic = 0;
00057
00058 #if KERNEL_USE_IDLE_FUNC
00059 ((Thread*)&m_clIdle)->InitIdle();
         m_pfIdle = 0;
00060
00061 #endif
00062
00063 #if KERNEL_USE_DEBUG
00064
         TraceBuffer::Init();
00065 #endif
       KERNEL_TRACE( STR_MARK3_INIT );
00066
       // Initialize the global kernel data - scheduler, timer-scheduler, and
// the global message pool.
00067
00068
00069
00070
         Scheduler::Init();
00071 #if KERNEL_USE_DRIVER
00072
       DriverList::Init();
00073 #endif
00074 #if KERNEL_USE_TIMERS
00075
         TimerScheduler::Init();
00076 #endif
00077 #if KERNEL_USE_MESSAGE
00078
         GlobalMessagePool::Init();
00079 #endif
00080 #if KERNEL_USE_PROFILER
00081
         Profiler::Init();
00082 #endif
00083 }
00084
00085 //--
00086 void Kernel::Start(void)
00087 {
00088
         KERNEL_TRACE( STR_THREAD_START );
00089
         m_bIsStarted = true;
         ThreadPort::StartThreads();
00090
00091
         KERNEL_TRACE( STR_START_ERROR );
00092 }
00093
00094 //--
00095 void Kernel::Panic(K_USHORT usCause_)
00096 {
00097
         m bIsPanic = true;
00098
         if (m_pfPanic)
00099
         {
00100
             m_pfPanic(usCause_);
        }
00101
00102
       else
00103
00104 #if KERNEL_AWARE_SIMULATION
             KernelAware::ExitSimulator();
00106 #endif
00107
             while(1);
00108
         }
00109 }
```

15.27 /home/moslevin/Project/R1/kernel/kernelaware.cpp File Reference

Kernel aware simulation support.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernelaware.h"
#include "threadport.h"
```

Classes

• union KernelAwareData_t

This structure is used to communicate between the kernel and a kernel- aware host.

15.28 kernelaware.cpp 133

Variables

• volatile K_BOOL g_blsKernelAware = false

Will be set to true by a kernel-aware host.

volatile K_UCHAR g_ucKACommand

Kernel-aware simulator command to execute.

KernelAwareData_t g_stKAData

Data structure used to communicate with host.

15.27.1 Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.cpp.

15.27.2 Variable Documentation

15.27.2.1 volatile K_BOOL g_blsKernelAware = false

Will be set to true by a kernel-aware host.

Definition at line 71 of file kernelaware.cpp.

15.27.2.2 KernelAwareData_t g_stKAData

Data structure used to communicate with host.

Definition at line 73 of file kernelaware.cpp.

15.28 kernelaware.cpp

```
00001
00002
00003
00004
00005
00006
00007
80000
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 #if KERNEL_AWARE_SIMULATION
00027
00028 //----
00037 typedef union
00038 {
          volatile K_USHORT ausBuffer[5];
00040
00044
          struct
00045
          {
00046
              volatile const K CHAR *szName;
00047
          } Profiler;
00052
          struct
00053
          {
00054
              volatile K_USHORT usFile;
00055
              volatile K_USHORT usLine;
volatile K_USHORT usCode;
00056
00057
              volatile K_USHORT usArg1;
00058
              volatile K_USHORT usArg2;
```

```
00059
          } Trace;
00064
         struct
00065
00066
             volatile const K_CHAR *szString;
          } Print;
00067
00068 } KernelAwareData_t;
00070 //---
                        g_bIskerne...
g_ucKACommand;
g_stKAData;
00071 volatile K_BOOL
                                g_bIsKernelAware = false;
00072 volatile K_UCHAR
00073 KernelAwareData_t
00074
00075
00076 //-----
00077 void KernelAware::ProfileInit(const K_CHAR *szStr_)
00078 {
          CS_ENTER();
00079
08000
          g_stKAData.Profiler.szName = szStr_;
          g_ucKACommand = KA_COMMAND_PROFILE_INIT;
00081
00082
         CS_EXIT();
00083 }
00084
00085 //---
00086 void KernelAware::ProfileStart(void)
00087 {
00088
          g_uckAcommand = KA_COMMAND_PROFILE_START;
00089 }
00090
00091 //--
00092 void KernelAware::ProfileStop(void)
00093 {
00094
          g_ucKACommand = KA_COMMAND_PROFILE_STOP;
00095 }
00096
00097 //---
00098 void KernelAware::ProfileReport (void)
00099 {
          g_ucKACommand = KA_COMMAND_PROFILE_REPORT;
00101 }
00102
00103 //---
00104 void KernelAware::ExitSimulator(void)
00105 {
00106
          g_ucKACommand = KA_COMMAND_EXIT_SIMULATOR;
00107 }
00108
00109 //----
00110 void KernelAware::Trace( K_USHORT usFile_,
00111
                                 K_USHORT usLine_,
00112
                                 K_USHORT usCode_ )
00113 {
00114
          Trace_i( usFile_, usLine_, usCode_, 0, 0, KA_COMMAND_TRACE_0 );
00115 }
00116
00117 //----
00118 void KernelAware::Trace( K_USHORT usFile_,
00119
                                 K_USHORT usLine_,
00120
                                 K_USHORT usCode_,
00121
                                 K_USHORT usArg1_)
00122 {
00123
          Trace_i( usFile_, usLine_, usCode_, usArg1_, 0 ,KA_COMMAND_TRACE_1 );
00124
00125 }
00126 //--
00127 void KernelAware::Trace( K_USHORT usFile_,
00128
                                 K_USHORT usLine_,
                                 K_USHORT usCode_,
00129
                                 K_USHORT usArg1_,
00130
00131
                                 K_USHORT usArg2_)
          Trace_i( usFile_, usLine_, usCode_, usArg1_, usArg2_,
00133
      KA_COMMAND_TRACE_2 );
00134 }
00135
00136 //---
00137 void KernelAware::Trace_i( K_USHORT usFile_,
00138
                                 K_USHORT usLine_,
00139
                                 K_USHORT usCode_,
                                 K_USHORT usArg1_,
00140
                                 K_USHORT usArg2_,
00141
00142
                                 KernelAwareCommand_t eCmd_ )
00143 {
          CS_ENTER();
00144
00145
          g_stKAData.Trace.usFile = usFile_;
          g_stKAData.Trace.usLine = usLine_;
g_stKAData.Trace.usCode = usCode_;
00146
00147
00148
         g_stKAData.Trace.usArg1 = usArg1_;
```

```
g_stKAData.Trace.usArg2 = usArg2_;
00150
         g_uckACommand = eCmd_;
00151
         CS_EXIT();
00152 }
00153
00154 //--
00155 void KernelAware::Print(const K_CHAR *szStr_)
00156 {
00157
         CS_ENTER();
00158
         g_stKAData.Print.szString = szStr_;
         g_ucKACommand = KA_COMMAND_PRINT;
00159
        CS_EXIT();
00160
00161 }
00163 //---
00164 K_BOOL KernelAware::IsSimulatorAware(void)
00165 {
00166
         return g_bIsKernelAware;
00167 }
00168
00169 #endif
```

15.29 /home/moslevin/Project/R1/kernel/ksemaphore.cpp File Reference

Semaphore Blocking-Object Implemenation.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "blocking.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

Macros

• #define __FILE_ID__ SEMAPHORE_CPP

File ID used in kernel trace calls.

Functions

void TimedSemaphore_Callback (Thread *pclOwner_, void *pvData_)
 TimedSemaphore_Callback.

15.29.1 Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

15.29.2 Function Documentation

```
15.29.2.1 void TimedSemaphore_Callback ( Thread * pclOwner_, void * pvData_ )
```

TimedSemaphore_Callback.

This function is called from the timer-expired context to trigger a timeout on this semphore. This results in the waking of the thread that generated the semaphore pend call that was not completed in time.

Parameters

pclOwner_	Pointer to the thread to wake
pvData_	Pointer to the semaphore object that the thread is blocked on

Definition at line 50 of file ksemaphore.cpp.

15.30 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 #include "kerneldebug.h"
00028 //----
00029 #if defined __FILE_ID_
         #undef __FILE_ID_
00030
00031 #endif
00032 #define __FILE_ID__
                               SEMAPHORE_CPP
00033
00034 #if KERNEL USE SEMAPHORE
00035
00036 #if KERNEL_USE_TIMEOUTS
00037 #include "timerlist.h"
00038
00039 //---
00050 void TimedSemaphore_Callback(Thread *pclOwner_, void *pvData_)
00051 {
00052
          Semaphore *pclSemaphore = static cast<Semaphore*>(pvData );
00053
00054
          // Indicate that the semaphore has expired on the thread
00055
          pclOwner_->SetExpired(true);
00056
00057
          // Wake up the thread that was blocked on this semaphore.
00058
          pclSemaphore->WakeMe (pclOwner_);
00059
00060
00061
          if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      () ->GetCurPriority())
00062
         {
00063
              Thread::Yield();
00064
          }
00065 }
00066
00067 //---
00068 void Semaphore::WakeMe(Thread *pclChosenOne_)
00069 {
00070
          // Remove from the semaphore waitlist and back to its ready list.
00071
          UnBlock (pclChosenOne_);
00072 }
00073
00074 #endif // KERNEL_USE_TIMEOUTS
00075
00076 //
00077 K_UCHAR Semaphore::WakeNext()
00078 {
00079
          Thread *pclChosenOne;
08000
00081
          pclChosenOne = m_clBlockList.HighestWaiter();
00082
00083
          // Remove from the semaphore waitlist and back to its ready list.
00084
          UnBlock (pclChosenOne);
00085
00086
          // Call a task switch if higher or equal priority thread
          if (pclChosenOne->GetCurPriority() >=
00087
      Scheduler::GetCurrentThread() ->GetCurPriority())
00088
          {
00089
              return 1;
```

```
00090
00091
          return 0;
00092 }
00093
00094 //---
00095 void Semaphore::Init(K_USHORT usInitVal_, K_USHORT usMaxVal_)
00097
          // Copy the paramters into the object - set the maximum value for this
00098
          \ensuremath{//} semaphore to implement either binary or counting semaphores, and set
          // the initial count. Clear the wait list for this object.
00099
00100
          m_usValue = usInitVal ;
          m_usMaxValue = usMaxVal_;
00101
00102
00103
          m_clBlockList.Init();
00104 }
00105
00106 //--
00107 K BOOL Semaphore::Post()
00108 {
          KERNEL_TRACE_1( STR_SEMAPHORE_POST_1, (K_USHORT)
00109
      g_pstCurrent->GetID() );
00110
00111
          K BOOL bThreadWake = 0;
00112
          K BOOL bBail = false:
00113
          // Increment the semaphore count - we can mess with threads so ensure this
          // is in a critical section. We don't just disable the scheudler since
00114
00115
           // we want to be able to do this from within an interrupt context as well.
00116
          CS_ENTER();
00117
          // If nothing is waiting for the semaphore
00118
00119
          if (m_clBlockList.GetHead() == NULL)
00120
          {
00121
               // Check so see if we've reached the maximum value in the semaphore
00122
               if (m_usValue < m_usMaxValue)</pre>
00123
              {
00124
                   // Increment the count value
00125
                  m usValue++;
00126
00127
              else
00128
00129
                   // Maximum value has been reached, bail out.
00130
                  bBail = true;
00131
00132
00133
          else
00134
00135
              \ensuremath{//} Otherwise, there are threads waiting for the semaphore to be
              // posted, so wake the next one (highest priority goes first).
bThreadWake = WakeNext();
00136
00137
00138
          }
00139
00140
          CS_EXIT();
00141
00142
          // If we weren't able to increment the semaphore count, fail out.
00143
          if (bBail)
00144
          {
00145
              return false:
00146
00147
          \ensuremath{//} if bThreadWake was set, it means that a higher-priority thread was
00148
          /\!/ woken. Trigger a context switch to ensure that this thread gets /\!/ to execute next.
00149
00150
00151
          if (bThreadWake)
00152
          {
00153
              Thread::Yield();
00154
00155
          return true;
00156 }
00157
00159 #if KERNEL_USE_TIMEOUTS
00160 K_BOOL Semaphore::Pend_i( K_ULONG ulWaitTimeMS_ )
00161 #else
00162 void Semaphore::Pend_i( void )
00163 #endif
00164 {
00165
          KERNEL_TRACE_1 ( STR_SEMAPHORE_PEND_1, (K_USHORT)
      g_pstCurrent->GetID() );
00166
00167 #if KERNEL USE TIMEOUTS
          Timer clSemTimer;
00168
00169
          K_BOOL bUseTimer = false;
00170 #endif
00171
00172
           // Once again, messing with thread data - ensure
          // we're doing all of these operations from within a thread-safe context.
00173
00174
          CS_ENTER();
```

```
00176
          // Check to see if we need to take any action based on the semaphore count
00177
          if (m_usValue != 0)
00178
00179
              // The semaphore count is non-zero, we can just decrement the count
00180
              // and go along our merry way.
             m_usValue--;
00181
00182
00183
         else
00184
00185
              // The semaphore count is zero - we need to block the current thread
              // and wait until the semaphore is posted from elsewhere.
00186
00187 #if KERNEL_USE_TIMEOUTS
00188
             if (ulWaitTimeMS_)
00189
00190
                 g_pstCurrent->SetExpired(false);
00191
                 clSemTimer.Init();
                 clSemTimer.Start(0, ulWaitTimeMS_, TimedSemaphore_Callback, (void*)this)
00192
00193
                 bUseTimer = true;
00194
00195 #endif
00196
             Block(g_pstCurrent);
00197
00198
              // Switch Threads immediately
00199
             Thread::Yield();
00200
         }
00201
00202
        CS_EXIT();
00203
00204 #if KERNEL_USE_TIMEOUTS
00205
       if (bUseTimer)
00206
00207
              clSemTimer.Stop();
00208
             return (g_pstCurrent->GetExpired() == 0);
00209
         }
00210
          return true;
00211 #endif
00212 }
00213
00214 //----
00215 // Redirect the untimed pend API to the timed pend, with a null timeout.
00216 void Semaphore::Pend()
00218 #if KERNEL_USE_TIMEOUTS
00219
         Pend_i(0);
00220 #else
00221
       Pend_i();
00222 #endif
00223 }
00225 #if KERNEL_USE_TIMEOUTS
00226 //--
00227 K_BOOL Semaphore::Pend( K_ULONG ulWaitTimeMS_ )
00228 {
00229
         return Pend i( ulWaitTimeMS );
00230 }
00231 #endif
00232
00233 //---
00234 K_USHORT Semaphore::GetCount()
00235 {
00236
         K_USHORT usRet;
00237
         CS_ENTER();
00238
         usRet = m_usValue;
        CS_EXIT();
00239
00240
         return usRet;
00241 }
00242
00243 #endif
```

15.31 /home/moslevin/Project/R1/kernel/II.cpp File Reference

Core Linked-List implementation, from which all kernel objects are derived.

```
#include "kerneltypes.h"
#include "kernel.h"
#include "ll.h"
#include "kerneldebug.h"
```

15.32 Il.cpp 139

Macros

#define __FILE_ID__ LL_CPP

File ID used in kernel trace calls.

15.31.1 Detailed Description

Core Linked-List implementation, from which all kernel objects are derived.

Definition in file II.cpp.

15.32 II.cpp

```
00001 /*----
00003
00004
00005
00006 1
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"
00022 #Include kernelty
00023 #include "kernel.h
00024 #include "11.h"
00025 #include "kerneldebug.h"
00026
00027 //---
00030 #endif
00031 #define __FILE_ID__
                             LL_CPP
00032
00033 //---
00034 void LinkListNode::ClearNode()
00035 {
00036
          next = NULL;
00037
         prev = NULL;
00038 }
00039
00040 //-
00041 void DoubleLinkList::Add(LinkListNode *node_)
00042 {
00043
          KERNEL_ASSERT( node_ );
00044
00045
          \ensuremath{//} Add a node to the end of the linked list.
00046
          if (!m_pstHead)
00047
         {
00048
              // If the list is empty, initilize the nodes
             m_pstHead = node_;
m_pstTail = node_;
00049
00050
00051
00052
             m pstHead->prev = NULL:
00053
             m_pstTail->next = NULL;
00054
              return;
00055
00056
          \ensuremath{//} Move the tail node, and assign it to the new node just passed in
00057
00058
          m_pstTail->next = node_;
00059
          node_->prev = m_pstTail;
          node_->next = NULL;
00060
00061
          m_pstTail = node_;
00062 }
00063
00064 //---
00065 void DoubleLinkList::Remove(LinkListNode *node_)
00066 {
00067
          KERNEL_ASSERT( node_ );
00068
00069
          if (node_->prev)
00070
00071 #if SAFE_UNLINK
00072
             if (node_->prev->next != node_)
00073
```

```
Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00075
              }
00076 #endif
00077
              node_->prev->next = node_->next;
00078
00079
          if (node_->next)
00081 #if SAFE_UNLINK
00082
               if (node_->next->prev != node_)
00083
              {
                  Kernel::Panic(PANIC LIST UNLINK FAILED);
00084
00085
              }
00086 #endif
00087
              node_->next->prev = node_->prev;
00088
00089
          if (node_ == m_pstHead)
00090
00091
              m pstHead = node ->next;
00092
00093
          if (node_ == m_pstTail)
00094
          {
00095
              m_pstTail = node_->prev;
00096
          }
00097
00098
          node_->ClearNode();
00099 }
00100
00101 //--
00102 void CircularLinkList::Add(LinkListNode *node_)
00103 {
00104
          KERNEL ASSERT ( node );
00105
00106
          \ensuremath{//} Add a node to the end of the linked list.
00107
          if (!m_pstHead)
00108
              // If the list is empty, initilize the nodes
00109
              m_pstHead = node_;
m_pstTail = node_;
00110
00111
00112
00113
              m_pstHead->prev = m_pstHead;
00114
              m_pstHead->next = m_pstHead;
00115
              return;
00116
          }
00117
00118
          \ensuremath{//} Move the tail node, and assign it to the new node just passed in
          m_pstTail->next = node_;
00119
          node_->prev = m_pstTail;
node_->next = m_pstHead;
00120
00121
          m_pstTail = node_;
00122
          m_pstHead->prev = node_;
00123
00124 }
00125
00126 //--
00127 void CircularLinkList::Remove(LinkListNode *node_)
00128 {
00129
          KERNEL ASSERT ( node );
00130
00131
          // Check to see if this is the head of the list...
00132
          if ((node_ == m_pstHead) && (m_pstHead == m_pstTail))
00133
00134
              // Clear the head and tail pointers - nothing else left.
00135
              m pstHead = NULL;
00136
              m_pstTail = NULL;
00137
              return;
00138
          }
00139
00140 #if SAFE UNLINK
        // Verify that all nodes are properly connected
00141
          if ((node_->prev->next != node_) || (node_->next->prev != node_))
00142
00143
          {
00144
              Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00145
00146 #endif
00147
00148
          // This is a circularly linked list - no need to check for connection,
00149
          // just remove the node.
00150
          node_->next->prev = node_->prev;
00151
          node_->prev->next = node_->next;
00152
00153
          if (node == m pstHead)
00154
          {
00155
              m_pstHead = m_pstHead->next;
00156
00157
          if (node_ == m_pstTail)
00158
              m_pstTail = m_pstTail->prev;
00159
00160
          }
```

