Mark3 Realtime Kernel

Generated by Doxygen 1.8.13

## **Contents**

1	The	Mark3 Realtime Kernel	1
2	Lice	ense	3
	2.1	License	3
3	Con	figuring The Kernel	5
	3.1	Overview	5
		3.1.1 Kernel Configuration Options	5
		3.1.2 Port Configuration Options	6
4	Buile	ding The Kernel	9
	4.1	Prerequisites	9
	4.2	Building	9
	4.3	Supported Targets	10
	4.4	More Info	10
5	The	Mark3 API	11
	5.1	Kernel Setup	11
	5.2	Threads	12
		5.2.1 Thread Setup	12
		5.2.2 Entry Functions	13
	5.3	Timers	14
	5.4	Semaphores	14
	5.5	Mutexes	15
	5.6	Event Flags	16

ii CONTENTS

	5.7	Messages	16
		5.7.1 Message Objects	17
		5.7.2 Message Queues	17
		5.7.3 Messaging Example	18
	5.8	Mailboxes	18
		5.8.1 Mailbox Example	18
	5.9	Notification Objects	19
		5.9.1 Notification Example	19
	5.10	Condition Variables	20
		5.10.1 Condition Variable Example	20
	5.11	Reader-Write Locks	21
		5.11.1 Reader-Write Lock Example	21
	5.12	Sleep	21
	5.13	Round-Robin Quantum	22
	5.14	Coroutines	22
	5.15	Critical Guards	23
	5.16	Lock Guards	23
	5.17	Scheduler Guards	24
6			
6		Scheduler Guards	24 <b>25</b>
6	Why		
	Why	Mark3?	25 27
	Why	Mark3? en to use an RTOS?	25 27
	Why Whe	en to use an RTOS?  The reality of system code	<b>25 27</b> 27
	Why Whe	Mark3?  In to use an RTOS?  The reality of system code	<b>25 27</b> 28
	Why Whe	Mark3?  In to use an RTOS?  The reality of system code  Superloops, and their limitations  7.2.1 Intro to Superloops	25 27 27 28 28
	Why Whe	m to use an RTOS? The reality of system code Superloops, and their limitations 7.2.1 Intro to Superloops 7.2.2 The simplest loop	25 27 27 28 28 29
	Why Whe	m to use an RTOS? The reality of system code Superloops, and their limitations 7.2.1 Intro to Superloops 7.2.2 The simplest loop 7.2.3 Interrupt-Driven Super-loop	25 27 27 28 28 29 29
	Why Whe	mark3?  In to use an RTOS?  The reality of system code  Superloops, and their limitations  7.2.1 Intro to Superloops  7.2.2 The simplest loop  7.2.3 Interrupt-Driven Super-loop  7.2.4 Cooperative multi-tasking	25 27 27 28 28 29 29
	Why Whe 7.1 7.2	Mark3?  In to use an RTOS?  The reality of system code  Superloops, and their limitations  7.2.1 Intro to Superloops  7.2.2 The simplest loop  7.2.3 Interrupt-Driven Super-loop  7.2.4 Cooperative multi-tasking  7.2.5 Hybrid cooperative/preemptive multi-tasking	25 27 27 28 28 29 29 31 32
	Why Whe 7.1 7.2	The reality of system code Superloops, and their limitations 7.2.1 Intro to Superloops 7.2.2 The simplest loop 7.2.3 Interrupt-Driven Super-loop 7.2.4 Cooperative multi-tasking 7.2.5 Hybrid cooperative/preemptive multi-tasking Problems with superloops	25 27 28 28 29 29 31 32 33
	Why Whe 7.1 7.2	m to use an RTOS?  The reality of system code  Superloops, and their limitations 7.2.1 Intro to Superloops 7.2.2 The simplest loop 7.2.3 Interrupt-Driven Super-loop 7.2.4 Cooperative multi-tasking 7.2.5 Hybrid cooperative/preemptive multi-tasking Problems with superloops 7.3.1 Hidden Costs	25 27 28 28 29 29 31 32 33 33
	Why Whe 7.1 7.2	Mark3?  In to use an RTOS?  The reality of system code  Superloops, and their limitations  7.2.1 Intro to Superloops  7.2.2 The simplest loop  7.2.3 Interrupt-Driven Super-loop  7.2.4 Cooperative multi-tasking  7.2.5 Hybrid cooperative/preemptive multi-tasking  Problems with superloops  7.3.1 Hidden Costs  7.3.2 Tightly-coupled code	25 27 28 28 29 29 31 32 33 33 33

8	Can	you affo	ord an RTOS?	35
	8.1	Intro .		35
	8.2	Applica	ation description	36
	8.3	Runtim	e Overhead	37
	8.4	Analys	is	38
9	Mark	3 Desig	gn Goals	39
	9.1	Overvie	ew	39
		9.1.1	Services Provided by an RTOS Kernel	39
		9.1.2	Guiding Principles of Mark3	39
		9.1.3	Be feature competitive	39
		9.1.4	No external dependencies, no non-language features	39
		9.1.5	Target the most popular hobbyist platforms available	40
		9.1.6	Maximize determinism – but be pragmatic	40
		9.1.7	Apply engineering principles – and that means discipline, measurement and verification	40
		9.1.8	Use Virtualization For Verification	41
10	Mark	3 Kerne	el Architecture	43
	10.1	Overvie	ew	43
	10.2	Thread	ls and Scheduling	46
		10.2.1	A Bit About Threads	46
		10.2.2	Thread States and ThreadLists	47
		10.2.3	Blocking and Unblocking	47
		10.2.4	Blocking Objects	47
	10.3	Inside t	the Mark3 Scheduler	48
		10.3.1	Considerations for Round-Robin Scheduling	50
		10.3.2	Context Switching	51
		10.3.3	Putting It All Together	51
	10.4	Timers		51
		10.4.1	Tick-based Timers	53
		10.4.2	Tickless Timers	53

iv CONTENTS

		10.4.3 Timer Processing Algorithm	53
	10.5	Synchronization and IPC	54
	10.6	Blocking Objects	54
		10.6.1 Semaphores	55
		10.6.2 Mutex	55
		10.6.3 Event Flags	56
		10.6.4 Notification Objects	56
	10.7	Messages and Message Queues	56
		10.7.1 Messages	56
		10.7.2 Message Objects	57
		10.7.3 Message Queues	57
		10.7.4 Mailboxes	57
		10.7.5 Atomic Operations	58
		10.7.6 Drivers	59
	10.8	Kernel Proper and Porting	61
11	C-lai	nguage bindings	69
11		nguage bindings  API Conventions	<b>69</b>
11	11.1		
11	11.1	API Conventions	69 70
	11.1 11.2 Rele	API Conventions	69
	11.1 11.2 Rele	API Conventions	69 70
	11.1 11.2 Rele 12.1	API Conventions	69 70 <b>71</b>
	11.1 11.2 Rele 12.1 12.2	API Conventions	69 70 <b>71</b> 71
	11.1 11.2 Rele 12.1 12.2 12.3	API Conventions	69 70 <b>71</b> 71
	11.1 11.2 Rele 12.1 12.2 12.3 12.4	API Conventions  Allocating Objects  Pase Notes  R10 Release  R9 Release  R8 Release	69 70 <b>71</b> 71 71
	11.1 11.2 Rele 12.1 12.2 12.3 12.4 12.5	API Conventions  Allocating Objects  Pase Notes  R10 Release  R9 Release  R8 Release  R8 (Full Throttle) Release	<ul><li>69</li><li>70</li><li>71</li><li>71</li><li>71</li><li>72</li></ul>
	11.1 11.2 Rele 12.1 12.2 12.3 12.4 12.5 12.6	API Conventions	69 70 <b>71</b> 71 71 71 72 72
	11.1 11.2 Rele 12.1 12.2 12.3 12.4 12.5 12.6 12.7	API Conventions Allocating Objects  Rase Notes R10 Release R9 Release R8 Release R7 (Full Throttle) Release R6 Release R5 Release	69 70 <b>71</b> 71 71 72 72
	11.1 11.2 Rele 12.1 12.2 12.3 12.4 12.5 12.6 12.7 12.8	API Conventions Allocating Objects  Rase Notes R10 Release R9 Release R8 Release R8 Release R7 (Full Throttle) Release R6 Release R5 Release R4 Release	69 70 <b>71</b> 71 71 72 72 73 74
	11.1 11.2 Rele 12.1 12.2 12.3 12.4 12.5 12.6 12.7 12.8 12.9	API Conventions Allocating Objects  Rase Notes R10 Release R9 Release R8 Release R8 Release R7 (Full Throttle) Release R5 Release R4 Release R3 Release	69 70 71 71 71 72 72 73 74 74