```
00161
          node_->ClearNode();
00162 }
00163
00164 //----
00165 void CircularLinkList::PivotForward()
00166 {
00167
          if (m_pstHead)
00168
00169
              m_pstHead = m_pstHead->next;
00170
              m_pstTail = m_pstTail->next;
00171
          }
00172 }
00173
00174 //---
00175 void CircularLinkList::PivotBackward()
00176 {
00177
          if (m_pstHead)
00178
              m_pstHead = m_pstHead->prev;
00180
              m_pstTail = m_pstTail->prev;
00181
00182 }
```

15.33 /home/moslevin/Project/R1/kernel/message.cpp File Reference

Inter-thread communications via message passing.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "message.h"
#include "threadport.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

Macros

#define __FILE_ID__ MESSAGE_CPP
 File ID used in kernel trace calls.

15.33.1 Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

15.34 message.cpp

```
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
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "message.h"
00026 #include "threadport.h"
00027 #include "kerneldebug.h"
00028
```

```
00030 #if defined __FILE_ID__
00031
         #undef ___FILE_ID___
00032 #endif
00033 #define __FILE_ID__
                           MESSAGE CPP
00034
00035
00036 #if KERNEL_USE_MESSAGE
00037
00038 #if KERNEL_USE_TIMEOUTS
         #include "timerlist.h"
00039
00040 #endif
00041
00042 Message GlobalMessagePool::m_aclMessagePool[
     GLOBAL_MESSAGE_POOL_SIZE];
00043 DoubleLinkList GlobalMessagePool::m_clList;
00044
00045 //-
00046 void GlobalMessagePool::Init()
00047 {
          K_UCHAR i;
00048
00049
         GlobalMessagePool::m_clList.Init();
         for (i = 0; i < GLOBAL_MESSAGE_POOL_SIZE; i++)</pre>
00050
00051
         {
00052
              GlobalMessagePool::m_aclMessagePool[i].Init();
00053
             GlobalMessagePool::m_clList.Add(&(GlobalMessagePool::m_aclMessagePool[i]));
00054
00055 }
00056
00057 //----
00058 void GlobalMessagePool::Push( Message *pclMessage_ )
00059 {
00060
          KERNEL_ASSERT( pclMessage_ );
00061
00062
         CS_ENTER();
00063
00064
         GlobalMessagePool::m_clList.Add(pclMessage_);
00065
00066
         CS_EXIT();
00067 }
00068
00069 //---
00070 Message *GlobalMessagePool::Pop()
00071 {
00072
          Message *pclRet;
00073
         CS_ENTER();
00074
00075
         pclRet = static_cast<Message*>( GlobalMessagePool::m_clList.GetHead() );
00076
          if (0 != pclRet)
00077
         {
00078
              GlobalMessagePool::m_clList.Remove( static_cast<LinkListNode*>( pclRet ) );
00079
08000
00081
         CS EXIT();
00082
         return pclRet;
00083 }
00084
00085 //---
00086 void MessageQueue::Init()
00087 {
00088
         m clSemaphore.Init(0, GLOBAL MESSAGE POOL SIZE);
00089 }
00090
00091 //---
00092 Message *MessageQueue::Receive()
00093 {
00094 #if KERNEL_USE_TIMEOUTS
00095
         return Receive i(0):
00096 #else
00097
         return Receive_i();
00098 #endif
00099 }
00100
00101 //----
00102 #if KERNEL_USE_TIMEOUTS
00103 Message *MessageQueue::Receive( K_ULONG ulTimeWaitMS_)
00104 {
00105
         return Receive_i( ulTimeWaitMS_ );
00106 }
00107 #endif
00108
00109 //-
00110 #if KERNEL_USE_TIMEOUTS
00111 Message *MessageQueue::Receive_i( K_ULONG ulTimeWaitMS_ )
00112 #else
00113 Message *MessageQueue::Receive_i( void )
00114 #endif
```

```
00115 {
00116
          Message *pclRet;
00117
00118
          \ensuremath{//} Block the current thread on the counting semaphore
00119 #if KERNEL USE TIMEOUTS
       if (!m_clSemaphore.Pend(ulTimeWaitMS_))
00120
00121
00122
              return NULL;
00123
00124 #else
00125
        m clSemaphore.Pend();
00126 #endif
00127
         CS_ENTER();
00128
00129
00130
         \ensuremath{//} Pop the head of the message queue and return it
          pclRet = static_cast<Message*>( m_clLinkList.GetHead() );
00131
         m_clLinkList.Remove(static_cast<Message*>(pclRet));
00132
00133
00134
          CS_EXIT();
00135
00136
          return pclRet;
00137 }
00138
00139 //-
00140 void MessageQueue::Send( Message *pclSrc_ )
00141 {
00142
          KERNEL_ASSERT( pclSrc_ );
00143
00144
          CS ENTER();
00145
00146
          // Add the message to the head of the linked list
00147
         m_clLinkList.Add( pclSrc_ );
00148
00149
          \ensuremath{//} Post the semaphore, waking the blocking thread for the queue.
00150
          m_clSemaphore.Post();
00151
00152
          CS_EXIT();
00153 }
00154
00155 //---
00156 K_USHORT MessageQueue::GetCount()
00157 {
00158
          return m_clSemaphore.GetCount();
00160 #endif //KERNEL_USE_MESSAGE
```

15.35 /home/moslevin/Project/R1/kernel/mutex.cpp File Reference

Mutual-exclusion object.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "mutex.h"
#include "kerneldebug.h"
```

Macros

• #define __FILE_ID__ MUTEX_CPP

File ID used in kernel trace calls.

Functions

void TimedMutex_Calback (Thread *pclOwner_, void *pvData_)

TimedMutex_Calback.

15.35.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

15.35.2 Function Documentation

```
15.35.2.1 void TimedMutex_Calback ( Thread * pclOwner_, void * pvData_ )
```

TimedMutex Calback.

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

pclOwner_	Pointer to the thread to wake
pvData_	Pointer to the mutex object that the thread is blocked on

Definition at line 48 of file mutex.cpp.

15.36 mutex.cpp

```
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
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022
00023 #include "blocking.h"
00024 #include "mutex.h"
00025 #include "kerneldebug.h"
00026 //-
00027 #if defined __FILE_ID__
00028
          #undef ___FILE_ID__
00029 #endif
00030 #define __FILE_ID__
                               MUTEX CPP
00031
00032
00033 #if KERNEL_USE_MUTEX
00034
00035 #if KERNEL_USE_TIMEOUTS
00036
00037 //
00048 void TimedMutex_Calback(Thread *pclOwner_, void *pvData_)
00049 {
00050
          Mutex *pclMutex = static_cast<Mutex*>(pvData_);
00051
00052
          \ensuremath{//} Indicate that the semaphore has expired on the thread
00053
          pclOwner_->SetExpired(true);
00054
00055
           // Wake up the thread that was blocked on this semaphore.
00056
          pclMutex->WakeMe(pclOwner_);
00057
          if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
00058
      () ->GetCurPriority())
00059
          {
00060
              Thread::Yield();
00061
00062 }
00063
00064 //--
00065 void Mutex::WakeMe(Thread *pclOwner_)
00066 {
```

15.36 mutex.cpp 145

```
// Remove from the semaphore waitlist and back to its ready list.
00068
          UnBlock(pclOwner_);
00069 }
00070
00071 #endif
00072
00073 //---
00074 K_UCHAR Mutex::WakeNext()
00075 {
00076
          Thread *pclChosenOne = NULL;
00077
00078
          // Get the highest priority waiter thread
00079
          pclChosenOne = m_clBlockList.HighestWaiter();
08000
00081
          // Unblock the thread
00082
          UnBlock (pclChosenOne);
00083
00084
          // The chosen one now owns the mutex
          m_pclOwner = pclChosenOne;
00085
00086
00087
          // Signal a context switch if it's a greater than or equal to the current priority
00088
          if (pclChosenOne->GetCurPriority() >=
     Scheduler::GetCurrentThread() ->GetCurPriority())
00089
          {
00090
              return 1;
00091
00092
          return 0;
00093 }
00094
00095 //-----
00096 void Mutex::Init()
00097 {
00098
          // Reset the data in the mutex
                           // The mutex is free.
00099
          m_bReady = 1;
          m_ucMaxPri = 0;
m_pclOwner = NULL;
                                     \ensuremath{//} Set the maximum priority inheritence state
00100
                                     // Clear the mutex owner
00101
          m_ucRecurse = 0;
                                     // Reset recurse count
00102
00103 }
00104
00105 //---
00106 #if KERNEL_USE_TIMEOUTS
00107 K_BOOL Mutex::Claim_i(K_ULONG ulWaitTimeMS_)
00108 #else
00109 void Mutex::Claim_i(void)
00110 #endif
00111 {
00112
          KERNEL_TRACE_1 ( STR_MUTEX_CLAIM_1, (K_USHORT)
g_pstCurrent->GetID() );
00113
00114 #if KERNEL_USE_TIMEOUTS
00115
          Timer clTimer;
00116
         K_BOOL bUseTimer = false;
00117 #endif
00118
          // Disable the scheduler while claiming the mutex - we're dealing with all
00119
          // sorts of private thread data, can't have a thread switch while messing
00120
00121
          // with internal data structures.
00122
          Scheduler::SetScheduler(0);
00123
00124
          \ensuremath{//} Check to see if the mutex is claimed or not
00125
          if (m_bReady != 0)
00126
          {
00127
              // Mutex isn't claimed, claim it.
00128
              m_bReady = 0;
00129
              m_ucRecurse = 0;
              m_ucMaxPri = g_pstCurrent->GetPriority();
m_pclOwner = g_pstCurrent;
00130
00131
00132
00133
              Scheduler::SetScheduler(1);
00134
00135 #if KERNEL_USE_TIMEOUTS
00136
              return true;
00137 #else
00138
              return;
00139 #endif
00140
         }
00141
00142
          // If the mutex is already claimed, check to see if this is the owner thread,
00143
         // since we allow the mutex to be claimed recursively.
          if (g_pstCurrent == m_pclOwner)
00144
00145
00146
               // Ensure that we haven't exceeded the maximum recursive-lock count
00147
              KERNEL_ASSERT( (m_ucRecurse < 255) );</pre>
00148
              m_ucRecurse++;
00149
              // Increment the lock count and bail
00150
00151
              Scheduler::SetScheduler(1);
```

```
00152 #if KERNEL_USE_TIMEOUTS
00153
             return true;
00154 #else
00155
              return;
00156 #endif
00157
        }
00158
00159
         // The mutex is claimed already - we have to block now. Move the
00160
          // current thread to the list of threads waiting on the mutex.
00161 #if KERNEL USE TIMEOUTS
00162
         if (ulWaitTimeMS_)
00163
          {
00164
              g_pstCurrent->SetExpired(false);
00165
              clTimer.Init();
00166
              clTimer.Start(0, ulWaitTimeMS_, (TimerCallback_t)
     TimedMutex_Calback, (void*)this);
bUseTimer = true;
00167
00168
00169 #endif
00170
         Block(g_pstCurrent);
00171
00172
          // Check if priority inheritence is necessary. We do this in order
          // to ensure that we don't end up with priority inversions in case
00173
          // multiple threads are waiting on the same resource.
00174
00175
          if (m_ucMaxPri <= g_pstCurrent->GetPriority())
00176
00177
              m_ucMaxPri = g_pstCurrent->GetPriority();
00178
00179
              Thread *pclTemp = static_cast<Thread*>(m_clBlockList.GetHead());
00180
              while (pclTemp)
00181
              {
00182
                  pclTemp->InheritPriority(m_ucMaxPri);
00183
                  if (pclTemp == static_cast<Thread*>(m_clBlockList.GetTail()) )
00184
00185
00186
00187
                  pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00188
00189
              m_pclOwner->InheritPriority(m_ucMaxPri);
00190
         }
00191
          \ensuremath{//} Done with thread data -reenable the scheduler
00192
00193
         Scheduler::SetScheduler(1):
00194
00195
          // Switch threads if this thread acquired the mutex
00196
00197
00198 #if KERNEL_USE_TIMEOUTS
         if (bUseTimer)
00199
00200
         {
00201
              clTimer.Stop();
00202
              return (g_pstCurrent->GetExpired() == 0);
00203
         }
00204
          return true;
00205 #endif
00206 }
00207
00208 //----
00209 void Mutex::Claim(void)
00210 {
00211 #if KERNEL_USE_TIMEOUTS
00212
         Claim_i(0);
00213 #else
00214
         Claim_i();
00215 #endif
00216 }
00217
00218 //---
00219 #if KERNEL_USE_TIMEOUTS
00220 K_BOOL Mutex::Claim(K_ULONG ulWaitTimeMS_)
00221 {
00222
          return Claim_i(ulWaitTimeMS_);
00223 }
00224 #endif
00225
00226 //-
00227 void Mutex::Release()
00228 {
         KERNEL_TRACE_1 ( STR_MUTEX_RELEASE_1, (K_USHORT)
00229
     g_pstCurrent->GetID());
00230
00231
          K_BOOL bSchedule = 0;
00232
00233
          // Disable the scheduler while we deal with internal data structures.
00234
          Scheduler::SetScheduler(0);
00235
00236
          // This thread had better be the one that owns the mutex currently...
```

```
00237
          KERNEL_ASSERT( (g_pstCurrent == m_pclOwner) );
00238
00239
          // If the owner had claimed the lock multiple times, decrease the lock
00240
          \ensuremath{//} count and return immediately.
00241
          if (m_ucRecurse)
00242
              m_ucRecurse--;
00244
              Scheduler::SetScheduler(1);
00245
00246
00247
         // Restore the thread's original priority
00248
          if (g_pstCurrent->GetCurPriority() != g_pstCurrent->
00249
00250
        {
00251
              g_pstCurrent->SetPriority(g_pstCurrent->
      GetPriority());
00252
00253
              // In this case, we want to reschedule
00254
              bSchedule = 1;
00255
00256
          \ensuremath{//} No threads are waiting on this semaphore?
00257
00258
         if (m_clBlockList.GetHead() == NULL)
00259
00260
              \ensuremath{//} Re-initialize the mutex to its default values
00261
              m_bReady = 1;
              m_ucMaxPri = 0;
00262
00263
              m_pclOwner = NULL;
00264
          }
00265
         else
00266
         {
00267
              // Wake the highest priority Thread pending on the mutex
00268
              if(WakeNext())
00269
00270
                   // Switch threads if it's higher or equal priority than the current thread
00271
                  bSchedule = 1;
00272
00273
         }
00274
00275
          \ensuremath{//} Must enable the scheduler again in order to switch threads.
00276
          Scheduler::SetScheduler(1);
00277
          if (bSchedule)
00278
00279
               // Switch threads if a higher-priority thread was woken
00280
00281
          }
00282 }
00283
00284 #endif //KERNEL_USE_MUTEX
```

15.37 /home/moslevin/Project/R1/kernel/profile.cpp File Reference

Code profiling utilities.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kernelprofile.h"
#include "threadport.h"
#include "kerneldebug.h"
```

Macros

#define __FILE_ID__ PROFILE_CPP
 File ID used in kernel trace calls.

15.37.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

15.38 profile.cpp

```
00001 /*----
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "profile.h"
00024 #include "kernelprofile.h"
00025 #include "threadport.h"
00026 #include "kerneldebug.h"
00027 //----
00028 #if defined __FILE_ID_
00029 #undef __FILE_ID_
00030 #endif
00031 #define __FILE_ID__
                             PROFILE_CPP
00032
00033
00034 #if KERNEL_USE_PROFILER
00035
00037 void ProfileTimer::Init()
00038 {
00039
          m ulCumulative = 0:
         m ulCurrentIteration = 0:
00040
00041
         m_usIterations = 0;
00042
         m_bActive = 0;
00043 }
00044
00045 //----
00046 void ProfileTimer::Start()
00047 {
00048
          if (!m_bActive)
00049
00050
              CS_ENTER();
00051
              m_ulCurrentIteration = 0;
00052
              m_ulInitialEpoch = Profiler::GetEpoch();
00053
              m_usInitial = Profiler::Read();
00054
              CS_EXIT();
00055
              m_bActive = 1;
00056
00057 }
00058
00059 //--
00060 void ProfileTimer::Stop()
00061 {
00062
          if (m_bActive)
00063
00064
              K_USHORT usFinal;
00065
              K_ULONG ulEpoch;
             CS_ENTER();
00066
             usFinal = Profiler::Read();
ulEpoch = Profiler::GetEpoch();
00067
00068
00069
              // Compute total for current iteration...
00070
              m_ulCurrentIteration = ComputeCurrentTicks(usFinal, ulEpoch)
00071
              m ulCumulative += m ulCurrentIteration;
00072
              m_usIterations++;
00073
              CS_EXIT();
00074
              m_bActive = 0;
00075
00076 }
00077
00078 //-
00079 K_ULONG ProfileTimer::GetAverage()
00080 {
00081
          if (m_usIterations)
00082
00083
              return m_ulCumulative / (K_ULONG) m_usIterations;
00084
00085
          return 0;
```

```
00086 }
00087
00088 //--
00089 K_ULONG ProfileTimer::GetCurrent()
00090 {
00091
00092
         if (m_bActive)
00093
00094
             K_USHORT usCurrent;
00095
             K_ULONG ulEpoch;
             CS_ENTER();
00096
            usCurrent = Profiler::Read();
00097
             ulEpoch = Profiler::GetEpoch();
00098
00099
             return ComputeCurrentTicks(usCurrent, ulEpoch);
00100
00101
         return m_ulCurrentIteration;
00102
00103 }
00104
00105 //-
00106 K_ULONG ProfileTimer::ComputeCurrentTicks(
     K_USHORT usCurrent_, K_ULONG ulEpoch_)
00107 {
         K ULONG ulTotal:
00108
00109
         K_ULONG ulOverflows;
00110
00111
         ulOverflows = ulEpoch_ - m_ulInitialEpoch;
00112
         // More than one overflow...
00113
00114
         if (ulOverflows > 1)
00115
             00116
00117
00118
                     (K_ULONG) usCurrent_;
00119
         ^{\prime\prime} Only one overflow, or one overflow that has yet to be processed
00120
00121
         else if (ulOverflows || (usCurrent_ < m_usInitial))</pre>
00122
00123
             ulTotal = (K_ULONG) (TICKS_PER_OVERFLOW - m_usInitial) +
00124
                     (K_ULONG) usCurrent_;
00125
         // No overflows, none pending.
00126
00127
         else
00128
         {
00129
             ulTotal = (K_ULONG) (usCurrent_ - m_usInitial);
00130
00131
00132
         return ulTotal;
00133 }
00134
00135 #endif
```

15.39 /home/moslevin/Project/R1/kernel/public/atomic.h File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "threadport.h"
```

15.39.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

15.40 atomic.h



```
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
00041 public:
00048
          static K_UCHAR Set( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
          static K_USHORT Set( K_USHORT *pusSource_, K_USHORT usVal_ );
static K_ULONG Set( K_ULONG *pulSource_, K_ULONG ulVal_ );
00049
00050
00051
          static K_UCHAR Add( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
00059
          static K_USHORT Add( K_USHORT *pusSource_, K_USHORT usVal_ );
00060
          static K_ULONG Add( K_ULONG *pulSource_, K_ULONG ulVal_ );
00061
00068
          static K_UCHAR Sub( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
00069
          static K_USHORT Sub( K_USHORT *pusSource_, K_USHORT usVal_ );
00070
          static K_ULONG Sub( K_ULONG *pulSource_, K_ULONG ulVal_ );
00071
00086
          static K_BOOL TestAndSet( K_BOOL *pbLock );
00087 };
88000
00089 #endif // KERNEL USE ATOMIC
00091 #endif //__ATOMIC_H_
```

15.41 /home/moslevin/Project/R1/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 primatives.