13	Code	e Size P	rofiling										77
	13.1	Informa	ation					 	 	 	 	 	 . 77
	13.2	Compil	er Version					 	 	 	 	 	 . 77
	13.3	Profilin	g Results					 	 	 	 	 	 . 77
14	Nam	espace	Index										79
			pace List					 	 	 		 	 . 79
15		archical											81
	15.1	Class H	Hierarchy					 	 	 	 	 	 . 81
16	Clas	s Index											83
	16.1	Class L	ist					 	 	 	 	 	 . 83
													07
17		ndex											87
	17.1	FIIE LIS	t					 	 	 	 	 	 . 87
18	Nam	espace	Documen	tation									91
	18.1	Mark3	Namespac	e Refere	ence			 	 	 	 	 	 . 91
		18.1.1	Typedef D	)ocumer	ntation .			 	 	 	 	 	 . 95
			18.1.1.1	AutoAll	ocAlloc	ator_t		 	 	 	 	 	 . 95
			18.1.1.2	AutoAll	ocFree_	_t		 	 	 	 	 	 . 95
			18.1.1.3	CoPriol	Мар			 	 	 	 	 	 . 96
			18.1.1.4	Corouti	neHano	dler .		 	 	 	 	 	 . 96
			18.1.1.5	IdleFun	IC			 	 	 	 	 	 . 96
			18.1.1.6	PanicF	unc			 	 	 	 	 	 . 96
			18.1.1.7	Priority	Мар			 	 	 	 	 	 . 96
			18.1.1.8	Thread	Context	tCallou	ıt	 	 	 	 	 	 . 96
			18.1.1.9	Thread	Create(	Callout		 	 	 	 	 	 . 97
			18.1.1.10	Thread	EntryFι	ınc .		 	 	 	 	 	 . 97
			18.1.1.11	Thread	ExitCall	lout .		 	 	 	 	 	 . 97
			18.1.1.12	TimerC	allback			 	 	 	 	 	 . 97
		18.1.2	Enumerat	tion Type	Docur	nentati	ion .	 	 	 	 	 	 . 97

vi

		18.1.2.1	AutoAllocType	. 97
		18.1.2.2	EventFlagOperation	. 98
		18.1.2.3	ThreadState	. 98
	18.1.3	Function	Documentation	. 99
		18.1.3.1	ISR()	. 99
		18.1.3.2	KernelTimer_Task()	. 99
		18.1.3.3	PORT_CLZ()	. 99
		18.1.3.4	PORT_CS_ENTER()	. 99
		18.1.3.5	PORT_CS_EXIT()	. 99
		18.1.3.6	PORT_CS_NESTING()	. 100
		18.1.3.7	PORT_IRQ_DISABLE()	. 100
		18.1.3.8	PORT_IRQ_ENABLE()	. 100
		18.1.3.9	Thread_Switch()	. 100
	18.1.4	Variable I	Documentation	. 100
		18.1.4.1	g_kwCriticalCount	. 100
		18.1.4.2	g_kwSFR	. 100
		18.1.4.3	naked	. 101
		18.1.4.4	SR	. 101
		18.1.4.5	uMaxTimerTicks	. 101
		18.1.4.6	uTimerFlagActive	. 101
		18.1.4.7	uTimerFlagCallback	. 101
		18.1.4.8	uTimerFlagExpired	. 102
		18.1.4.9	uTimerFlagOneShot	. 102
		18.1.4.10	uTimerTicksInvalid	. 102
18.2	Mark3::	:Atomic Na	amespace Reference	. 102
	18.2.1	Detailed	Description	. 103
	18.2.2	Function	Documentation	. 103
		18.2.2.1	Add()	. 103
		18.2.2.2	Set()	. 103
		18.2.2.3	Sub()	. 104
		18.2.2.4	TestAndSet()	. 104

CONTENTS vii

19	Clas	s Docur	mentation	105
	19.1	Mark3:	:AutoAlloc Class Reference	105
		19.1.1	Detailed Description	106
		19.1.2	Member Function Documentation	106
			19.1.2.1 Allocate()	106
			19.1.2.2 DestroyObject()	106
			19.1.2.3 DestroyRawData()	106
			19.1.2.4 DestroyUserTypeAllocation()	107
			19.1.2.5 Free()	107
			19.1.2.6 Init()	107
			19.1.2.7 NewObject()	108
			19.1.2.8 NewRawData()	108
			19.1.2.9 NewUserTypeAllocation()	108
			19.1.2.10 SetAllocatorFunctions()	109
		19.1.3	Member Data Documentation	109
			19.1.3.1 m_pfAllocator	109
			19.1.3.2 m_pfFree	109
	19.2	Mark3:	:BlockingObject Class Reference	110
		19.2.1	Detailed Description	111
		19.2.2	Constructor & Destructor Documentation	111
			19.2.2.1 BlockingObject()	111
			19.2.2.2 ~BlockingObject()	111
		19.2.3	Member Function Documentation	111
			19.2.3.1 Block()	111
			19.2.3.2 BlockPriority()	112
			19.2.3.3   IsInitialized()	112
			19.2.3.4 SetInitialized()	112
			19.2.3.5 UnBlock()	113
		19.2.4	Member Data Documentation	114
			19.2.4.1 m_clBlockList	114

viii CONTENTS

		19.2.4.2	m_u8	Initialize	ed				 	 	 	 	 	 	114
		19.2.4.3	m_uE	locking	InitCo	okie .			 	 	 	 	 	 	114
		19.2.4.4	m_uE	locking	Invalid	JCook	ie .		 	 	 	 	 	 	114
19.3	Mark3:	:CircularLii	nkList	Class R	leferer	nce .			 	 	 	 	 	 	115
	19.3.1	Detailed I	Descrip	otion .					 	 	 	 	 	 	115
	19.3.2	Construct	tor & D	estructo	or Doc	umer	ntatio	n	 	 	 	 	 	 	115
		19.3.2.1	Circu	arLinkL	.ist() .				 	 	 	 	 	 	115
	19.3.3	Member F	Functio	n Docu	ımenta	ation			 	 	 	 	 	 	116
		19.3.3.1	Add()						 	 	 	 	 	 	116
		19.3.3.2	Insert	NodeBe	efore()	)			 	 	 	 	 	 	116
		19.3.3.3	opera	tor new	·()				 	 	 	 	 	 	116
		19.3.3.4	Pivot	3ackwa	rd() .				 	 	 	 	 	 	117
		19.3.3.5	Pivot	- orward	I()				 	 	 	 	 	 	117
		19.3.3.6	Remo	ove() .					 	 	 	 	 	 	117
19.4	Mark3:	:CoList Cla	ass Re	ference					 	 	 	 	 	 	117
	19.4.1	Detailed I	Descrip	otion .					 	 	 	 	 	 	118
	19.4.2	Member F	Functio	n Docu	ımenta	ation			 	 	 	 	 	 	118
		19.4.2.1	Add()						 	 	 	 	 	 	118
		19.4.2.2	Remo	ove() .					 	 	 	 	 	 	119
		19.4.2.3	SetPr	ioMap()	)				 	 	 	 	 	 	119
		19.4.2.4	SetPr	iority()					 	 	 	 	 	 	119
	19.4.3	Member I	Data D	ocumer	ntation	1			 	 	 	 	 	 	120
		19.4.3.1	m_pc	IPrioMa	ър				 	 	 	 	 	 	120
		19.4.3.2	m_uP	riority .					 	 	 	 	 	 	120
19.5	Mark3:	:Condition	Variabl	e Class	Refer	rence			 	 	 	 	 	 	120
	19.5.1	Detailed I	Descrip	otion .					 	 	 	 	 	 	121
	19.5.2	Member F	Functio	n Docu	ımenta	ation			 	 	 	 	 	 	121
		19.5.2.1	Broad	lcast().					 	 	 	 	 	 	121
		19.5.2.2	Init()						 	 	 	 	 	 	122
		19.5.2.3	opera	tor new	ı()				 	 	 	 	 	 	122