15.41.1 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 consitutes a Block or Unblock condition.

15.42 blocking.h 151

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.

15.42 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"
00053 #include "11.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
00101
          void UnBlock(Thread *pclThread_);
00102
00107
          ThreadList m_clBlockList;
00108 };
00109
00110 #endif
00112 #endif
```

15.43 /home/moslevin/Project/R1/kernel/public/debugtokens.h File Reference

Hex codes/translation tables used for efficient string tokenization.

Macros

- #define BLOCKING_CPP 0x0001 /* SUBSTITUTE="blocking.cpp" */
 - Source file names start at 0x0000.
- #define BLOCKING_H 0x1000 /* SUBSTITUTE="blocking.h" */

Header file names start at 0x1000.

#define STR_PANIC 0x2000 /* SUBSTITUTE="!Panic!" */

Indexed strings start at 0x2000.

15.43.1 Detailed Description

Hex codes/translation tables used for efficient string tokenization. We use this for efficiently encoding strings used for kernel traces, debug prints, etc. The upside - this is really fast and efficient for encoding strings and data. Downside? The tools need to parse this header file in order to convert the enumerated data into actual strings, decoding them.

Definition in file debugtokens.h.

15.44 debugtokens.h

```
00002
00003
00004
00005
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 =====
00025 #ifndef __DEBUG_TOKENS_H__
00026 #define __DEBUG_TOKENS_H_
00027 //----
00029 #define BLOCKING_CPP
                                                 /* SUBSTITUTE="blocking.cpp" */
                                    0x0001
                                                 /* SUBSTITUTE="driver.cpp"
00030 #define DRIVER_CPP
                                    0x0002
00031 #define KERNEL_CPP
                                    0x0003
                                                  /* SUBSTITUTE="kernel.cpp" */
00032 #define LL_CPP
                                    0x0004
                                                  /* SUBSTITUTE="11.cpp" */
00033 #define MESSAGE_CPP
                                    0x0005
                                                 /* SUBSTITUTE="message.cpp" */
                                                  /* SUBSTITUTE="mutex.cpp" */
00034 #define MUTEX_CPP
                                    0×0006
                                                  /* SUBSTITUTE="profile.cpp" */
00035 #define PROFILE CPP
                                    0×0007
                                                  /* SUBSTITUTE="quantum.cpp" */
00036 #define QUANTUM_CPP
                                    0x0008
00037 #define SCHEDULER_CPP
                                                  /* SUBSTITUTE="scheduler.cpp" */
                                    0x0009
                                                 /* SUBSTITUTE="semaphore.cpp" */
/* SUBSTITUTE="thread.cpp" */
00038 #define SEMAPHORE_CPP
                                     0x000A
00039 #define THREAD_CPP
                                    0x000B
                                                 /* SUBSTITUTE="threadlist.cpp" */
00040 #define THREADLIST_CPP
                                    0×000C
                                                  /* SUBSTITUTE="timerlist.cpp" */
00041 #define TIMERLIST_CPP
                                    0x000D
                                                  /* SUBSTITUTE="kernelswi.cpp" */
00042 #define KERNELSWI CPP
                                    0x000E
                                                  /* SUBSTITUTE="kerneltimer.cpp"
00043 #define KERNELTIMER_CPP
                                     0x000F
                                                  /* SUBSTITUTE="kernelprofile.cpp" */
00044 #define KPROFILE_CPP
                                     0x0010
                                                  /* SUBSTITUTE="threadport.cpp"
00045 #define THREADPORT_CPP
                                    0×0011
                                                  /* SUBSTITUTE="timer.cpp" */
00046 #define TIMER_CPP
                                    0x0012
00047
00048 //-
00050 #define BLOCKING_H
                                    0x1000
                                                 /* SUBSTITUTE="blocking.h" */
00051 #define DRIVER_H
                                                  /* SUBSTITUTE="driver.h"
                                                  /* SUBSTITUTE="kernel.h" */
00052 #define KERNEL_H
                                    0x1002
                                                 /* SUBSTITUTE="kerneltypes.h" */
00053 #define KERNELTYPES_H
                                    0x1003
                                                  /* SUBSTITUTE="11.h" */
00054 #define LL H
                                    0 \times 1004
                                                 /* SUBSTITUTE="manual.h" */
00055 #define MANUAL H
                                    0x1005
00056 #define MARK3CFG_H
                                    0x1006
                                                  /* SUBSTITUTE="mark3cfg.h" */
00057 #define MESSAGE_H
                                                  /* SUBSTITUTE="message.h" */
                                    0x1007
00058 #define MUTEX_H
                                                  /* SUBSTITUTE="mutex.h"
                                    0x1008
                                                 /* SUBSTITUTE="profile.h" */
/* SUBSTITUTE="profiling_results.h" */
/* SUBSTITUTE="quantum.h" */
00059 #define PROFILE_H
                                    0x1009
00060 #define PROFILING_RESULTS_H 0x100A
00061 #define OUANTUM H
                                    0x100B
                                                  /* SUBSTITUTE="scheduler.h"
00062 #define SCHEDULER_H
                                    0x100C
                                                 /* SUBSTITUTE="ksemaphore.h"
/* SUBSTITUTE="thread.h" */
00063 #define SEMAPHORE_H
                                    0x100D
00064 #define THREAD_H
                                    0x100E
                                                 /* SUBSTITUTE="threadlist.h" */
00065 #define THREADLIST H
                                    0x100F
                                                  /* SUBSTITUTE="timerlist.h" */
00066 #define TIMERLIST_H
                                    0x1010
                                                  /* SUBSTITUTE="kernelswi.h */
00067 #define KERNELSWI H
                                    0x1011
00068 #define KERNELTIMER_H
                                                  /* SUBSTITUTE="kerneltimer.h */
                                    0x1012
                                                  /* SUBSTITUTE="kernelprofile.h" */
00069 #define KPROFILE_H
                                    0x1013
                                                  /* SUBSTITUTE="threadport.h" */
00070 #define THREADPORT_H
                                    0x1014
00071
00072 //----
00074 #define STR_PANIC
                                                      /* SUBSTITUTE="!Panic!" */
                                         0x2000
                                                      /* SUBSTITUTE="Initializing Kernel Objects" */
00075 #define STR MARK3 INIT
                                         0x2001
                                                      /* SUBSTITUTE="Starting Kernel"
00076 #define STR_KERNEL_ENTER
                                         0x2002
00077 #define STR_THREAD_START
                                                      /* SUBSTITUTE="Switching to First Thread" */
                                         0x2003
00078 #define STR_START_ERROR return" */
                                                      /* SUBSTITUTE="Error starting kernel - function should never
                                         0x2004
00079 #define STR_THREAD_CREATE
00080 #define STR_STACK_SIZE_1
                                         0 \times 2005
                                                      /* SUBSTITUTE="Creating Thread"
                                                      /* SUBSTITUTE=" Stack Size: %1" */
/* SUBSTITUTE=" Priority: %1" */
                                         0x2006
00081 #define STR_PRIORITY_1
                                         0x2007
                                                      /* SUBSTITUTE="
00082 #define STR_THREAD_ID_1
                                         0x2008
                                                                        Thread ID: %1"
```

```
00083 #define STR_ENTRYPOINT_1
                                                      /* SUBSTITUTE=" EntryPoint: %1" */
                                                      /* SUBSTITUTE="Context Switch To Thread: %1" */
00084 #define STR_CONTEXT_SWITCH_1
                                         0x200A
                                                     /* SUBSTITUTE="Idling CPU" */
00085 #define STR_IDLING
                                         0x200B
                                                     /* SUBSTITUTE="Waking up" */
00086 #define STR_WAKEUP
                                         0x200C
                                                     /* SUBSTITUTE="Semaphore Pend: %1" */
00087 #define STR_SEMAPHORE_PEND_1
                                         0x200D
                                                     /* SUBSTITUTE="Semaphore Post: %1" */
00088 #define STR_SEMAPHORE_POST_1
                                         0x200E
0x200F
                                                     /* SUBSTITUTE="Mutex Claim: %1" */
00089 #define STR_MUTEX_CLAIM_1
00090 #define STR_MUTEX_RELEASE_1
                                         0x2010 /* SUBSTITUTE="Mutex Release: %1" */
                                                      /* SUBSTITUTE="Thread %1 Blocked" */
00091 #define STR_THREAD_BLOCK_1
                                         0x2011
                                         0x2012-2015
00092 #define STR_THREAD_UNBLOCK_1
                                                          /* SUBSTITUTE="Thread %1 Unblocked" */
                                         0x2013 /* SUBSTITUTE="Inread %1 Unblock
0x2014 /* SUBSTITUTE="Scheduler chase %1"
00093 #define STR_ASSERT_FAILED
                                         0x2014 /* SUBSTITUTE="Scheduler chose %1" */
0x2015 /* SUBSTITUTE="Thread Start: %1" */
0x2016 /* SUBSTITUTE="Thread Exit: %1" */
00094 #define STR_SCHEDULE_1
00095 #define STR_THREAD_START_1
00096 #define STR_THREAD_EXIT_1
00097
00098 //----
00099 #define STR_UNDEFINED
                                       Oxffff /* SUBSTITUTE="UNDEFINED" */
00100 #endif
```

15.45 /home/moslevin/Project/R1/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.

15.45.1 Detailed Description

Driver abstraction framework.

15.45.2 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.

15.45.3 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.

15.45.4 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 he 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:

```
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!");
```

Definition in file driver.h.

15.46 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 "11.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 K_UCHAR Open() = 0;
00139
00147
         virtual K_UCHAR Close() = 0;
00148
         virtual K_USHORT Read( K_USHORT usBytes_,
00164
00165
                                       K_UCHAR *pucData_) = 0;
00166
00183
         virtual K_USHORT Write( K_USHORT usBytes_,
00184
                                        K_UCHAR *pucData_) = 0;
00185
00208
         virtual K_USHORT Control (K_USHORT usEvent_
00209
                                          void *pvDataIn ,
00210
                                          K_USHORT usSizeIn_,
00211
                                          void *pvDataOut_,
00212
                                          K_USHORT usSizeOut_ ) = 0;
00213
00222
         void SetName( const K_CHAR *pcName_ ) { m_pcPath = pcName_; }
         const K_CHAR *GetPath() { return m_pcPath; }
00232
00233 private:
00234
          const K_CHAR *m_pcPath;
00236
00237 };
00239 //----
00244 class DriverList
00245 {
00246 public:
00254
        static void Init();
00255
         static void Add( Driver *pclDriver_ ) { m_clDriverList.
     Add(pclDriver_); }
00265
         static void Remove( Driver *pclDriver_) { m_clDriverList.
00274
     Remove(pclDriver_); }
00275
00282
         static Driver *FindByPath( const K_CHAR *m_pcPath );
00283
00284 private:
00285
         static DoubleLinkList m clDriverList;
00287
00288 };
00290 #endif //KERNEL_USE_DRIVER
00291
00292 #endif
```

15.47 /home/moslevin/Project/R1/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.

15.47.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

15.48 eventflag.h

```
00001
00002
00003
00004
00005
00006
00007
80000
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 #ifndef __EVENTFLAG_H__
00020 #define ___EVENTFLAG_H_
00021
00022 #include "mark3cfg.h"
00023 #include "kernel.h'
00024 #include "kerneltypes.h"
00025 #include "blocking.h"
00026 #include "thread.h"
00027
00028 #if KERNEL_USE_EVENTFLAG
00030 //---
00046 class EventFlag : public BlockingObject
00047 {
00048 public:
          void Init() { m_usSetMask = 0; m_clBlockList.
00052
00053
00061
          K_USHORT Wait(K_USHORT usMask_, EventFlagOperation_t eMode_);
00062
00063 #if KERNEL USE TIMEOUTS
00064
00072
          K_USHORT Wait(K_USHORT usMask_, EventFlagOperation_t eMode_,
      K_ULONG ulTimeMS_);
00073
00081
          void WakeMe(Thread *pclOwner_);
00082
00083 #endif
00084
          void Set(K_USHORT usMask_);
00091
00096
          void Clear(K_USHORT usMask_);
00097
          K USHORT GetMask():
00102
00103
00104 private:
00105
00106 #if KERNEL_USE_TIMEOUTS
00107
          K_USHORT Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_
00119
        K ULONG ulTimeMS );
00120 #else
00121
00131
          K_USHORT Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_
00132 #endif
00133
00134
          K_USHORT m_usSetMask;
00135 };
00136
00137 #endif //KERNEL_USE_EVENTFLAG
00138 #endif //__EVENTFLAG_H__
00139
```

15.49 /home/moslevin/Project/R1/kernel/public/kernel.h File Reference

Kernel initialization and startup class.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "paniccodes.h"
#include "thread.h"
```

15.50 kernel.h 157

Classes

· class Kernel

Class that encapsulates all of the kernel startup functions.

15.49.1 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.

15.50 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:
00059
          static void Init(void);
00060
00073
          static void Start (void);
00074
                                         { return m_bIsStarted;
00080
          static K BOOL IsStarted()
00081
          static void SetPanic( panic_func_t pfPanic_ ) {
00089
      m_pfPanic = pfPanic_; }
00090
00095
          static K_BOOL IsPanic()
                                         { return m_bIsPanic; }
00096
          static void Panic(K_USHORT usCause_);
00101
00102
00103 #if KERNEL_USE_IDLE_FUNC
00104
00109
          static void SetIdleFunc( idle_func_t pfIdle_ ) {    m_pfIdle = pfIdle_; }
00110
          static void IdleFunc(void) { if (m pfIdle != 0 ) { m pfIdle(); } }
00115
00116
          static Thread *GetIdleThread(void) { return (Thread*)&
00124
      m_clIdle; }
00125 #endif
00126
00127 private:
          static K_BOOL m_bIsStarted;
00128
00129
          static K_BOOL m_bIsPanic;
00130
          static panic_func_t m_pfPanic;
```

15.51 /home/moslevin/Project/R1/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, K-A_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.

15.51.1 Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.h.

15.51.2 Enumeration Type Documentation

15.51.2.1 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

15.52 kernelaware.h

KA_COMMAND_TRACE_1 1-argument kernel traceKA_COMMAND_TRACE_2 2-argument kernel traceKA_COMMAND_PRINT Print an arbitrary string of data.

Definition at line 33 of file kernelaware.h.

15.52 kernelaware.h

```
00001 /*
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
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,
          KA_COMMAND_PROFILE_START,
00037
          KA_COMMAND_PROFILE_STOP,
00038
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 K_CHAR *szStr_ );
08000
00081
00089
          static void ProfileStart ( void );
00090
00091
00098
          static void ProfileStop( void );
00099
00100
          static void ProfileReport( void );
00108
00109
00110
00118
          static void ExitSimulator( void );
00119
00120
00128
          static void Print( const K_CHAR *szStr_ );
00129
00130
          static void Trace ( K_USHORT usFile_,
00141
00142
                         K_USHORT usLine_,
00143
                         K_USHORT usCode_ );
00144
00145
          static void Trace( K_USHORT usFile_,
00157
                         K_USHORT usLine_,
00158
00159
                         K_USHORT usCode_,
00160
                         K_USHORT usArg1_);
00161
00162
          static void Trace( K_USHORT usFile_,
00175
00176
                         K_USHORT usLine_,
00177
                         K_USHORT usCode_,
```

```
K_USHORT usArg1_,
00179
                         K_USHORT usArg2_);
00180
00181
00191
          static K_BOOL IsSimulatorAware(void);
00192
00193 private:
00194
00195
          static void Trace_i( K_USHORT usFile_,
00209
                                K_USHORT usLine_,
00210
00211
                                K_USHORT usCode_,
                                K_USHORT usArg1_,
00212
00213
                                K_USHORT usArg2_,
00214
                                KernelAwareCommand_t eCmd_);
00215 };
00216
00217 #endif
00218
00219 #endif
```

15.53 /home/moslevin/Project/R1/kernel/public/kerneldebug.h File Reference

Macros and functions used for assertions, kernel traces, etc.

```
#include "debugtokens.h"
#include "mark3cfg.h"
#include "tracebuffer.h"
#include "kernelaware.h"
#include "paniccodes.h"
#include "kernel.h"
```

Macros

```
#define __FILE_ID__ 0
```

Null ID.

• #define KERNEL_TRACE(x)

Null Kernel Trace Macro.

• #define KERNEL_TRACE_1(x, arg1)

Null Kernel Trace Macro.

• #define KERNEL_TRACE_2(x, arg1, arg2)

Null Kernel Trace Macro.

• #define KERNEL ASSERT(x)

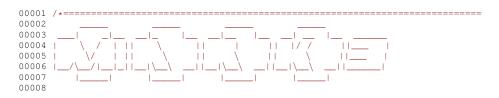
Null Kernel Assert Macro.

15.53.1 Detailed Description

Macros and functions used for assertions, kernel traces, etc.

Definition in file kerneldebug.h.