		19.5.2.4 Signal()	22
		19.5.2.5 Wait() [1/2]	22
		19.5.2.6 Wait() [2/2]	23
	19.5.3	Member Data Documentation	23
		19.5.3.1 m_clMutex	23
		19.5.3.2 m_clSemaphore	23
		19.5.3.3 m_u8Waiters	23
19.6	Mark3:	:Coroutine Class Reference	24
	19.6.1	Detailed Description	24
	19.6.2	Constructor & Destructor Documentation	25
		19.6.2.1 ~Coroutine()	25
	19.6.3	Member Function Documentation	25
		19.6.3.1 Activate()	25
		19.6.3.2 GetPriority()	25
		19.6.3.3 Init()	25
		19.6.3.4 Run()	26
		19.6.3.5 SetPriority()	26
	19.6.4	Member Data Documentation	26
		19.6.4.1 m_bQueued	26
		19.6.4.2 m_pclOwner	27
		19.6.4.3 m_pfHandler	27
		19.6.4.4 m_pvContext	27
		19.6.4.5 m_uPriority	27
19.7	Mark3:	:CoScheduler Class Reference	27
	19.7.1	Detailed Description	28
	19.7.2	Member Function Documentation	28
		19.7.2.1 GetCoList()	28
		19.7.2.2 GetPrioMap()	29
		19.7.2.3 GetStopList()	29
		19.7.2.4 Init()	29

19.7.2.5 Schedule()	130
19.7.3 Member Data Documentation	130
19.7.3.1 m_aclPriorities	130
19.7.3.2 m_clPrioMap	130
19.7.3.3 m_clStopList	130
19.8 Mark3::CriticalGuard Class Reference	131
19.8.1 Detailed Description	131
19.8.2 Constructor & Destructor Documentation	131
19.8.2.1 CriticalGuard()	131
19.8.2.2 ~CriticalGuard()	131
19.9 Mark3::CriticalSection Class Reference	132
19.9.1 Detailed Description	132
19.9.2 Member Function Documentation	132
19.9.2.1 Enter()	132
19.9.2.2 Exit()	133
19.9.2.3 NestingCount()	133
19.10Mark3::DoubleLinkList Class Reference	133
19.10.1 Detailed Description	134
19.10.2 Constructor & Destructor Documentation	134
19.10.2.1 DoubleLinkList()	134
19.10.3 Member Function Documentation	134
19.10.3.1 Add()	134
19.10.3.2 operator new()	134
19.10.3.3 Remove()	135
19.11Mark3::EventFlag Class Reference	135
19.11.1 Detailed Description	136
19.11.2 Constructor & Destructor Documentation	137
19.11.2.1 ~EventFlag()	137
19.11.3 Member Function Documentation	137
19.11.3.1 Clear()	137

CONTENTS xi

19.11.3.2 GetMask()	. 137
19.11.3.3 Init()	. 137
19.11.3.4 operator new()	. 138
19.11.3.5 Set()	. 138
19.11.3.6 Wait() [1/2]	. 138
19.11.3.7 Wait() [2/2]	. 139
19.11.3.8 Wait_i()	. 139
19.11.3.9 WakeMe()	. 140
19.11.4 Member Data Documentation	. 140
19.11.4.1 m_u16SetMask	. 140
19.12Mark3::Kernel Class Reference	. 140
19.12.1 Detailed Description	. 142
19.12.2 Member Function Documentation	. 142
19.12.2.1 CompleteStart()	. 142
19.12.2.2 DebugPrint()	. 142
19.12.2.3 GetStackGuardThreshold()	. 143
19.12.2.4 GetThreadContextSwitchCallout()	. 143
19.12.2.5 GetThreadCreateCallout()	. 143
19.12.2.6 GetThreadExitCallout()	. 143
19.12.2.7 GetTicks()	. 144
19.12.2.8 Init()	. 144
19.12.2.9 IsPanic()	. 144
19.12.2.10sStarted()	. 144
19.12.2.11Panic()	. 144
19.12.2.12SetDebugPrintFunction()	. 145
19.12.2.13SetPanic()	. 145
19.12.2.14SetStackGuardThreshold()	. 146
19.12.2.15SetThreadContextSwitchCallout()	. 146
19.12.2.16SetThreadCreateCallout()	. 146
19.12.2.17SetThreadExitCallout()	. 147

xii CONTENTS

19.12.2.18Start()
19.12.2.19Tick()
19.12.3 Member Data Documentation
19.12.3.1 m_blsPanic
19.12.3.2 m_blsStarted
19.12.3.3 m_pfDebugPrintFunction
19.12.3.4 m_pfPanic
19.12.3.5 m_pfThreadContextCallout
19.12.3.6 m_pfThreadCreateCallout
19.12.3.7 m_pfThreadExitCallout
19.12.3.8 m_u16GuardThreshold
19.12.3.9 m_u32Ticks
19.13Mark3::KernelSWI Class Reference
19.13.1 Detailed Description
19.13.2 Member Function Documentation
19.13.2.1 Config()
19.13.2.2 Start()
19.13.2.3 Trigger()
19.14Mark3::KernelTimer Class Reference
19.14.1 Detailed Description
19.14.2 Member Function Documentation
19.14.2.1 Config()
19.14.2.2 Start()
19.14.2.3 Stop()
19.15Mark3::LinkList Class Reference
19.15.1 Detailed Description
19.15.2 Member Function Documentation
19.15.2.1 GetHead()
19.15.2.2 GetTail()
19.15.2.3 Init()

CONTENTS xiii

19.15.2.4 SetHead()	54
19.15.2.5 SetTail()	54
19.15.3 Member Data Documentation	54
19.15.3.1 m_pclHead	55
19.15.3.2 m_pclTail	55
19.16Mark3::LinkListNode Class Reference	55
19.16.1 Detailed Description	56
19.16.2 Constructor & Destructor Documentation	56
19.16.2.1 LinkListNode()	56
19.16.3 Member Function Documentation	56
19.16.3.1 ClearNode()	56
19.16.3.2 GetNext()	57
19.16.3.3 GetPrev()	57
19.16.4 Friends And Related Function Documentation	57
19.16.4.1 CircularLinkList	57
19.16.4.2 DoubleLinkList	57
19.16.4.3 LinkList	58
19.16.5 Member Data Documentation	58
19.16.5.1 next	58
19.16.5.2 prev	58
19.17Mark3::LockGuard Class Reference	58
19.17.1 Detailed Description	59
19.17.2 Constructor & Destructor Documentation	59
19.17.2.1 LockGuard() [1/2]	59
19.17.2.2 LockGuard() [2/2]	59
19.17.2.3 ~LockGuard()	60
19.17.3 Member Function Documentation	60
19.17.3.1 isAcquired()	60
19.17.4 Member Data Documentation	60
19.17.4.1 m_blsAcquired	60