15.54 kerneldebug.h



15.54 kerneldebug.h

```
00009 -- [Mark3 Realtime Platform] -----
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========
00020 #ifndef __KERNEL_DEBUG_H_
00021 #define __KERNEL_DEBUG_H_
00022
00023 #include "debugtokens.h"
00024 #include "mark3cfg.h"
00025 #include "tracebuffer.h"
00026 #include "kernelaware.h"
00027 #include "paniccodes.h"
00028 #include "kernel.h"
00029 //--
00030 #if (KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION)
00031
00032 //-
00033 #define __FILE_ID__ STR_UNDEFINED
00035 //----
00036 #define KERNEL_TRACE( x ) \
00037 {
          K_USHORT ausMsg__[5]; \
00038
00039
           ausMsg_[0] = 0xACDC;
           ausMsg_[1] = __FILE_ID__; \
00040
00041
           ausMsg_{[2]} = _LINE_;
           ausMsg_[3] = TraceBuffer::Increment(); \
ausMsg_[4] = (K_USHORT)(x); \
00042
00043
           TraceBuffer::Write(ausMsg__, 5); \
00044
00045 };
00046
00047 //----
00048 #define KERNEL_TRACE_1( x, arg1 ) \
00049 {
           K USHORT ausMsq__[6];
00050
           ausMsg_[0] = 0xACDC; \
ausMsg_[1] = __FILE_ID__; \
00051
00052
           ausMsg_[2] = __LINE__; \
ausMsg_[3] = TraceBuffer::Increment(); \
00053
00054
           ausMsg_{[4]} = (K_{USHORT})(x); \
00055
           ausMsg__[5] = arg1; \
00056
00057
           TraceBuffer::Write(ausMsg
00058 }
00059
00060 //--
00061 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00062 {
00063
           K USHORT ausMsq [7]:
           ausMsg_{[0]} = 0xACDC;
00064
00065
           ausMsg__[1] = __FILE_ID__; \
           ausMsg_[2] = _LINE__; \
ausMsg_[3] = TraceBuffer::Increment(); \
00066
00067
           ausMsg__[4] = (K_USHORT)(x); \setminus
00068
00069
           ausMsg_{[5]} = arg1;
           ausMsg__[6] = arg2;
00070
00071
           TraceBuffer::Write(ausMsg__, 7); \
00072 }
00073
00074 //---
00075 #define KERNEL_ASSERT( x ) \
00076 {
00077
           if( (x) == false) \
00078
00079
               K_USHORT ausMsg__[5];
08000
               ausMsg_{[0]} = 0xACDC;
               ausMsg_[1] = __FILE_ID__;
ausMsg_[2] = __LINE__; \
ausMsg_[3] = TraceBuffer::Increment(); \
ausMsg_[4] = STR_ASSERT_FAILED; \
00081
00082
00083
00084
00085
                TraceBuffer::Write(ausMsg___, 5);
00086
                Kernel::Panic(PANIC_ASSERT_FAILED); \
00087
          }
00088 }
00089
00090 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION)
00091 //---
00092 #define ___FILE_ID__
                                     STR_UNDEFINED
00093
00094 //----
00095 #define KERNEL_TRACE( x )
00096 {
00097
           KernelAware::Trace( __FILE_ID__, __LINE__, x ); \
00098 };
00099
00100 //----
00101 #define KERNEL_TRACE_1( x, arg1 ) \
```

```
KernelAware::Trace( __FILE_ID__, __LINE__, x, arg1 ); \
00104 }
00105
00106 //--
00107 #define KERNEL_TRACE_2(x, arg1, arg2) \
00109
          KernelAware::Trace( __FILE_ID__, __LINE__, x, arg1, arg2 ); \
00110 }
00111
00112 //----
00113 #define KERNEL_ASSERT( x ) \
00114 {
00115
          if( (x) == false) \
00116
              KernelAware::Trace( __FILE_ID__, __LINE__, STR_ASSERT_FAILED ); \
Kernel::Panic( PANIC_ASSERT_FAILED ); \
00117
00118
00119
00120 }
00121
00122 #else
00123 //---
00124 // Note -- when kernel-debugging is disabled, we still have to define the
\tt 00125 // macros to ensure that the expressions compile (albeit, by elimination
00126 // during pre-processing).
00128 #define __FILE_ID__
00129 //-
00130 #define KERNEL_TRACE( x )
00131 //-
00132 #define KERNEL TRACE 1(x, argl)
00133 /
00134 #define KERNEL_TRACE_2( x, arg1, arg2 )
00135 //-
00136 #define KERNEL_ASSERT( x )
00137
00138 #endif // KERNEL_USE_DEBUG
00140 #endif
```

15.55 /home/moslevin/Project/R1/kernel/public/kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

```
#include <stdint.h>
```

Macros

```
• #define K_BOOL uint8_t
```

Basic boolean data type (true = 1, false = 0)

• #define K_CHAR char

The 8-bit signed integer type used by Mark3.

• #define K UCHAR uint8 t

The 8-bit unsigned integer type used by Mark3.

• #define K_USHORT uint16_t

The 16-bit unsigned integer type used by Mark3.

• #define K_SHORT int16_t

The 16-bit signed integer type used by Mark3.

#define K_ULONG uint32_t

The 32-bit unsigned integer type used by Mark3.

• #define K LONG int32 t

The 32-bit signed integer type used by Mark3.

• #define K_ADDR uint32_t

Primative datatype representing address-size.

#define K_WORD uint32_t

Primative datatype representing a data word.

15.56 kerneltypes.h 163

Typedefs

typedef void(* panic_func_t)(K_USHORT usPanicCode_)
 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.

15.55.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

15.55.2 Enumeration Type Documentation

15.55.2.1 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 56 of file kerneltypes.h.

15.56 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(bool)
00026
         #define K BOOL
                                     bool
00027 #else
00028
          #define K_BOOL
                                     uint8_t
00029 #endif
```

```
00030
00031 #define K_CHAR
00032 #define K_UCHAR
                                  uint8_t
00033 #define K_USHORT
                                  uint16_t
00034 #define K_SHORT 00035 #define K_ULONG
                                  int16_t
uint32_t
00036 #define K_LONG
                                  int32_t
00037
00038 #if !defined(K_ADDR)
00039
           #define K_ADDR
                                  uint32 t
00040 #endif
00041 #if !defined(K_WORD)
           #define K_WORD
                                  uint32_t
00043 #endif
00044
00045 //---
00049 typedef void (*panic_func_t)( K_USHORT usPanicCode_ );
00050
00051 //---
00056 typedef enum
00057 {
00058
           EVENT_FLAG_ALL,
          EVENT_FLAG_ANY,
EVENT_FLAG_ALL_CLEAR,
EVENT_FLAG_ANY_CLEAR,
00059
00060
00061
00062 //--
00063
          EVENT_FLAG_MODES,
00064
          EVENT_FLAG_PENDING_UNBLOCK
00065 } EventFlagOperation_t;
00066
00067
00068 #endif
```

15.57 /home/moslevin/Project/R1/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

Counting semaphore, based on BlockingObject base class.

15.57.1 Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

15.58 ksemaphore.h

```
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:
00049
          void Init(K_USHORT usInitVal_, K_USHORT usMaxVal_);
00050
00059
          K BOOL Post();
00060
00067
          void Pend();
00068
00069
          K USHORT GetCount();
00081
00082
00083 #if KERNEL_USE_TIMEOUTS
00095
          K_BOOL Pend( K_ULONG ulWaitTimeMS_);
00096
00107
          void WakeMe(Thread *pclChosenOne_);
00108 #endif
00109
00110 private:
00111
00117
          K_UCHAR WakeNext();
00118
00119 #if KERNEL_USE_TIMEOUTS
00120
          K_BOOL Pend_i( K_ULONG ulWaitTimeMS_ );
00129 #else
00130
00136
          void Pend_i( void );
00137 #endif
00138
          K_USHORT m_usValue;
00139
00140
        K_USHORT m_usMaxValue;
00141
00142
00143 };
00144
00145 #endif //KERNEL_USE_SEMAPHORE
00146
00147 #endif
```

15.59 /home/moslevin/Project/R1/kernel/public/II.h File Reference

Core linked-list declarations, used by all kernel list types.

```
#include "kerneltypes.h"
```

Classes

class LinkListNode

Basic linked-list node data structure.

class LinkList

Abstract-data-type from which all other linked-lists are derived.

· class DoubleLinkList

Doubly-linked-list data type, inherited from the base LinkList type.

• class CircularLinkList

Circular-linked-list data type, inherited from the base LinkList type.

15.59.1 Detailed Description

Core linked-list declarations, used by all kernel list types. At the heart of RTOS data structures are linked lists. Having a robust and efficient set of linked-list types that we can use as a foundation for building the rest of our kernel types allows us to keep our RTOS code efficient and logically-separated.

So what data types rely on these linked-list classes?

-Threads -ThreadLists -The Scheduler -Timers, -The Timer Scheduler -Blocking objects (Semaphores, Mutexes, etc...)

Pretty much everything in the kernel uses these linked lists. By having objects inherit from the base linked-list node type, we're able to leverage the double and circular linked-list classes to manager virtually every object type in the system without duplicating code. These functions are very efficient as well, allowing for very deterministic behavior in our code.

Definition in file II.h.

15.60 II.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
00043 #ifndef __LL_H_
00044 #define __LL_H_
00045
00046 #include "kerneltypes.h"
00047
00048 //-
00049 #ifndef NULL
00050 #define NULL
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 };
00107
00108 //--
00112 class LinkList
00113 {
00114 protected:
         LinkListNode *m_pstHead;
00115
00116
         LinkListNode *m pstTail:
00117
00118 public:
```

```
void Init() { m_pstHead = NULL; m_pstTail = NULL; }
00123
00131
          virtual void Add(LinkListNode *node_) = 0;
00132
00140
          virtual void Remove(LinkListNode *node ) = 0;
00141
          LinkListNode *GetHead() { return m_pstHead; }
00150
00158
          LinkListNode *GetTail() { return m_pstTail; }
00159 };
00160
00161 //
00165 class DoubleLinkList : public LinkList
00166 {
00167 public:
00171
         DoubleLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00172
00180
         virtual void Add(LinkListNode *node );
00181
00189
          virtual void Remove(LinkListNode *node_);
00190 };
00191
00192 //---
00196 class CircularLinkList : public LinkList
00197 {
00198 public:
00199
          CircularLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00200
00208
         virtual void Add(LinkListNode *node_);
00209
00217
         virtual void Remove(LinkListNode *node_);
00218
00225
         void PivotForward();
00226
00233
          void PivotBackward();
00234 };
00235
00236 #endif
```

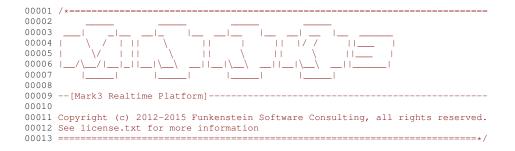
15.61 /home/moslevin/Project/R1/kernel/public/manual.h File Reference

Ascii-format documentation, used by doxygen to create various printable and viewable forms.

15.61.1 Detailed Description

Ascii-format documentation, used by doxygen to create various printable and viewable forms. Definition in file manual.h.

15.62 manual.h



15.63 /home/moslevin/Project/R1/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 "atomic.h"
#include "driver.h"
#include "kernelaware.h"
#include "profile.h"
```

15.63.1 Detailed Description

Single include file given to users of the Mark3 Kernel API.

Definition in file mark3.h.

15.64 mark3.h

```
00001 /*
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
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_
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
00041 #include "atomic.h"
00042 #include "driver.h"
00043
00044 #include "kernelaware.h"
00045
00046 #include "profile.h"
00047 #endif
```

15.65 /home/moslevin/Project/R1/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 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 inheritence, 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_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 (0)

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

#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.

15.65.1 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.

15.65.2 Macro Definition Documentation

15.65.2.1 #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 mechansims 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 144 of file mark3cfg.h.

15.65.2.2 #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 210 of file mark3cfg.h.

15.65.2.3 #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.

15.65.2.4 #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 194 of file mark3cfg.h.

15.65.2.5 #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 176 of file mark3cfg.h.

15.65.2.6 #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 123 of file mark3cfg.h.

15.65.2.7 #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 219 of file mark3cfg.h.

15.65.2.8 #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 131 of file mark3cfg.h.

15.65.2.9 #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 182 of file mark3cfg.h.

15.65.2.10 #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.

15.65.2.11 #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 109 of file mark3cfg.h.

15.65.2.12 #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 K_CHAR* pointer to the size of the thread object.

Definition at line 168 of file mark3cfg.h.

```
15.65.2.13 #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.

```
15.65.2.14 #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 (used for round-robin scheduling) is dependent on this module, as is Thread Sleep functionality. Definition at line 41 of file mark3cfg.h.

```
15.65.2.15 #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 202 of file mark3cfg.h.

15.65.2.16 #define THREAD_QUANTUM_DEFAULT (4)

This value defines the default thread quantum when KERNEL USE QUANTUM is enabled.

The thread quantum value is in milliseconds

Definition at line 101 of file mark3cfg.h.

15.66 mark3cfg.h



```
00029 #ifndef __MARK3CFG_H__
00030 #define __MARK3CFG_H_
00031
00041 #define KERNEL USE TIMERS
                                              (1)
00042
00061 #if KERNEL_USE_TIMERS
00062
          #define KERNEL_TIMERS_TICKLESS
00063 #endif
00064
00076 #if KERNEL_USE_TIMERS
         #define KERNEL_USE_TIMEOUTS
00077
00078 #else
00079
         #define KERNEL_USE_TIMEOUTS
00080 #endif
00081
00091 #if KERNEL_USE_TIMERS
00092
         #define KERNEL_USE_QUANTUM
00093 #else
        #define KERNEL_USE_QUANTUM
00095 #endif
00096
00101 #define THREAD_QUANTUM_DEFAULT
                                              (4)
00102
00109 #define KERNEL_USE_SEMAPHORE
00110
00116 #define KERNEL_USE_MUTEX
00117
00123 #define KERNEL_USE_EVENTFLAG
                                              (1)
00124
00130 #if KERNEL_USE_SEMAPHORE
         #define KERNEL_USE_MESSAGE
00132 #else
00133
       #define KERNEL_USE_MESSAGE
                                              (0)
00134 #endif
00135
00143 #if KERNEL_USE_MESSAGE
        #define GLOBAL_MESSAGE_POOL_SIZE
00146
00151 #if KERNEL_USE_TIMERS && KERNEL_USE_SEMAPHORE
00152
         #define KERNEL_USE_SLEEP
00153 #else
        #define KERNEL_USE_SLEEP
00154
00155 #endif
00156
00161 #define KERNEL_USE_DRIVER
                                             (1)
00162
00168 #define KERNEL USE THREADNAME
00169
00176 #define KERNEL_USE_DYNAMIC_THREADS
00177
00182 #define KERNEL_USE_PROFILER
00183
00188 #define KERNEL_USE_DEBUG
00189
00194 #define KERNEL_USE_ATOMIC
00195
00202 #define SAFE_UNLINK
                                              (0)
00203
00210 #define KERNEL AWARE SIMULATION
00211
00219 #define KERNEL_USE_IDLE_FUNC
                                              (1)
00220 #endif
```

15.67 /home/moslevin/Project/R1/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.

15.67.1 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.

15.67.2 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 10\,\mathrm{ms}
    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.

15.68 message.h 175

15.68 message.h

```
00001 /*=======
00002
00003
00004
00005
00006 1
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 "11.h"
00087 #include "ksemaphore.h"
00088
00089 #if KERNEL_USE_MESSAGE
00090
00091 #if KERNEL USE TIMEOUTS
         #include "timerlist.h"
00092
00093 #endif
00094
00095 //-
00099 class Message : public LinkListNode
00100 {
00101 public:
00107
          void Init() { ClearNode(); m_pvData = NULL; m_usCode = 0; }
00108
00116
          void SetData( void *pvData_ ) { m_pvData = pvData_; }
00117
00125
          void *GetData() { return m_pvData; }
00126
00134
          void SetCode( K_USHORT usCode_ ) { m_usCode = usCode_; }
00135
          K_USHORT GetCode() { return m_usCode; }
00144 private:
00145
          void *m_pvData;
00147
00148
00150
          K_USHORT m_usCode;
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
          static DoubleLinkList m clList:
00193
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( K_ULONG ulTimeWaitMS_ );
00237 #endif
00238
00247
          void Send( Message *pclSrc_ );
00248
00249
00257
          K_USHORT GetCount();
00258 private:
00259
```

```
00260 #if KERNEL_USE_TIMEOUTS
00270
         Message *Receive_i( K_ULONG ulTimeWaitMS_ );
00271 #else
00272
00279
          Message *Receive_i( void );
00280 #endif
00281
00283
          Semaphore m_clSemaphore;
00284
         DoubleLinkList m clLinkList:
00286
00287 };
00288
00289 #endif //KERNEL_USE_MESSAGE
00290
00291 #endif
```

15.69 /home/moslevin/Project/R1/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.

15.69.1 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.

15.69.2 Initializing

Initializing a mutex object by calling:

```
clMutex.Init();
```

15.69.3 Resource protection example

```
clMutex.Claim();
...
<resource protected block>
...
clMutex.Release();
```

Definition in file mutex.h.

15.70 mutex.h 177

15.70 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
00085
         void Claim();
00086
00087 #if KERNEL_USE_TIMEOUTS
00088
00097
          K_BOOL Claim(K_ULONG ulWaitTimeMS_);
00098
00111
         void WakeMe( Thread *pclOwner_ );
00112
00113 #endif
00114
00121
          void Release();
00122
00123 private:
00124
          K_UCHAR WakeNext();
00130
00131
00132
00133 #if KERNEL_USE_TIMEOUTS
00134
00142
         K_BOOL Claim_i( K_ULONG ulWaitTimeMS_ );
00143 #else
00144
00150
          void Claim_i(void);
00151 #endif
00152
00153
          K_UCHAR m_ucRecurse;
         K_UCHAR m_bReady;
00154
00155
          K_UCHAR m_ucMaxPri;
          Thread *m_pclOwner;
00157
00158 };
00159
00160 #endif //KERNEL USE MUTEX
00161
00162 #endif //__MUTEX_H_
00163
```

15.71 /home/moslevin/Project/R1/kernel/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

15.71.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file paniccodes.h.

15.72 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
00024 #define PANIC_LIST_UNLINK_FAILED
00025 #define PANIC_STACK_SLACK_VIOLATED
00026
00027 #endif // __PANIC_CODES_H
00028
```

15.73 /home/moslevin/Project/R1/kernel/public/profile.h File Reference

High-precision profiling timers.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

Classes

class ProfileTimer

Profiling timer.

15.73.1 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();</pre>
```

15.74 profile.h 179

```
}
// Get the average execution time of all iterations
ulAverageTimer = clMyTimer.GetAverage();

// Get the execution time from the last iteration
ulLastTimer = clMyTimer.GetCurrent();
```

Definition in file profile.h.

15.74 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 "11.h"
00059
00060 #if KERNEL_USE_PROFILER
00061
00069 class ProfileTimer
00070 {
00071
00072 public:
00079
          void Init();
08000
00087
          void Start();
00088
00095
          void Stop();
00096
00104
          K ULONG GetAverage();
00105
00114
          K_ULONG GetCurrent();
00115
00116 private:
00117
00126
          K_ULONG ComputeCurrentTicks(K_USHORT usCount_,
      K_ULONG ulEpoch_);
00127
00128
          K_ULONG m_ulCumulative;
00129
          K_ULONG m_ulCurrentIteration;
          K_USHORT m_usInitial;
00130
00131
          K ULONG m ulInitialEpoch:
00132
          K_USHORT m_usIterations;
00133
          K_UCHAR m_bActive;
00134 };
00135
00136 #endif // KERNEL_USE_PROFILE
00137
00138 #endif
```

15.75 /home/moslevin/Project/R1/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.

15.75.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

15.76 quantum.h

```
00001 /*===
00002
00003
00004
00005
                 1.11
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"
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 K_UCHAR m_bActive;
00102
          static K_UCHAR m_bInTimer;
00103 };
00104
00105 #endif //KERNEL_USE_QUANTUM
00106
00107 #endif
```

15.77 /home/moslevin/Project/R1/kernel/public/scheduler.h File Reference

Thread scheduler function declarations.

15.78 scheduler.h 181

```
#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_pstNext

Pointer to the currently-chosen next-running thread.

Thread * g pstCurrent

Pointer to the currently-running thread.

15.77.1 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 s 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.

15.78 scheduler.h

```
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_pstNext;
00054 extern Thread *g_pstCurrent;
00055
00056 #define NUM PRIORITIES
                                          (8)
00057 //-
00058
00062 class Scheduler
00063 {
00064 public:
00070
         static void Init();
00079
         static void Schedule();
08000
00088
         static void Add(Thread *pclThread_);
00089
00098
         static void Remove(Thread *pclThread);
00099
00112
          static K_BOOL SetScheduler(K_BOOL bEnable_);
00113
00119
          static Thread *GetCurrentThread() { return g_pstCurrent; }
00120
00127
          static volatile Thread *GetNextThread() { return g_pstNext; }
00128
00137
          static ThreadList *GetThreadList(K_UCHAR ucPriority_) {    return &
     m_aclPriorities[ucPriority_]; }
00138
00145
          static ThreadList *GetStopList() { return &m_clStopList; }
00146
00155
          static K UCHAR IsEnabled() { return m bEnabled; }
00156
          static void QueueScheduler() { m_bQueuedSchedule = true; }
00164
00165 private:
          static K_BOOL m_bEnabled;
00167
00168
          static K_BOOL m_bQueuedSchedule;
00170
00171
00173
          static ThreadList m_clStopList;
00174
         static ThreadList m_aclPriorities[NUM_PRIORITIES];
00176
00177
00179
          static K_UCHAR m_ucPriFlag;
00180 };
00181 #endif
00182
```

15.79 /home/moslevin/Project/R1/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"
```

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.