xiv CONTENTS

19.17.4.2 m_pclMutex	60
19.18Mark3::Mailbox Class Reference	61
19.18.1 Detailed Description	62
19.18.2 Constructor & Destructor Documentation	63
19.18.2.1 ~Mailbox()	63
19.18.3 Member Function Documentation	63
19.18.3.1 CopyData()	63
19.18.3.2 GetFreeSlots()	63
19.18.3.3 GetHeadPointer()	64
19.18.3.4 GetTailPointer()	64
19.18.3.5 Init() [1/2]	64
19.18.3.6 Init() [2/2]	65
19.18.3.7 IsEmpty()	65
19.18.3.8 IsFull()	65
19.18.3.9 MoveHeadBackward()	65
19.18.3.10MoveHeadForward()	66
19.18.3.11MoveTailBackward()	66
19.18.3.12MoveTailForward()	66
19.18.3.13operator new()	66
19.18.3.14Receive() [1/2]	66
19.18.3.15Receive() [2/2]	67
19.18.3.16Receive_i()	67
19.18.3.17ReceiveTail() [1/2]	68
19.18.3.18ReceiveTail() [2/2]	68
19.18.3.19Send() [1/2]	69
19.18.3.20Send() [2/2]	69
19.18.3.21Send_i()	70
19.18.3.2\(\times\)endTail() [1/2]	70
19.18.3.23SendTail() [2/2]	71
19.18.4 Member Data Documentation	71

CONTENTS xv

19.18.4.1 m_clRecvSem	 171
19.18.4.2 m_clSendSem	 171
19.18.4.3 m_pvBuffer	 172
19.18.4.4 m_u16Count	 172
19.18.4.5 m_u16ElementSize	 172
19.18.4.6 m_u16Free	 172
19.18.4.7 m_u16Head	 172
19.18.4.8 m_u16Tail	 173
19.19Mark3::MemUtil Class Reference	 173
19.19.1 Detailed Description	 174
19.19.2 Member Function Documentation	 174
19.19.2.1 Checksum16()	 174
19.19.2.2 Checksum8()	 175
19.19.2.3 CompareMemory()	 175
19.19.2.4 CompareStrings() [1/2]	 176
19.19.2.5 CompareStrings() [2/2]	 176
19.19.2.6 CopyMemory()	 176
19.19.2.7 CopyString()	 177
19.19.2.8 DecimalToHex() [1/4]	 177
19.19.2.9 DecimalToHex() [2/4]	 178
19.19.2.1@ecimalToHex() [3/4]	 178
19.19.2.11DecimalToHex() [4/4]	 178
19.19.2.12DecimalToString() [1/4]	 178
19.19.2.13DecimalToString() [2/4]	 179
19.19.2.14DecimalToString() [3/4]	 179
19.19.2.15DecimalToString() [4/4]	 179
19.19.2.16SetMemory()	 179
19.19.2.17StringLength()	 180
19.19.2.1&tringSearch()	 180
19.19.2.1%tringToDecimal16()	 181

xvi CONTENTS

19.19.2.20StringToDecimal32()	1
19.19.2.21StringToDecimal64()	1
19.19.2.22StringToDecimal8()	1
19.19.2.23Tokenize()	2
19.20Mark3::Message Class Reference	2
19.20.1 Detailed Description	3
19.20.2 Member Function Documentation	3
19.20.2.1 GetCode()	3
19.20.2.2 GetData()	4
19.20.2.3 Init()	4
19.20.2.4 operator new()	4
19.20.2.5 SetCode()	4
19.20.2.6 SetData()	5
19.20.3 Member Data Documentation	5
19.20.3.1 m_pvData	5
19.20.3.2 m_u16Code	5
19.21 Mark3::MessagePool Class Reference	6
19.21.1 Detailed Description	6
19.21.2 Constructor & Destructor Documentation	6
19.21.2.1 ~MessagePool()	6
19.21.3 Member Function Documentation	37
19.21.3.1 GetHead()	۶7
19.21.3.2 Init()	١7
19.21.3.3 operator new()	۶7
19.21.3.4 Pop()	8
19.21.3.5 Push()	8
19.21.4 Member Data Documentation	8
19.21.4.1 m_clList	9
19.22Mark3::MessageQueue Class Reference	9
19.22.1 Detailed Description	10