15.80 thread.h 183

Typedefs

typedef void(* ThreadEntry_t)(void *pvArg_)
 Function pointer type used for thread entrypoint functions.

Enumerations

· enum ThreadState t

Enumeration representing the different states a thread can exist in.

15.79.1 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.

15.80 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
00047 //--
00051 typedef void (*ThreadEntry_t)(void *pvArg_);
00052
00053 //---
00057 typedef enum
00058 {
00059
           THREAD_STATE_EXIT = 0,
00060
           THREAD_STATE_READY,
00061
           THREAD_STATE_BLOCKED,
00062
           THREAD_STATE_STOP,
00063 //--
          THREAD_STATES
00064
00065 } ThreadState_t;
00066
00067 //--
00071 class Thread : public LinkListNode
00072 {
00073 public:
00093
          void Init(K_WORD *paucStack_
00094
                      K_USHORT usStackSize_,
```

```
00095
                    K_UCHAR ucPriority_,
00096
                    ThreadEntry_t pfEntryPoint_,
00097
                    void *pvArg_ );
00098
          void Start();
00106
00107
00108
00115
          void Stop();
00116
00117 #if KERNEL USE THREADNAME
00118
00127
          void SetName(const K CHAR *szName) { m szName = szName ; }
00128
00135
          const K_CHAR* GetName() { return m_szName; }
00136 #endif
00137
          ThreadList *GetOwner(void) { return m_pclOwner; }
00146
00147
00155
          ThreadList *GetCurrent(void) { return m_pclCurrent; }
00156
00165
          K_UCHAR GetPriority(void) { return m_ucPriority; }
00166
00174
          K_UCHAR GetCurPriority(void) { return m_ucCurPriority; }
00175
00176 #if KERNEL_USE_QUANTUM
00177
00184
          void SetQuantum( K_USHORT usQuantum_ ) { m_usQuantum = usQuantum_; }
00185
00193
          K_USHORT GetQuantum(void) { return m_usQuantum; }
00194 #endif
00195
00203
          void SetCurrent( ThreadList *pclNewList_ ) {
      m_pclCurrent = pclNewList_; }
00204
00212
          void SetOwner( ThreadList *pclNewList_ ) { m_pclOwner = pclNewList_; }
00213
00214
          void SetPriority(K_UCHAR ucPriority_);
00228
00238
          void InheritPriority(K_UCHAR ucPriority_);
00239
00240 #if KERNEL USE DYNAMIC THREADS
00241
00252
          void Exit();
00253 #endif
00254
00255 #if KERNEL_USE_SLEEP
00256
00264
          static void Sleep (K ULONG ulTimeMs );
00265
00274
          static void USleep(K_ULONG ulTimeUs_);
00275 #endif
00276
00284
          static void Yield (void);
00285
00293
          void SetID( K UCHAR ucID ) { m ucThreadID = ucID ; }
00294
00302
          K_UCHAR GetID() { return m_ucThreadID; }
00303
00304
00317
          K USHORT GetStackSlack():
00318
00319 #if KERNEL_USE_EVENTFLAG
00320
00327
          K_USHORT GetEventFlagMask() { return m_usFlagMask; }
00328
00333
          void SetEventFlagMask(K_USHORT usMask_) {
      m_usFlagMask = usMask_; }
00334
00340
          void SetEventFlagMode(EventFlagOperation_t eMode_ ) {
      m_eFlagMode = eMode_; }
00341
00346
          EventFlagOperation_t GetEventFlagMode() { return
      m_eFlagMode; }
00347 #endif
00348
00349 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00350
00353
          Timer *GetTimer();
00354 #endif
00355 #if KERNEL USE TIMEOUTS
00356
00364
          void SetExpired( K_BOOL bExpired_ );
00365
00372
         K_BOOL GetExpired();
00373 #endif
00374
```

15.80 thread.h 185

```
00375 #if KERNEL_USE_IDLE_FUNC
00376
00381
         void InitIdle();
00382 #endif
00383
          ThreadState_t GetState()
                                                 { return
00390
     m_eState; }
00391
00399
          void SetState( ThreadState_t eState_ ) { m_eState = eState_; }
00400
00401
         friend class ThreadPort:
00402
00403 private:
00411
         static void ContextSwitchSWI (void);
00412
00417
          void SetPriorityBase(K_UCHAR ucPriority_);
00418
00420
         K WORD *m pwStackTop;
00421
00423
         K_WORD *m_pwStack;
00424
00426
          K_UCHAR m_ucThreadID;
00427
00429
          K UCHAR m ucPriority;
00430
00432
          K_UCHAR m_ucCurPriority;
00433
00435
          ThreadState_t m_eState;
00436
00437 #if KERNEL USE THREADNAME
00438
         const K_CHAR *m_szName;
00440 #endif
00441
00443
          K_USHORT m_usStackSize;
00444
         ThreadList *m_pclCurrent;
00446
00447
          ThreadList *m_pclOwner;
00450
00452
          ThreadEntry_t m_pfEntryPoint;
00453
00455
         void *m_pvArg;
00456
00457 #if KERNEL_USE_QUANTUM
00458
         K_USHORT m_usQuantum;
00460 #endif
00461
00462 #if KERNEL_USE_EVENTFLAG
         K_USHORT m_usFlagMask;
00463
00465
00467
         EventFlagOperation_t m_eFlagMode;
00468 #endif
00469
00470 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00471
         Timer m_clTimer;
00473 #endif
00474 #if KERNEL_USE_TIMEOUTS
00475
         K_BOOL m_bExpired;
00477 #endif
00478
00479 };
00480
00481 #if KERNEL_USE_IDLE_FUNC
00482 //-
00494 typedef struct
00495 {
00496
          LinkListNode *next;
          LinkListNode *prev;
00497
00498
         K_WORD *m_pwStackTop;
00501
00503
          K_WORD *m_pwStack;
00504
         K_UCHAR m_ucThreadID;
00506
00507
00509
          K_UCHAR m_ucPriority;
00510
00512
          K_UCHAR m_ucCurPriority;
00513
00515
          ThreadState_t m_eState;
00516
00517 #if KERNEL_USE_THREADNAME
00518
         const K_CHAR *m_szName;
00520 #endif
00521
00522 } FakeThread t;
00523 #endif
```

```
00524
00525 #endif
```

15.81 /home/moslevin/Project/R1/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.

15.81.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

15.82 threadlist.h

```
00001 /*----
00002
00003
00004
00005
00006 1
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 CircularLinkList
00035 {
00036 public:
00040
         ThreadList() { m_ucPriority = 0; m_pucFlag = NULL; }
00041
00049
         void SetPriority(K_UCHAR ucPriority_);
00050
00059
         void SetFlagPointer(K_UCHAR *pucFlag_);
00060
00068
         void Add(LinkListNode *node_);
00069
00083
         void Add(LinkListNode *node_, K_UCHAR *pucFlag_,
     K_UCHAR ucPriority_);
00084
00092
         void Remove(LinkListNode *node_);
00093
00101
         Thread *HighestWaiter();
00102 private:
00103
00105
         K_UCHAR m_ucPriority;
00106
         K_UCHAR *m_pucFlag;
00108
00109 };
00110
00111 #endif
```

00112

15.83 /home/moslevin/Project/R1/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.

15.83.1 Detailed Description

Timer object declarations.

Definition in file timer.h.

15.83.2 Macro Definition Documentation

15.83.2.1 #define TIMERLIST_FLAG_EXPIRED (0x08)

Timer is actually expired.

Definition at line 36 of file timer.h.

15.83.3 Typedef Documentation

15.83.3.1 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.

pclOwner_ 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 94 of file timer.h.

15.84 timer.h

```
00001 /
00002
00003
00004 |
                 1.11
00005 I
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
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
                                               (0 \times 0.4)
00036 #define TIMERLIST_FLAG_EXPIRED
                                               (0x08)
00037
00038 //--
00039 #if KERNEL_TIMERS_TICKLESS
00040
00041 //---
00042 #define MAX_TIMER_TICKS
                                              (0x7FFFFFFF)
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 us...
00048
         Max time, SECONDS_TO_TICKS: 68719s
Max time, MSECONDS_TO_TICKS: 6871.9s
00049
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)
                                               ((((K_ULONG)x) * TIMER_FREQ))
00059 #define MSECONDS_TO_TICKS(x)
                                               (((((((K_ULONG)x) * (TIMER_FREQ/100)) + 5) / 10))
                                               00060 #define USECONDS_TO_TICKS(x)
00061
00062 //--
00063 #define MIN_TICKS
00064 //--
00065
00066 #else
00067 //---
00068 // Tick-based timers, assuming 1khz tick rate
00069 #define MAX_TIMER_TICKS
                                              (0x7FFFFFFF)
```

15.84 timer.h 189

```
00072 // add time because we don't know how far in an epoch we are when a call is made.
                                     (((K_ULONG)(x) * 1000) + 1)
00073 #define SECONDS_TO_TICKS(x)
00074 #define MSECONDS_TO_TICKS(x)
                                               ((K ULONG)(x + 1))
00075 #define USECONDS_TO_TICKS(x)
                                               (((K_ULONG)(x + 999)) / 1000)
00076
00077 //---
00078 #define MIN_TICKS
                                                (1)
00079 //---
08000
00081 #endif // KERNEL_TIMERS_TICKLESS
00082
00083 //----
00094 typedef void (*TimerCallback_t)(Thread *pclOwner_, void *pvData_);
00095
00096 //----
00097 class TimerList:
00098 class TimerScheduler;
00099 class Quantum;
00105 class Timer : public LinkListNode
00106 {
00107 public:
00111
          Timer() { }
00112
          void Init() { ClearNode(); m_ulInterval = 0;
00116
      m_ulTimerTolerance = 0; m_ulTimeLeft = 0;
      m_ucFlags = 0; }
00117
00127
          void Start( K_BOOL bRepeat_, K_ULONG ulIntervalMs_,
      TimerCallback_t pfCallback_, void *pvData_ );
00128
      void Start( K_BOOL bRepeat_, K_ULONG ulIntervalMs_,
K_ULONG ulToleranceMs_, TimerCallback_t pfCallback_, void *pvData_ );
00140
00141
00146
          void Stop();
00147
00157
          void SetFlags (K UCHAR ucFlags ) { m ucFlags = ucFlags ; }
00158
00166
          void SetCallback( TimerCallback_t pfCallback_) {
     m_pfCallback = pfCallback_; }
00167
00175
          void SetData( void *pvData_ ) { m_pvData = pvData_; }
00176
00185
          void SetOwner( Thread *pclOwner_) { m_pclOwner = pclOwner_; }
00186
00194
          void SetIntervalTicks(K_ULONG ulTicks_);
00195
          void SetIntervalSeconds(K_ULONG ulSeconds_);
00203
00204
00205
00206
          K_ULONG GetInterval()
                                  { return m_ulInterval; }
00207
00215
          void SetIntervalMSeconds(K_ULONG ulMSeconds_);
00216
00224
          void SetIntervalUSeconds(K ULONG uluSeconds);
00225
00235
          void SetTolerance(K_ULONG ulTicks_);
00236
00237 private:
00238
00239
          friend class TimerList:
00240
00242
          K_UCHAR m_ucFlags;
00243
00245
          TimerCallback_t m_pfCallback;
00246
          K_ULONG m_ulInterval;
00248
00249
00251
          K_ULONG m_ulTimeLeft;
00252
00254
          K_ULONG m_ulTimerTolerance;
00255
00257
          Thread *m_pclOwner;
00258
00260
          void
                 *m pvData;
00261 };
00262
00263 #endif // KERNEL_USE_TIMERS
00264
00265 #endif
```

15.85 /home/moslevin/Project/R1/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.

15.85.1 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.

15.86 timerlist.h

```
00001
00002
00003
00004
00005
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"
00030 #include "timer.h"
00031 #if KERNEL_USE_TIMERS
00032
00033 //
00037 class TimerList : public DoubleLinkList
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
          K_ULONG m_ulNextWakeup;
00077
          K_UCHAR m_bTimerActive;
00080 };
00081
00082 #endif // KERNEL_USE_TIMERS
00083
00084 #endif
```

15.87 /home/moslevin/Project/R1/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.

15.87.1 Detailed Description

Timer scheduler declarations.

Definition in file timerscheduler.h.

15.88 timerscheduler.h

```
00001 /*
00002
00003
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00021 #ifndef __TIMERSCHEDULER_H_
00022 #define __TIMERSCHEDULER_H_
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026
00027 #include "ll.h"
00028 #include "timer.h"
00029 #include "timerlist.h"
00030
00031 #if KERNEL_USE_TIMERS
00032
00033 //---
00038 class TimerScheduler
00039 {
00040 public:
00047
         static void Init() { m_clTimerList.Init(); }
00048
         static void Add(Timer *pclListNode)
00057
00058
            {m_clTimerList.Add(pclListNode_);
00059
00068
         static void Remove(Timer *pclListNode_)
00069
             {m_clTimerList.Remove(pclListNode_); }
00070
00079
         static void Process() {m clTimerList.Process();}
00080 private:
00081
00083
          static TimerList m_clTimerList;
00084 };
00085
00086 #endif //KERNEL USE TIMERS
00087
00088 #endif //__TIMERSCHEDULER_H_
00089
```

15.89 /home/moslevin/Project/R1/kernel/public/tracebuffer.h File Reference

Kernel trace buffer class declaration.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "writebuf16.h"
```

15.89.1 Detailed Description

Kernel trace buffer class declaration. 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.

Definition in file tracebuffer.h.

15.90 tracebuffer.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 __TRACEBUFFER_H_
00025 #define __TRACEBUFFER_H_
00026
00027 #include "kerneltypes.h"
00028 #include "mark3cfg.h"
00029 #include "writebuf16.h"
00030
00031 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00032
00033 #define TRACE_BUFFER_SIZE
00034
00038 class TraceBuffer
00039 {
00040 public:
          static void Init();
00047
00055
           static K_USHORT Increment();
00056
           static void Write( K_USHORT *pusData_, K_USHORT usSize_ );
00065
00066
           void SetCallback( WriteBufferCallback pfCallback_ )
00076
               { m_clBuffer.SetCallback( pfCallback_ ); }
00077 private:
00078
00079
           static WriteBuffer16 m_clBuffer;
00080
          static volatile K USHORT m usIndex;
00081
          static K_USHORT m_ausBuffer[ (TRACE_BUFFER_SIZE / sizeof( K_USHORT )) ];
00082 };
00083
00084 #endif //KERNEL_USE_DEBUG
00085
00086 #endif
```

15.91 /home/moslevin/Project/R1/kernel/public/writebuf16.h File Reference

Thread-safe circular buffer implementation with 16-bit elements.

15.92 writebuf16.h 193

```
#include "kerneltypes.h"
#include "mark3cfg.h"
```

15.91.1 Detailed Description

Thread-safe circular buffer implementation with 16-bit elements.

Definition in file writebuf16.h.

15.92 writebuf16.h

```
00002
00003
00004
00005 1
00006 1
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 __WRITEBUF16_H_
00021 #define __WRITEBUF16_H_
00022
00023 #include "kerneltypes.h" 00024 #include "mark3cfg.h"
00025
00026 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00032 typedef void (*WriteBufferCallback)( K_USHORT *pusData_, K_USHORT usSize_);
00033
00040 class WriteBuffer16
00041 {
00042 public:
00053
          void SetBuffers( K_USHORT *pusData_, K_USHORT usSize_ )
00054
00055
               m_pusData = pusData_;
00056
              m_usSize = usSize_;
00057
              m_usHead = 0;
              m_usTail = 0;
00058
00059
00060
00072
          void SetCallback( WriteBufferCallback pfCallback_ )
00073
              { m_pfCallback = pfCallback_; }
00074
00083
          void WriteData( K_USHORT *pusBuf_, K_USHORT usLen_ );
00084
00094
          void WriteVector( K_USHORT **ppusBuf_, K_USHORT *pusLen_,
      K_UCHAR ucCount_);
00095
00096 private:
00097
          K_USHORT *m_pusData;
00099
          volatile K_USHORT m_usSize;
00100
          volatile K_USHORT m_usHead;
00101
          volatile K_USHORT m_usTail;
00102
00103
          WriteBufferCallback m pfCallback;
00104 };
00105 #endif
00106
00107 #endif
```

15.93 /home/moslevin/Project/R1/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 "kerneldebug.h"
#include "kernelaware.h"
```

Macros

#define __FILE_ID__ QUANTUM_CPP
 File ID used in kernel trace calls.

Functions

static void QuantumCallback (Thread *pclThread_, void *pvData_)
 QuantumCallback.

15.93.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

15.93.2 Function Documentation

```
15.93.2.1 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

pclThread_	Pointer to the thread currently executing
pvData_	Unused in this context.

Definition at line 56 of file quantum.cpp.

15.94 quantum.cpp

```
00001
00002
00003
00004
00005
00006
00007
80000
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"
```

15.94 quantum.cpp 195

```
00027 #include "quantum.h"
00028 #include "kerneldebug.h"
00029 #include "kernelaware.h"
00030 //----
00031 #if defined __FILE_ID__
00032
        #undef ___FILE_ID__
00033 #endif
00034 #define __FILE_ID__
                              QUANTUM_CPP
00035
00036 #if KERNEL_USE_QUANTUM
00037
00038 //---
00039 static volatile K_BOOL bAddQuantumTimer; // Indicates that a timer add is pending
00040
00041 //---
00042 Timer Quantum::m_clQuantumTimer; // The global timernodelist_t object
00043 K_UCHAR Quantum::m_bActive;
00044 K_UCHAR Quantum::m_bInTimer;
00045 //-
00056 static void QuantumCallback(Thread *pclThread_, void *pvData_)
00057 {
00058
          // Validate thread pointer, check that source/destination match (it's
         // in its real priority list). Also check that this thread was part of // the highest-running priority level.
00059
00060
          if (pclThread_->GetPriority() >= Scheduler::GetCurrentThread()->
00061
     GetPriority())
00062
       {
00063
             if (pclThread_->GetCurrent()->GetHead() != pclThread_->
     GetCurrent()->GetTail() )
00064
       {
00065
                  bAddOuantumTimer = true;
00066
                  pclThread_->GetCurrent()->PivotForward();
00067
00068
         }
00069 }
00070
00071 //-
00072 void Quantum::SetTimer(Thread *pclThread_)
00073 {
          m_clQuantumTimer.SetIntervalMSeconds(pclThread_->
00074
     GetQuantum());
00075
         m_clQuantumTimer.SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00076
          m clOuantumTimer.SetData(NULL):
00077
         m_clQuantumTimer.SetCallback((TimerCallback_t)
     QuantumCallback);
00078
         m_clQuantumTimer.SetOwner(pclThread_);
00079 }
08000
00081 //---
00082 void Quantum::AddThread(Thread *pclThread_)
00083 {
00084
          if (m_bActive
00085 #if KERNEL_USE_IDLE_FUNC
00086
                 || (pclThread_ == Kernel::GetIdleThread())
00087 #endif
00088
            )
00089
00090
             return;
00091
         }
00092
          // If this is called from the timer callback, queue a timer {\tt add}\dots
00093
00094
          if (m_bInTimer)
00095
         {
00096
              bAddQuantumTimer = true;
00097
00098
          }
00099
00100
          // If this isn't the only thread in the list.
          if ( pclThread_->GetCurrent() ->GetHead() !=
00101
00102
                pclThread_->GetCurrent()->GetTail() )
00103
00104
              Quantum::SetTimer(pclThread_);
00105
              TimerScheduler::Add(&m_clQuantumTimer);
00106
              m_bActive = 1;
00107
          }
00108 }
00109
00110 //---
00111 void Quantum::RemoveThread(void)
00112 {
00113
          if (!m_bActive)
00114
          {
00115
              return;
00116
          }
00117
          // Cancel the current timer
00118
00119
          TimerScheduler::Remove(&m_clQuantumTimer);
```

```
m_bActive = 0;
00121 }
00122
00123 //---
00124 void Quantum:: UpdateTimer (void)
00125 {
          // If we have to re-add the quantum timer (more than 2 threads at the
          // high-priority level...)
00128
         if (bAddQuantumTimer)
00129
              // Trigger a thread yield - this will also re-schedule the
00130
              // thread \starand\star reset the round-robin scheduler.
00131
00132
              Thread::Yield();
00133
              bAddQuantumTimer = false;
00134
00135 }
00136
00137 #endif //KERNEL_USE_QUANTUM
```

15.95 /home/moslevin/Project/R1/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 "kerneldebug.h"
```

Macros

• #define __FILE_ID__ SCHEDULER_CPP

File ID used in kernel trace calls.

Variables

volatile Thread * g_pstNext

Pointer to the currently-chosen next-running thread.

Thread * g_pstCurrent

Pointer to the currently-running thread.

This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

15.95.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

15.95.2 Variable Documentation

```
15.95.2.1 const K_UCHAR aucCLZ[16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,3,3,} [static]
```

This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

15.96 scheduler.cpp 197

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 us to quickly identify whether or not subsequent 4-bit LUT operations are required to complete the scheduling process.

Definition at line 56 of file scheduler.cpp.

15.96 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 #include "kerneldebug.h"
00029 //----
00030 #if defined __FILE_ID_
00031
         #undef ___FILE_ID_
00033 #define __FILE_ID__
                               SCHEDULER_CPP
00034
00035 //---
00036 volatile Thread *g_pstNext;
00037 Thread *g_pstCurrent;
00038
00039 //---
00040 K_BOOL Scheduler::m_bEnabled;
00041 K_BOOL Scheduler::m_bQueuedSchedule;
00042
00043 ThreadList Scheduler::m clStopList;
00044 ThreadList Scheduler::m_aclPriorities[
      NUM_PRIORITIES];
00045 K_UCHAR Scheduler::m_ucPriFlag;
00046
00047 //----
00056 static const K UCHAR aucCLZ[16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,3};
00057
00059 void Scheduler::Init()
00060 {
          m_ucPriFlag = 0;
for (int i = 0; i < NUM_PRIORITIES; i++)</pre>
00061
00062
00063
00064
               m_aclPriorities[i].SetPriority(i);
              m_aclPriorities[i].SetFlagPointer(&
00065
      m_ucPriFlag);
00066
00067
          m bOueuedSchedule = false;
00068 }
00069
00070 //-
00071 void Scheduler::Schedule()
00072 {
          K UCHAR ucPri = 0:
00073
00074
00075
           // Figure out what priority level has ready tasks (8 priorities max)
00076
          // To do this, we apply our current active-thread bitmap (m_ucPriFlag)
00077
           \ensuremath{//} and perform a CLZ on the upper four bits. If no tasks are found
00078
          // in the higher priority bits, search the lower priority bits. This
00079
          \ensuremath{//} also assumes that we always have the idle thread ready-to-run in
          // priority level zero.
00080
00081
          ucPri = aucCLZ[m_ucPriFlag >> 4 ];
00082
          if (ucPri == 0xFF)
00083
          {
00084
              ucPri = aucCLZ[m_ucPriFlag & 0x0F];
00085
00086
          else
00087
00088
              ucPri += 4;
```

```
00089
00090
00091 #if KERNEL_USE_IDLE_FUNC
       if (ucPri == 0xFF)
00092
00093
00094
              // There aren't any active threads at all - set q_pstNext to IDLE
              g_pstNext = Kernel::GetIdleThread();
00096
00097
          else
00098 #endif
00099
        {
              // Get the thread node at this priority.
00100
              g_pstNext = (Thread*)( m_aclPriorities[ucPri].GetHead() );
00101
00102
00103
          KERNEL_TRACE_1( STR_SCHEDULE_1, (K_USHORT)((Thread*)g_pstNext)->GetID() );
00104
00105 }
00106
00108 void Scheduler::Add(Thread *pclThread_)
00109 {
00110
          m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00111 }
00112
00113 //-
00114 void Scheduler::Remove(Thread *pclThread_)
00115 {
00116
          m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00117 }
00118
00119 //-
00120 K_BOOL Scheduler::SetScheduler(K_BOOL bEnable_)
00121 {
00122
          K_BOOL bRet ;
00123
          CS_ENTER();
00124
          bRet = m_bEnabled;
          m_bEnabled = bEnable_;
00125
         // If there was a queued scheduler evevent, dequeue and trigger an
00127
         // immediate Yield
00128
          if (m_bEnabled && m_bQueuedSchedule)
00129
              m bQueuedSchedule = false;
00130
00131
              Thread::Yield();
00132
00133
          CS_EXIT();
00134
          return bRet;
00135 }
```

15.97 /home/moslevin/Project/R1/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 "kerneldebug.h"
```

Macros

• #define __FILE_ID__ THREAD_CPP

File ID used in kernel trace calls.

15.98 thread.cpp 199

Functions

static void ThreadSleepCallback (Thread *pclOwner_, void *pvData_)

This callback is used to wake up a thread once the interval has expired.

15.97.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

15.98 thread.cpp

```
00001 /*-----
00002
00003
00004
00005
00006 1
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 #include "kerneldebug.h"
00033
00034 //---
00035 #if defined __FILE_ID__
00036 #undef __FILE_ID__
00037 #endif
00038 #define __FILE_ID__
                              THREAD CPP
00039
00040 //---
00041 void Thread::Init( K_WORD *pwStack_,
00042
                      K_USHORT usStackSize_,
00043
                      K_UCHAR ucPriority_,
00044
                       ThreadEntry_t pfEntryPoint_,
00045
                      void *pvArg_ )
00046 {
00047
          static K_UCHAR ucThreadID = 0;
00048
00049
          KERNEL_ASSERT ( pwStack_
00050
          KERNEL_ASSERT( pfEntryPoint_ );
00051
00052
          ClearNode();
00053
00054
          m_ucThreadID = ucThreadID++;
00055
00056
          KERNEL_TRACE_1( STR_STACK_SIZE_1, usStackSize_ );
          KERNEL_TRACE_1( STR_PRIORITY_1, (K_UCHAR)ucPriority_ );
KERNEL_TRACE_1( STR_THREAD_ID_1, (K_USHORT)
00057
00058
      m ucThreadID );
00059
          KERNEL_TRACE_1( STR_ENTRYPOINT_1, (K_USHORT)pfEntryPoint_ );
00060
00061
          \ensuremath{//} Initialize the thread parameters to their initial values.
00062
          m_pwStack = pwStack_;
          m_pwStackTop = TOP_OF_STACK(pwStack_, usStackSize_);
00063
00064
00065
          m_usStackSize = usStackSize_;
00066
00067 #if KERNEL_USE_QUANTUM
00068
          m_usQuantum = THREAD_QUANTUM_DEFAULT;
00069 #endif
00070
00071
          m_ucPriority = ucPriority_ ;
00072
          m_ucCurPriority = m_ucPriority;
```

```
m_pfEntryPoint = pfEntryPoint_;
         m_pvArg = pvArg_;
m_eState = THREAD_STATE_STOP;
00074
00075
00076
00077 #if KERNEL USE THREADNAME
00078
         m szName = NULL:
00079 #endif
00080 #if KERNEL_USE_TIMERS
00081
         m_clTimer.Init();
00082 #endif
00083
00084
          // Call CPU-specific stack initialization
00085
         ThreadPort::InitStack(this);
00086
00087
          // Add to the global "stop" list.
         CS_ENTER();
00088
         m_pclOwner = Scheduler::GetThreadList(
00089
     m_ucPriority);
00090
        m_pclCurrent = Scheduler::GetStopList();
00091
          m_pclCurrent->Add(this);
00092
          CS_EXIT();
00093 }
00094
00095 //---
00096 void Thread::Start(void)
00097 {
00098
          // Remove the thread from the scheduler's "stopped" list, and add it
nnnaa
          // to the scheduler's ready list at the proper priority.
00100
         KERNEL_TRACE_1( STR_THREAD_START_1, (K_USHORT)
     m_ucThreadID );
00101
00102
          CS_ENTER();
00103
         Scheduler::GetStopList()->Remove(this);
00104
         Scheduler::Add(this);
00105
         m_pclOwner = Scheduler::GetThreadList(
     m_ucPriority);
        m_pclCurrent = m_pclOwner;
00106
         m_eState = THREAD_STATE_READY;
00107
00108
00109 #if KERNEL_USE_QUANTUM
00110
         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
     GetCurPriority())
00111
       {
00112
              // Deal with the thread Quantum
00113
              Quantum::RemoveThread();
00114
              Quantum::AddThread(this);
00115
00116 #endif
00117
00118
          if (Kernel::IsStarted())
        if (GetCurPriority() >= Scheduler::GetCurrentThread()->
00119
     GetCurPriority())
00121
         {
00122
                  Thread::Yield();
00123
              }
00124
00125
          CS_EXIT();
00126 }
00127
00128 //---
00129 void Thread::Stop()
00130 {
00131
          K_BOOL bReschedule = 0;
00132
00133
         CS_ENTER();
00134
          // If a thread is attempting to stop itself, ensure we call the scheduler
00135
00136
         if (this == Scheduler::GetCurrentThread())
00137
         {
00138
              bReschedule = true;
00139
00140
         // Add this thread to the stop-list (removing it from active scheduling) // Remove the thread from scheduling
00141
00142
00143
          if (m_eState == THREAD_STATE_READY)
00144
         {
00145
              Scheduler::Remove(this);
00146
          else if (m eState == THREAD STATE BLOCKED)
00147
00148
00149
              m_pclCurrent->Remove(this);
00150
00151
00152
         m_pclOwner = Scheduler::GetStopList();
00153
          m_pclCurrent = m_pclOwner;
00154
         m_pclOwner->Add(this);
```

15.98 thread.cpp 201

```
m_eState = THREAD_STATE_STOP;
00156
00157 #if KERNEL_USE_TIMERS
          // Just to be safe - attempt to remove the thread's timer \,
00158
          // from the timer-scheduler (does no harm if it isn't
00159
           // in the timer-list)
00160
          TimerScheduler::Remove(&m_clTimer);
00161
00162 #endif
00163
00164
          CS EXIT();
00165
00166
          if (bReschedule)
00167
          {
00168
               Thread::Yield();
00169
00170 }
00171
00172 #if KERNEL USE DYNAMIC THREADS
00173 //-
00174 void Thread::Exit()
00175 {
00176
          K_BOOL bReschedule = 0;
00177
00178
          KERNEL_TRACE_1( STR_THREAD_EXIT_1, m_ucThreadID );
00179
00180
          CS ENTER();
00181
00182
           // If this thread is the actively-running thread, make sure we run the
          // scheduler again.
if (this == Scheduler::GetCurrentThread())
00183
00184
00185
          {
00186
               bReschedule = 1;
00187
00188
00189
          \ensuremath{//} Remove the thread from scheduling
          if (m_eState == THREAD_STATE_READY)
00190
00191
          {
00192
               Scheduler::Remove(this);
00193
00194
          else if (m_eState == THREAD_STATE_BLOCKED)
00195
00196
              m pclCurrent->Remove(this);
00197
          }
00198
00199
          m_pclCurrent = 0;
00200
          m_pclOwner = 0;
00201
          m_eState = THREAD_STATE_EXIT;
00202
00203
          // We've removed the thread from scheduling, but interrupts might
          // trigger checks against this thread's currently priority before // we get around to scheduling new threads. As a result, set the
00204
00205
00206
           // priority to idle to ensure that we always wind up scheduling
          // new threads.
00207
00208
          m_ucCurPriority = 0;
00209
          m_ucPriority = 0;
00210
00211 #if KERNEL_USE_TIMERS
00212
          // Just to be safe - attempt to remove the thread's timer
           // from the timer-scheduler (does no harm if it isn't
00213
           // in the timer-list)
00214
00215
          TimerScheduler::Remove(&m clTimer);
00216 #endif
00217
00218
          CS_EXIT();
00219
00220
          if (bReschedule)
00221
          {
               // Choose a new "next" thread if we must
00222
00223
               Thread::Yield();
00224
          }
00225 }
00226 #endif
00227
00228 #if KERNEL USE SLEEP
00229 //-
00231 static void ThreadSleepCallback( Thread *pclOwner_, void *pvData_ )
00232 {
00233
          Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00234
          \ensuremath{//} Post the semaphore, which will wake the sleeping thread.
00235
          pclSemaphore->Post();
00236 }
00237
00238 //--
00239 void Thread::Sleep(K_ULONG ulTimeMs_)
00240 {
00241
           Semaphore clSemaphore;
00242
          Timer *pclTimer = g_pstCurrent->GetTimer();
```

```
00243
           // Create a semaphore that this thread will block on
00244
00245
          clSemaphore.Init(0, 1);
00246
00247
          // Create a one-shot timer that will call a callback that posts the
          // semaphore, waking our thread.
pclTimer->Init();
00248
00249
00250
          pclTimer->SetIntervalMSeconds(ulTimeMs_);
00251
          pclTimer->SetCallback(ThreadSleepCallback);
00252
          pclTimer->SetData((void*)&clSemaphore);
00253
          pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00254
00255
          // Add the new timer to the timer scheduler, and block the thread
00256
          TimerScheduler::Add(pclTimer);
00257
          clSemaphore.Pend();
00258 }
00259
00260 //--
00261 void Thread::USleep(K_ULONG ulTimeUs_)
00262 {
           Semaphore clSemaphore;
00263
00264
          Timer *pclTimer = g_pstCurrent->GetTimer();
00265
          \ensuremath{//} Create a semaphore that this thread will block on
00266
00267
          clSemaphore.Init(0, 1);
00268
00269
          // Create a one-shot timer that will call a callback that posts the
00270
          // semaphore, waking our thread.
00271
          pclTimer->Init();
          pclTimer->SetIntervalUSeconds(ulTimeUs_);
00272
          pclTimer->SetCallback(ThreadSleepCallback);
00273
00274
          pclTimer->SetData((void*)&clSemaphore);
00275
          pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00276
00277
           // Add the new timer to the timer scheduler, and block the thread
          TimerScheduler::Add(pclTimer);
00278
00279
          clSemaphore.Pend();
00280 }
00281 #endif // KERNEL_USE_SLEEP
00282
00283 //-
00284 K_USHORT Thread::GetStackSlack()
00285 {
00286
          K_USHORT usCount = 0;
00287
00288
          CS_ENTER();
00289
          for (usCount = 0; usCount < m_usStackSize; usCount++)</pre>
00291
00292
00293
              if (m pwStack[usCount] != 0xFF)
00294
              {
00295
00296
              }
00297
          }
00298
00299
          CS_EXIT();
00300
00301
          return usCount:
00302 }
00303
00304 //----
00305 void Thread::Yield()
00306 {
00307
          CS_ENTER();
00308
          // Run the scheduler
00309
          if (Scheduler::IsEnabled())
00310
00311
              Scheduler::Schedule();
00312
              \ensuremath{//} Only switch contexts if the new task is different than the old task
00314
              if (Scheduler::GetCurrentThread() !=
     Scheduler::GetNextThread())
00315
00316 #if KERNEL_USE_QUANTUM
                  // new thread scheduled. Stop current quantum timer (if it exists), // and restart it for the new thread (if required).
00317
00318
                   Quantum::RemoveThread();
00319
00320
                  Quantum::AddThread((Thread*)g_pstNext);
00321 #endif
00322
                   Thread::ContextSwitchSWI():
00323
              }
00324
          }
          else
00325
00326
00327
              Scheduler::QueueScheduler();
          }
00328
00329
```

15.98 thread.cpp 203

```
00330
         CS_EXIT();
00331 }
00332
00333 //----
00334 void Thread::SetPriorityBase(K_UCHAR ucPriority_)
00335 {
           GetCurrent()->Remove(this);
00337
00338
          SetCurrent (Scheduler::GetThreadList(
     m_ucPriority));
00339
00340
          GetCurrent()->Add(this):
00341 }
00342
00343 //---
00344 void Thread::SetPriority(K_UCHAR ucPriority_)
00345 {
00346
         K BOOL bSchedule = 0;
00348
         CS_ENTER();
00349
          // If this is the currently running thread, it's a good idea to reschedule
         // Or, if the new priority is a higher priority than the current thread's.
00350
          if ((g_pstCurrent == this) || (ucPriority_ > g_pstCurrent->
00351
     GetPriority()))
00352
         {
00353
             bSchedule = 1;
00354
00355
          Scheduler::Remove(this);
00356
         CS_EXIT();
00357
00358
          m_ucCurPriority = ucPriority_;
00359
         m_ucPriority = ucPriority_;
00360
          CS_ENTER();
00361
00362
          Scheduler::Add(this);
00363
          CS_EXIT();
00364
00365
          if (bSchedule)
00366
          {
00367
              if (Scheduler::IsEnabled())
00368
              {
00369
                  CS ENTER();
00370
                  Scheduler::Schedule();
00371
         #if KERNEL_USE_QUANTUM
00372
                 // new thread scheduled. Stop current quantum timer (if it exists),
00373
                  // and restart it for the new thread (if required).
00374
                  Quantum::RemoveThread();
00375
                  Quantum::AddThread((Thread*)g_pstNext);
00376
         #endif
00377
                  CS_EXIT();
00378
                  Thread::ContextSwitchSWI();
00379
00380
              else
00381
             {
00382
                  Scheduler::OueueScheduler();
00383
              }
00384
00385 }
00386
00387 //---
00388 void Thread::InheritPriority(K_UCHAR ucPriority_)
00389 {
00390
          SetOwner(Scheduler::GetThreadList(ucPriority_));
00391
         m_ucCurPriority = ucPriority_;
00392 }
00393
00394 //---
00395 void Thread::ContextSwitchSWI()
00396 {
00397
          // Call the context switch interrupt if the scheduler is enabled.
00398
         if (Scheduler::IsEnabled() == 1)
00399
     KERNEL_TRACE_1( STR_CONTEXT_SWITCH_1, (K_USHORT)((
Thread*)g_pstNext)->GetID() );
00400
            KernelSWI::Trigger();
00401
00402
00403 }
00404
00405 #if KERNEL_USE_TIMEOUTS
00406 //----
00407 Timer *Thread::GetTimer()
                                                    { return &
     m_clTimer; }
00408
00409 //---
00410 void Thread::SetExpired( K_BOOL bExpired_ ) {
      m_bExpired = bExpired_; }
00411
```

```
00413 K_BOOL Thread::GetExpired()
                                                               { return
      m_bExpired; }
00414 #endif
00415
00416 #if KERNEL_USE_IDLE_FUNC
00417 //--
00418 void Thread::InitIdle( void )
00419 {
00420
           ClearNode();
00421
00422
           m_ucPriority = 0;
           m_ucCurPriority = 0;
00423
00424
           m_pfEntryPoint = 0;
00425
           m_pvArg = 0;
           m_ucThreadID = 255;
00426
00427 m_eState = THREAD_STATE_READY;
00428 #if KERNEL_USE_THREADNAME
00429 m_szName = "IDLE";
00430 #endif
00431 }
00432 #endif
```

15.99 /home/moslevin/Project/R1/kernel/threadlist.cpp File Reference

Thread linked-list definitions.

```
#include "kerneltypes.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
#include "kerneldebug.h"
```

Macros

#define __FILE_ID__ THREADLIST_CPP
 File ID used in kernel trace calls.

15.99.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

15.100 threadlist.cpp

```
00001 /
00002
00003
00004
00005
00006 1
00007
80000
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 "11.h"
00024 #include "threadlist.h"
00025 #include "thread.h"
00026 #include "kerneldebug.h"
00027 //---
00028 #if defined __FILE_ID_
00029 #undef __FILE_ID_
```

15.100 threadlist.cpp 205

```
00030 #endif
00031 #define __FILE_ID__ THREADLIST_CPP
00032
00033 //----
00034 void ThreadList::SetPriority(K_UCHAR ucPriority_)
00035 {
          m_ucPriority = ucPriority_;
00037 }
00038
00039 //---
00040 void ThreadList::SetFlagPointer(K_UCHAR *pucFlag_)
00041 {
00042
          m pucFlag = pucFlag;
00043 }
00044
00045 //---
00046 void ThreadList::Add(LinkListNode *node_) {
00047
         CircularLinkList::Add(node_);
CircularLinkList::PivotForward();
00049
00050
          // We've specified a bitmap for this threadlist
00051
          if (m_pucFlag)
00052
          {
              // Set the flag for this priority level
*m_pucFlag |= (1 << m_ucPriority);</pre>
00053
00054
00055
00056 }
00057
00058 //----
00059 void ThreadList::Add(LinkListNode *node_, K_UCHAR *pucFlag_,
      K UCHAR ucPriority ) {
         // Set the threadlist's priority level, flag pointer, and then add the
00061
          // thread to the threadlist
00062
          SetPriority(ucPriority_);
00063
          SetFlagPointer(pucFlag_);
00064
          Add (node_);
00065 }
00066
00067 //---
00068 void ThreadList::Remove(LinkListNode *node_) {
00069
          \ensuremath{//} Remove the thread from the list
00070
          CircularLinkList::Remove(node_);
00071
00072
          // If the list is empty...
00073
          if (!m_pstHead)
00074
          {
00075
              // Clear the bit in the bitmap at this priority level
00076
              if (m_pucFlag)
00077
              {
00078
                   *m_pucFlag &= ~(1 << m_ucPriority);</pre>
00079
              }
08000
          }
00081 }
00082
00083 //----
00084 Thread *ThreadList::HighestWaiter()
00085 {
00086
           Thread *pclTemp = static_cast<Thread*>(GetHead());
00087
          Thread *pclChosen = pclTemp;
00088
00089
          K UCHAR ucMaxPri = 0:
00090
00091
          // Go through the list, return the highest-priority thread in this list.
00092
          while(1)
00093
00094
              // Compare against current \max-priority thread
00095
              if (pclTemp->GetPriority() >= ucMaxPri)
00096
              {
00097
                  ucMaxPri = pclTemp->GetPriority();
00098
                  pclChosen = pclTemp;
00099
00100
00101
              \ensuremath{//} Break out if this is the last thread in the list
00102
              if (pclTemp == static_cast<Thread*>(GetTail()))
00103
              {
00104
                  break:
00105
00106
00107
              pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00108
          return pclChosen;
00109
00110 }
```

15.101 /home/moslevin/Project/R1/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 "kerneldebug.h"
#include "quantum.h"
```

Macros

#define __FILE_ID__ TIMER_CPP
 File ID used in kernel trace calls.

15.101.1 Detailed Description

Timer implementations.

Definition in file timer.cpp.

15.102 timer.cpp

```
00001 /
00002
00003
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
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 "kerneldebug.h"
00031 #include "quantum.h"
00032
00033 //----
00034 #if defined __FILE_ID_
         #undef __FILE_ID__
00035
00036 #endif
00037 #define __FILE_ID__
00038
00039 #if KERNEL_USE_TIMERS
00040
00041 //--
00042 void Timer::Start ( K_BOOL bRepeat_, K_ULONG ulIntervalMs_,
      TimerCallback_t pfCallback_, void *pvData_ )
00043 {
00044
         SetIntervalMSeconds(ulIntervalMs_);
00045
         m ulTimerTolerance = 0;
         m_pfCallback = pfCallback_;
00046
00047
         m_pvData = pvData_;
00048
         if (!bRepeat_)
```

```
00049
        {
00050
            m_ucFlags = TIMERLIST_FLAG_ONE_SHOT;
00051
00052
         else
00053
        {
00054
            m ucFlags = 0;
00056
         m_pclOwner = Scheduler::GetCurrentThread();
00057
         TimerScheduler::Add(this);
00058 }
00059
00060 //----
00062 {
00063
         m_ulTimerTolerance = MSECONDS_TO_TICKS(ulToleranceMs_);
00064
        Start(bRepeat_, ulIntervalMs_, pfCallback_, pvData_);
00065 }
00066
00068 void Timer::Stop()
00069 {
00070
         TimerScheduler::Remove(this);
00071 }
00072
00074 void Timer::SetIntervalTicks( K_ULONG ulTicks_ )
00075 {
00076
         m_ulInterval = ulTicks_;
00077 }
00078
00082 void Timer::SetIntervalSeconds( K_ULONG ulSeconds_)
00083 {
00084
         m_ulInterval = SECONDS_TO_TICKS(ulSeconds_);
00085 }
00087 //-
00088 void Timer::SetIntervalMSeconds( K_ULONG ulMSeconds_)
00089 {
00090
         m ulInterval = MSECONDS TO TICKS(ulMSeconds);
00091 }
00092
00094 void Timer::SetIntervalUSeconds( K_ULONG uluSeconds_)
00095 {
00096
         m_ulInterval = USECONDS_TO_TICKS(ulUSeconds_);
00097 }
00098
00099 //--
00100 void Timer::SetTolerance(K_ULONG ulTicks_)
00101 {
00102
         m_ulTimerTolerance = ulTicks_;
00103 }
00104
00105 #endif
00106
```

15.103 /home/moslevin/Project/R1/kernel/timerlist.cpp File Reference

Implements timer list processing algorithms, responsible for all timer tick and expiry logic.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timerlist.h"
#include "kerneltimer.h"
#include "threadport.h"
#include "kerneldebug.h"
#include "quantum.h"
```

Macros

• #define __FILE_ID__ TIMERLIST_CPP

File ID used in kernel trace calls.

15.103.1 Detailed Description

Implements timer list processing algorithms, responsible for all timer tick and expiry logic.

Definition in file timerlist.cpp.

15.104 timerlist.cpp

```
00001
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 #include "timerlist.h"
00027 #include "kerneltimer.h"
00028 #include "threadport.h"
00029 #include "kerneldebug.h"
00030 #include "quantum.h"
00031
00032 //---
00035 #endif
00036 #define __FILE_ID__
                            TIMERLIST_CPP
00037
00038 #if KERNEL USE TIMERS
00039
00040 //----
00041 TimerList TimerScheduler::m_clTimerList;
00042
00043 //----
00044 void TimerList::Init(void)
00045 {
00046
         m_bTimerActive = 0;
00047
         m_ulNextWakeup = 0;
00048 }
00049
00050 //---
00051 void TimerList::Add(Timer *pclListNode_)
00052 {
00053 #if KERNEL_TIMERS_TICKLESS
00054
          K_BOOL bStart = 0;
00055
         K_LONG lDelta;
00056 #endif
00057
00058
         CS_ENTER();
00059
00060 #if KERNEL_TIMERS_TICKLESS
00061
          if (GetHead() == NULL)
00062
00063
              bStart = 1:
00064
00065 #endif
00066
00067
          pclListNode_->ClearNode();
00068
          DoubleLinkList::Add(pclListNode_);
00069
00070
         // Set the initial timer value
00071
         pclListNode_->m_ulTimeLeft = pclListNode_->m_ulInterval;
00072
00073 #if KERNEL_TIMERS_TICKLESS
00074
          if (!bStart)
00075
00076
              // If the new interval is less than the amount of time remaining...
00077
              1Delta = KernelTimer::TimeToExpiry() - pclListNode_->
      m_ulInterval;
```

15.104 timerlist.cpp 209

```
00078
00079
              if (lDelta > 0)
00080
00081
                  \ensuremath{//} Set the new expiry time on the timer.
00082
                  m_ulNextWakeup = KernelTimer::SubtractExpiry((
      K_ULONG) lDelta);
00083
           }
00084
00085
          else
00086
          {
00087
              m_ulNextWakeup = pclListNode_->m_ulInterval;
              KernelTimer::SetExpiry(m_ulNextWakeup);
00088
00089
              KernelTimer::Start();
00090
00091 #endif
00092
         // Set the timer as active.
00093
00094
         pclListNode_->m_ucFlags |= TIMERLIST_FLAG_ACTIVE;
00095
         CS_EXIT();
00096 }
00097
00098 //---
00099 void TimerList::Remove(Timer *pclLinkListNode_)
00100 {
00101
          CS_ENTER();
00102
00103
          DoubleLinkList::Remove(pclLinkListNode_);
00104
00105 #if KERNEL_TIMERS_TICKLESS
       if (this->GetHead() == NULL)
00106
00107
00108
              KernelTimer::Stop();
00109
00110 #endif
00111
         CS EXIT();
00112
00113 }
00115 //--
00116 void TimerList::Process(void)
00117 (
00118 #if KERNEL_TIMERS_TICKLESS
         K_ULONG ulNewExpiry;
00119
          K_ULONG ulOvertime;
00120
00121
         K_BOOL bContinue;
00122 #endif
00123
00124
         Timer *pclNode;
00125
        Timer *pclPrev;
00126
00127 #if KERNEL_USE_QUANTUM
00128
         Quantum::SetInTimer();
00129 #endif
00130 #if KERNEL_TIMERS_TICKLESS
          // Clear the timer and its expiry time - keep it running though
00131
          KernelTimer::ClearExpiry();
00132
         do
00134
00135 #endif
00136
              pclNode = static_cast<Timer*>(GetHead());
              pclPrev = NULL;
00137
00138
00139 #if KERNEL_TIMERS_TICKLESS
00140
             bContinue = 0;
00141
              ulNewExpiry = MAX_TIMER_TICKS;
00142 #endif
00143
              // Subtract the elapsed time interval from each active timer.
00144
00145
              while (pclNode)
00146
              {
00147
                  // Active timers only...
00148
                  if (pclNode->m_ucFlags & TIMERLIST_FLAG_ACTIVE)
00149
                  {
00150
                      // Did the timer expire?
00151 #if KERNEL_TIMERS_TICKLESS
00152 if (pclNode->m_ulTimeLeft <= m_ulNextWakeup)
00153 #else
00154
                      pclNode->m_ulTimeLeft--;
00155
                      if (0 == pclNode->m_ulTimeLeft)
00156 #endif
00157
00158
                          // Yes - set the "callback" flag - we'll execute the callbacks later
00159
                          pclNode->m_ucFlags |= TIMERLIST_FLAG_CALLBACK;
00160
00161
                          if (pclNode->m_ucFlags & TIMERLIST_FLAG_ONE_SHOT)
00162
00163
                               // If this was a one-shot timer, deactivate the timer.
```

```
00164
                                pclNode->m_ucFlags |= TIMERLIST_FLAG_EXPIRED;
                                pclNode->m_ucFlags &= ~TIMERLIST_FLAG_ACTIVE;
00165
00166
                           }
00167
                           else
00168
00169
                                // Reset the interval timer.
00171
                                // I think we're good though...
00172
                                pclNode->m_ulTimeLeft = pclNode->
      m ulInterval;
00173
00174 #if KERNEL TIMERS TICKLESS
00175
                                // If the time remaining (plus the length of the tolerance interval)
00176
                                // is less than the next expiry interval, set the next expiry interval.
00177
                                K_ULONG ulTmp = pclNode->m_ulTimeLeft + pclNode->
     m_ulTimerTolerance;
00178
00179
                                if (ulTmp < ulNewExpiry)</pre>
00180
00181
                                    ulNewExpiry = ulTmp;
00182
00183 #endif
00184
                           }
00185
00186 #if KERNEL_TIMERS_TICKLESS
                      else
00188
00189
                            // Not expiring, but determine how K_LONG to run the next timer interval for.
00190
                           pclNode->m_ulTimeLeft -= m_ulNextWakeup;
                           if (pclNode->m_ulTimeLeft < ulNewExpiry)</pre>
00191
00192
00193
                                ulNewExpiry = pclNode->m_ulTimeLeft;
00194
00195
00196 #endif
00197
00198
                  pclNode = static_cast<Timer*>(pclNode->GetNext());
00199
              }
00200
00201
              // Process the expired timers callbacks.
00202
              pclNode = static_cast<Timer*>(GetHead());
               while (pclNode)
00203
00204
              {
00205
                  pclPrev = NULL;
00206
00207
                   // If the timer expired, run the callbacks now.
00208
                   if (pclNode->m_ucFlags & TIMERLIST_FLAG_CALLBACK)
00209
                       // Run the callback. these callbacks must be very fast...
00210
                       pclNode->m_pfCallback( pclNode->m_pclOwner, pclNode->
00211
      m_pvData );
00212
                       pclNode->m_ucFlags &= ~TIMERLIST_FLAG_CALLBACK;
00213
                       // If this was a one-shot timer, let's remove it.
if (pclNode->m_ucFlags & TIMERLIST_FLAG_ONE_SHOT)
00214
00215
00216
00217
                           pclPrev = pclNode;
00218
00219
00220
                   pclNode = static_cast<Timer*>(pclNode->GetNext());
00221
00222
                   // Remove one-shot-timers
00223
                   if (pclPrev)
00224
00225
                       Remove (pclPrev);
00226
                   }
00227
              }
00228
00229 #if KERNEL_TIMERS_TICKLESS
              // Check to see how much time has elapsed since the time we
00231
               // acknowledged the interrupt...
00232
              ulOvertime = KernelTimer::GetOvertime();
00233
00234
              if( ulOvertime >= ulNewExpiry ) {
00235
                  m_ulNextWakeup = ulOvertime;
00236
                   bContinue = 1;
00237
00238
00239
          // If it's taken longer to go through this loop than would take us to
00240
          // the next expiry, re-run the timing loop \,
00241
00242
          } while (bContinue);
00243
00244
          // This timer elapsed, but there's nothing more to do...
          // Turn the timer off.
if (ulNewExpiry >= MAX_TIMER_TICKS)
00245
00246
00247
          {
```

```
00248
              KernelTimer::Stop();
00249
00250
          else
00251
              // Update the timer with the new "Next Wakeup" value, plus whatever
00252
00253
              // overtime has accumulated since the last time we called this handler
00255
              m_ulNextWakeup = KernelTimer::SetExpiry(ulNewExpiry +
      ulOvertime);
00256
00257 #endif
00258 #if KERNEL_USE_QUANTUM
00259
          Quantum::ClearInTimer();
00260 #endif
00261 }
00262
00263
00264 #endif //KERNEL USE TIMERS
```

15.105 /home/moslevin/Project/R1/kernel/tracebuffer.cpp File Reference

Kernel trace buffer class definition.

```
#include "kerneltypes.h"
#include "tracebuffer.h"
#include "mark3cfg.h"
#include "writebuf16.h"
#include "kerneldebug.h"
```

15.105.1 Detailed Description

Kernel trace buffer class definition.

Definition in file tracebuffer.cpp.

15.106 tracebuffer.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
80000
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 #include "writebuf16.h"
00023 #include "kerneldebug.h"
00024
00025 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00026 //
00027 WriteBuffer16 TraceBuffer::m_clBuffer;
00028 volatile K_USHORT TraceBuffer::m_usIndex;
00029 K_USHORT TraceBuffer::m_ausBuffer[ (TRACE_BUFFER_SIZE/sizeof(K_USHORT)) ];
00030
00031 //-
00032 void TraceBuffer::Init()
00033 {
00034
          m_clBuffer.SetBuffers(m_ausBuffer, TRACE_BUFFER_SIZE/sizeof(K_USHORT));
00035
          m_usIndex = 0;
00036 }
00037
00038 //--
00039 K_USHORT TraceBuffer::Increment()
```

15.107 /home/moslevin/Project/R1/kernel/writebuf16.cpp File Reference

16 bit circular buffer implementation with callbacks.

```
#include "kerneltypes.h"
#include "writebuf16.h"
#include "kerneldebug.h"
#include "threadport.h"
```

15.107.1 Detailed Description

16 bit circular buffer implementation with callbacks.

Definition in file writebuf16.cpp.

15.108 writebuf16.cpp

```
00001 /*=========
00002
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00020 #include "kerneltypes.h'
00021 #include "writebuf16.h"
00022 #include "kerneldebug.h"
00023 #include "threadport.h"
00024
00025 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00026
00027 //
00028 void WriteBuffer16::WriteData( K_USHORT *pusBuf_, K_USHORT usLen_ )
00029 {
00030
         K_USHORT *apusBuf[1];
00031
        K_USHORT ausLen[1];
00032
         apusBuf[0] = pusBuf_;
00033
         ausLen[0] = usLen_;
00034
00035
00036
         WriteVector( apusBuf, ausLen, 1 );
00037 }
00038
00039 //--
00040 void WriteBuffer16::WriteVector( K_USHORT **ppusBuf_, K_USHORT *pusLen_,
00041 {
00042
         K_USHORT usTempHead;
00043
         K UCHAR i;
         K_UCHAR j;
K_USHORT usTotalLen = 0;
00044
00045
00046
         K_BOOL bCallback = false;
```

```
00047
           K_BOOL bRollover = false;
           // Update the head pointer synchronously, using a small // critical section in order to provide thread safety without
00048
00049
           // compromising on responsiveness by adding lots of extra
00050
00051
           // interrupt latency.
00052
           CS_ENTER();
00054
00055
           usTempHead = m_usHead;
00056
               for (i = 0; i < ucCount_; i++)</pre>
00057
00058
00059
                   usTotalLen += pusLen_[i];
00060
00061
               m_usHead = (usTempHead + usTotalLen) % m_usSize;
00062
           CS EXIT():
00063
00064
00065
           // Call the callback if we cross the 50% mark or rollover
00066
           if (m_usHead < usTempHead)</pre>
00067
00068
               if (m_pfCallback)
00069
               {
00070
                   bCallback = true:
00071
                   bRollover = true;
00072
00073
00074
           else if ((usTempHead < (m_usSize >> 1)) && (m_usHead >= (m_usSize >> 1)))
00075
00076
               // Only trigger the callback if it's non-null
00077
               if (m_pfCallback)
00078
               {
00079
                   bCallback = true;
08000
               }
00081
           }
00082
           // Are we going to roll-over?
for (j = 0; j < ucCount_; j++)</pre>
00083
00084
00085
00086
               K_USHORT usSegmentLength = pusLen_[j];
00087
               if (usSegmentLength + usTempHead >= m_usSize)
00088
               {
                    // We need to two-part this... First part: before the rollover
00089
00090
                   K_USHORT usTempLen;
                   K_USHORT *pusTmp = &m_pusData[ usTempHead ];
K_USHORT *pusSrc = ppusBuf_[j];
00091
00092
00093
                   usTempLen = m_usSize - usTempHead;
00094
                    for (i = 0; i < usTempLen; i++)
00095
00096
                        *pusTmp++ = *pusSrc++;
00097
                   }
00098
00099
                   // Second part: after the rollover
00100
                   usTempLen = usSegmentLength - usTempLen;
00101
                    pusTmp = m_pusData;
                    for (i = 0; i < usTempLen; i++)</pre>
00102
00103
00104
                        *pusTmp++ = *pusSrc++;
00105
00106
00107
               else
00108
00109
                    // No rollover - do the copy all at once.
                   K_USHORT *pusSrc = ppusBuf_[j];
K_USHORT *pusTmp = &m_pusData[ usTempHead ];
00110
00111
00112
                   for (K_USHORT i = 0; i < usSegmentLength; i++)</pre>
00113
00114
                        *pusTmp++ = *pusSrc++;
00115
                    }
00116
               }
00117
          }
00118
00119
           // Call the callback if necessary
00120
00121
           if (bCallback)
00122
00123
                if (bRollover)
00124
                    // Rollover - process the back-half of the buffer
00125
                    m_pfCallback( &m_pusData[ m_usSize >> 1], m_usSize >> 1 );
00126
00127
               }
00128
               else
00129
               {
00130
                    // 50% point - process the front-half of the buffer
00131
                   m_pfCallback( m_pusData, m_usSize >> 1);
00132
               }
00133
           }
```

00134 } 00135 00136 #endif

Index

/home/moslevin/Project/R1/kernel/atomic.cpp, 107	/home/moslevin/Project/R1/kernel/public/II.h, 165, 166
/home/moslevin/Project/R1/kernel/blocking.cpp, 109	/home/moslevin/Project/R1/kernel/public/manual.h, 167
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	cd/keme/postey.in/Project/R1/kernel/public/mark3.h, 167,
cpp, 110	168
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	
cpp, 111, 112	169, 172
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	
cpp, 113	173, 175
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	cc/paner/mes/evip/Gneject/R1/kernel/public/mutex.h, 176,
h, 115	177
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	cc/pane/nesteviswProject/R1/kernel/public/paniccodes.h,
h, 116	177, 178
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	
h, 117	179
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	cc/panec/naseguip/Project/R1/kernel/public/quantum.h,
h, 118, 119	179, 180
/home/moslevin/Project/R1/kernel/cpu/avr/atmega328p/gd	cc/hhmea/mostevin/Project/R1/kernel/public/scheduler.h,
cpp, 121, 122	180, 181
/home/moslevin/Project/R1/kernel/driver.cpp, 124, 125	/home/moslevin/Project/R1/kernel/public/thread.h, 182,
/home/moslevin/Project/R1/kernel/eventflag.cpp, 126,	183
127	/home/moslevin/Project/R1/kernel/public/threadlist.h,
/home/moslevin/Project/R1/kernel/kernel.cpp, 130, 131	186
/home/moslevin/Project/R1/kernel/kernelaware.cpp,	/home/moslevin/Project/R1/kernel/public/timer.h, 187,
132, 133	188
/home/moslevin/Project/R1/kernel/ksemaphore.cpp,	/home/moslevin/Project/R1/kernel/public/timerlist.h, 190
135, 136	/home/moslevin/Project/R1/kernel/public/timerscheduler.
/home/moslevin/Project/R1/kernel/II.cpp, 138, 139	h, 191
/home/moslevin/Project/R1/kernel/message.cpp, 141	/home/moslevin/Project/R1/kernel/public/tracebuffer.h,
/home/moslevin/Project/R1/kernel/mutex.cpp, 143, 144	/home/moslevin/Project/R1/kernel/public/writebuf16.h,
/home/moslevin/Project/R1/kernel/profile.cpp, 147, 148	192, 193
/home/moslevin/Project/R1/kernel/public/atomic.h, 149	/home/moslevin/Project/R1/kernel/quantum.cpp, 193,
/home/moslevin/Project/R1/kernel/public/blocking.h,	194
150, 151	/home/moslevin/Project/R1/kernel/scheduler.cpp, 196,
/home/moslevin/Project/R1/kernel/public/debugtokens	197
h, 151, 152	/home/moslevin/Project/R1/kernel/thread.cpp, 198, 199
/home/moslevin/Project/R1/kernel/public/driver.h, 153,	/home/moslevin/Project/R1/kernel/threadlist.cpp, 204
154	/home/moslevin/Project/R1/kernel/timer.cpp, 206
/home/moslevin/Project/R1/kernel/public/eventflag.h,	/home/moslevin/Project/R1/kernel/timerlist.cpp, 200,
155, 156	208
/home/moslevin/Project/R1/kernel/public/kernel.h, 156,	/home/moslevin/Project/R1/kernel/tracebuffer.cpp, 211
157	/home/moslevin/Project/R1/kernel/writebuf16.cpp, 212
/home/moslevin/Project/R1/kernel/public/kernelaware.h,	moneymosicviii/i Tojoce/TT/Norme/wittebul To.opp, 212
158, 159	Add
/home/moslevin/Project/R1/kernel/public/kerneldebug.h,	CircularLinkList, 45
160	DoubleLinkList, 48
/home/moslevin/Project/R1/kernel/public/kerneltypes.h,	DriverList, 52
162, 163	LinkList, 68
/home/moslevin/Project/R1/kernel/public/ksemaphore.h,	Scheduler, 82
164	ThreadList, 97

TimerList, 104	Add, 52
TimerScheduler, 105	FindByPath, 53
AddThread	Init, 53
Quantum, 80	Remove, 53
aucCLZ	DrvCmp
scheduler.cpp, 196	driver.cpp, 124
Block	EVENT_FLAG_ALL
BlockingObject, 43	kerneltypes.h, 163
BlockingObject, 43	EVENT_FLAG_ALL_CLEAR
Block, 43	kerneltypes.h, 163
UnBlock, 44	EVENT_FLAG_ANY
OO ENTER	kerneltypes.h, 163
CS_ENTER	EVENT_FLAG_ANY_CLEAR
threadport.h, 119	kerneltypes.h, 163
CircularLinkList, 44	EVENT_FLAG_MODES
Add, 45	kerneltypes.h, 163
Remove, 45	EVENT_FLAG_PENDING_UNBLOCK
Claim	kerneltypes.h, 163
Mutex, 75	EventFlag, 53
Claim_i	Clear, 54
Mutex, 75	GetMask, 54
Clear	Set, 55
EventFlag, 54	Wait, <u>55</u>
ClearInTimer	Wait_i, 55
Quantum, 80	WakeMe, 56
Close	EventFlagOperation_t
DevNull, 46	kerneltypes.h, 163
Driver, 50	eventflag.cpp
ComputeCurrentTicks	TimedEventFlag_Callback, 126
ProfileTimer, 78	Exit
ContextSwitchSWI	Thread, 91
Thread, 91	ExitSimulator
Control	KernelAware, 61
DevNull, 46	
Driver, 50	FakeThread_t, 56
	FindByPath
DI	DriverList, 53
KernelSWI, 65	
DevNull, 45	
	g_blsKernelAware
Close, 46	kernelaware.cpp, 133
Close, 46 Control, 46	kernelaware.cpp, 133 g_stKAData
Close, 46 Control, 46 Open, 47	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133
Close, 46 Control, 46 Open, 47 Read, 47	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51 SetName, 51	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent ProfileTimer, 78 Thread, 91
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent ProfileTimer, 78
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51 SetName, 51 Write, 51 driver.cpp	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent ProfileTimer, 78 Thread, 91
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51 SetName, 51 Write, 51	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent ProfileTimer, 78 Thread, 91 GetCurrentThread
Close, 46 Control, 46 Open, 47 Read, 47 Write, 47 DoubleLinkList, 48 Add, 48 Remove, 49 Driver, 49 Close, 50 Control, 50 GetPath, 50 Open, 51 Read, 51 SetName, 51 Write, 51 driver.cpp	kernelaware.cpp, 133 g_stKAData kernelaware.cpp, 133 GetAverage ProfileTimer, 78 GetCode Message, 71 GetCount MessageQueue, 73 Semaphore, 85 GetCurPriority Thread, 91 GetCurrent ProfileTimer, 78 Thread, 91 GetCurrentThread Scheduler, 83

GetEventFlagMask	Thread, 93
Thread, 91	InitStack
GetEventFlagMode	ThreadPort, 99
Thread, 91	IsEnabled
GetExpired	Scheduler, 83
Thread, 92	IsPanic
GetHead	Kernel, 59
LinkList, 68	IsSimulatorAware
GetID	KernelAware, 61
Thread, 92	IsStarted
GetIdleThread	Kernel, 59
Kernel, 59	
GetMask	KA_COMMAND_EXIT_SIMULATOR
EventFlag, 54	kernelaware.h, 158
GetNext	KA_COMMAND_IDLE
LinkListNode, 70	kernelaware.h, 158
GetNextThread	KA_COMMAND_PRINT
Scheduler, 83	kernelaware.h, 159
GetOvertime	KA_COMMAND_PROFILE_INIT
KernelTimer, 66	kernelaware.h, 158
GetOwner	KA COMMAND PROFILE REPORT
Thread, 92	kernelaware.h, 158
GetPath	KA COMMAND PROFILE START
	kernelaware.h, 158
Driver, 50	KA COMMAND PROFILE STOP
GetPrev	kernelaware.h, 158
LinkListNode, 70	KA_COMMAND_TRACE_0
GetPriority	kernelaware.h, 158
Thread, 92	KA_COMMAND_TRACE_1
GetQuantum	kernelaware.h, 158
Thread, 92	KA_COMMAND_TRACE_2
GetStackSlack	kernelaware.h, 159
Thread, 92	KERNEL USE ATOMIC
GetState	
Thread, 93	mark3cfg.h, 170
GetStopList	KERNEL_USE_MESSAGE
Scheduler, 83	mark3cfg.h, 171
GetTail	KERNEL_USE_PROFILER
LinkList, 68	mark3cfg.h, 171
GetThreadList	KERNEL_USE_QUANTUM
Scheduler, 83	mark3cfg.h, 171
GlobalMessagePool, 57	KERNEL_USE_TIMEOUTS
Pop, 57	mark3cfg.h, 172
Push, 57	KERNEL_USE_TIMERS
	mark3cfg.h, 172
HighestWaiter	Kernel, 58
ThreadList, 97	GetIdleThread, 59
	Init, 59
InheritPriority	IsPanic, 59
Thread, 93	IsStarted, 59
Init	Panic, 59
DriverList, 53	SetIdleFunc, 60
Kernel, 59	SetPanic, 60
Profiler, 77	Start, 60
ProfileTimer, 78	KernelAware, 60
Semaphore, 85	ExitSimulator, 61
Thread, 93	IsSimulatorAware, 61
TimerList, 104	Print, 62
TimerScheduler, 105	ProfileInit, 62
InitIdle	ProfileReport, 62
	•

ProfileStart, 62	KERNEL_USE_TIMEOUTS, 172
ProfileStop, 62	KERNEL_USE_TIMERS, 172
Trace, 62, 63	SAFE UNLINK, 172
Trace_i, 63	Message, 70
KernelAwareCommand_t	GetCode, 71
kernelaware.h, 158	GetData, 71
KernelAwareData t, 64	SetCode, 71
KernelSWI, 64	•
	SetData, 72
DI, 65	MessageQueue, 72
RI, 65	GetCount, 73
KernelTimer, 65	Receive, 73
GetOvertime, 66	Receive_i, 73
RI, 66	Send, 74
Read, 66	Mutex, 74
SetExpiry, 67	Claim, 75
SubtractExpiry, 67	Claim_i, 75
TimeToExpiry, 67	Release, 76
kernelaware.h	WakeMe, 76
KA_COMMAND_EXIT_SIMULATOR, 158	mutex.cpp
KA_COMMAND_IDLE, 158	TimedMutex_Calback, 144
KA_COMMAND_PRINT, 159	_ ,
KA_COMMAND_PROFILE_INIT, 158	Open
KA_COMMAND_PROFILE_REPORT, 158	DevNull, 47
KA COMMAND PROFILE START, 158	Driver, 51
KA COMMAND PROFILE STOP, 158	Driver, 31
KA COMMAND TRACE 0, 158	Panic
KA_COMMAND_TRACE_1, 158	
KA_COMMAND_TRACE_2, 159	Kernel, 59
kernelaware.cpp	Pend
g_blsKernelAware, 133	Semaphore, 87
g_stKAData, 133	Pend_i
kernelaware.h	Semaphore, 87
KernelAwareCommand_t, 158	Pop
kerneltypes.h	GlobalMessagePool, 57
EVENT_FLAG_ALL, 163	Post
EVENT_FLAG_ALL_CLEAR, 163	Semaphore, 87
EVENT_FLAG_ALL_CLEAN, 163 EVENT_FLAG_ANY, 163	Print
	KernelAware, 62
EVENT_FLAG_ANY_CLEAR, 163	Process
EVENT_FLAG_MODES, 163	TimerList, 104
EVENT_FLAG_PENDING_UNBLOCK, 163	TimerScheduler, 106
kerneltypes.h	ProfileInit
EventFlagOperation_t, 163	KernelAware, 62
ksemaphore.cpp	ProfileReport
TimedSemaphore_Callback, 135	KernelAware, 62
1:11:10	ProfileStart
LinkList, 67	KernelAware, 62
Add, 68	ProfileStop
GetHead, 68	KernelAware, 62
GetTail, 68	ProfileTimer, 77
Remove, 69	
LinkListNode, 69	ComputeCurrentTicks, 78
GetNext, 70	GetAverage, 78
GetPrev, 70	GetCurrent, 78
	Init, 78
mark3cfg.h	Start, 79
KERNEL_USE_ATOMIC, 170	Profiler, 76
KERNEL_USE_MESSAGE, 171	Init, 77
KERNEL_USE_PROFILER, 171	Push
KERNEL_USE_QUANTUM, 171	GlobalMessagePool, 57

Quantum, 79	Pend, 87
AddThread, 80	Pend i, 87
ClearInTimer, 80	Post, 87
RemoveThread, 80	· ·
	WakeMe, 87
SetTimer, 80	Send
SetTimer, 80	MessageQueue, 74
UpdateTimer, 80	Set
quantum.cpp	EventFlag, 55
QuantumCallback, 194	SetCallback
QuantumCallback	Timer, 101
quantum.cpp, 194	SetCode
QueueScheduler	Message, 71
Scheduler, 83	SetCurrent
	Thread, 94
RI	SetData
KernelSWI, 65	Message, 72
KernelTimer, 66	Timer, 101
Read	SetEventFlagMask
DevNull, 47	Thread, 94
Driver, 51	SetEventFlagMode
KernelTimer, 66	Thread, 94
Receive	
MessageQueue, 73	SetExpired
Receive i	Thread, 94
MessageQueue, 73	SetExpiry
Release	KernelTimer, 67
Mutex, 76	SetFlagPointer
Remove	ThreadList, 98
	SetFlags
CircularLinkList, 45	Timer, 101
DoubleLinkList, 49	SetID
DriverList, 53	Thread, 94
LinkList, 69	SetIdleFunc
Scheduler, 84	Kernel, 60
ThreadList, 98	SetInTimer
TimerList, 104	Quantum, 80
TimerScheduler, 106	SetIntervalMSeconds
RemoveThread	
Quantum, 80	Timer, 101
	SetIntervalSeconds
SAFE_UNLINK	Timer, 101
mark3cfg.h, 172	SetIntervalTicks
Schedule	Timer, 102
Scheduler, 84	SetIntervalUSeconds
Scheduler, 81	Timer, 102
Add, 82	SetName
GetCurrentThread, 83	Driver, 51
GetNextThread, 83	SetOwner
GetStopList, 83	Thread, 94
GetThreadList, 83	Timer, 102
IsEnabled, 83	SetPanic
QueueScheduler, 83	Kernel, 60
Remove, 84	SetPriority
Schedule, 84	Thread, 95
SetScheduler, 84	ThreadList, 98
scheduler.cpp	SetPriorityBase
aucCLZ, 196	Thread, 95
Semaphore, 84	SetQuantum
GetCount, 85	Thread, 95
Init, 85	SetScheduler

Scheduler, 84	CS_ENTER, 119
SetState	TimeToExpiry
Thread, 95	KernelTimer, 67
SetTimer	TimedEventFlag_Callback
Quantum, 80	eventflag.cpp, 126
SetTolerance	TimedMutex_Calback
Timer, 102	mutex.cpp, 144
Sleep	TimedSemaphore_Callback
Thread, 95	ksemaphore.cpp, 135
Start	Timer, 99
Kernel, 60	SetCallback, 101
ProfileTimer, 79	SetData, 101
Timer, 102, 103	SetFlags, 101
Stop	SetIntervalMSeconds, 101
Thread, 96	SetIntervalSeconds, 101
Timer, 103	
SubtractExpiry	SetIntervalTicks, 102
KernelTimer, 67	SetIntervalUSeconds, 102
Remember, 67	SetOwner, 102
Thread, 88	SetTolerance, 102
ContextSwitchSWI, 91	Start, 102, 103
Exit, 91	Stop, 103
GetCurPriority, 91	timer.h
GetCurrent, 91	TimerCallback_t, 188
GetEventFlagMask, 91	TimerCallback_t
	timer.h, 188
GetEventFlagMode, 91	TimerList, 103
GetExpired, 92	Add, 104
GetID, 92	Init, 104
GetOwner, 92	Process, 104
GetPriority, 92	Remove, 104
GetQuantum, 92	TimerScheduler, 105
GetStackSlack, 92	Add, 105
GetState, 93	•
InheritPriority, 93	Init, 105
Init, 93	Process, 106
InitIdle, 93	Remove, 106
SetCurrent, 94	Trace
SetEventFlagMask, 94	KernelAware, 62, 63
SetEventFlagMode, 94	Trace_i
SetExpired, 94	KernelAware, 63
SetID, 94	
SetOwner, 94	USleep
SetPriority, 95	Thread, 96
SetPriorityBase, 95	UnBlock
SetQuantum, 95	BlockingObject, 44
SetState, 95	UpdateTimer
Sleep, 95	Quantum, 80
Stop, 96	
USleep, 96	Wait
Yield, 96	EventFlag, 55
ThreadList, 96	Wait_i
Add, 97	EventFlag, 55
HighestWaiter, 97	WakeMe
	EventFlag, 56
Remove, 98	<u> </u>
SetFlagPointer, 98	Mutex, 76
SetPriority, 98	Semaphore, 87
ThreadPort, 98	Write
InitStack, 99	DevNull, 47
threadport.h	Driver, 51

Yield

Thread, 96