CONTENTS xvii

19.22.2 Constructor & Destructor Documentation	190
19.22.2.1 ~MessageQueue()	190
19.22.3 Member Function Documentation	190
19.22.3.1 GetCount()	190
19.22.3.2 Init()	190
19.22.3.3 operator new()	191
19.22.3.4 Receive() [1/2]	191
19.22.3.5 Receive() [2/2] 1	191
19.22.3.6 Receive_i()	192
19.22.3.7 Send()	192
19.22.4 Member Data Documentation	192
19.22.4.1 m_clLinkList	193
19.22.4.2 m_clSemaphore	193
19.23Mark3::Mutex Class Reference	193
19.23.1 Detailed Description	194
19.23.2 Constructor & Destructor Documentation	194
19.23.2.1 ~Mutex()	194
19.23.3 Member Function Documentation	194
19.23.3.1 Claim() [1/2]	195
19.23.3.2 Claim() [2/2]	195
19.23.3.3 Claim_i()	195
19.23.3.4 Init()	196
19.23.3.5 operator new()	196
19.23.3.6 Release()	196
19.23.3.7 WakeMe()	197
19.23.3.8 WakeNext()	197
19.23.4 Member Data Documentation	197
19.23.4.1 m_bReady	197
19.23.4.2 m_bRecursive	198
19.23.4.3 m_pclOwner	198

xviii CONTENTS

19.23.4.4 m_u8Recurse	 198
19.23.4.5 m_uMaxPri	 198
19.24Mark3::Notify Class Reference	 199
19.24.1 Detailed Description	 199
19.24.2 Constructor & Destructor Documentation	 200
19.24.2.1 ~Notify()	 200
19.24.3 Member Function Documentation	 200
19.24.3.1 Init()	 200
19.24.3.2 operator new()	 200
19.24.3.3 Signal()	 200
19.24.3.4 Wait() [1/2]	 200
19.24.3.5 Wait() [2/2]	 201
19.24.3.6 WakeMe()	 201
19.24.4 Member Data Documentation	 202
19.24.4.1 m_bPending	 202
19.25Mark3::PriorityMapL1< T, C > Class Template Reference	 202
19.25Mark3::PriorityMapL1 < T, C > Class Template Reference	
	 203
19.25.1 Detailed Description	 203 203
19.25.1 Detailed Description	 <ul><li>203</li><li>203</li><li>203</li></ul>
19.25.1 Detailed Description	 <ul><li>203</li><li>203</li><li>203</li><li>203</li></ul>
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation	 <ul><li>203</li><li>203</li><li>203</li><li>203</li><li>203</li></ul>
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation	<ul><li>203</li><li>203</li><li>203</li><li>203</li><li>204</li></ul>
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()	203 203 203 203 203 204 204
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()  19.25.3.3 PrioBit()	203 203 203 203 203 204 204
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()  19.25.3.3 PrioBit()  19.25.3.4 PrioMapWordIndex()	203 203 203 203 204 204 204 204
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()  19.25.3.3 PrioBit()  19.25.3.4 PrioMapWordIndex()  19.25.3.5 PriorityFromBitmap()	203 203 203 203 204 204 204 204 204
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()  19.25.3.3 PrioBit()  19.25.3.4 PrioMapWordIndex()  19.25.3.5 PriorityFromBitmap()  19.25.3.6 Set()	203 203 203 203 204 204 204 204 204 205
19.25.1 Detailed Description  19.25.2 Constructor & Destructor Documentation  19.25.2.1 PriorityMapL1()  19.25.3 Member Function Documentation  19.25.3.1 Clear()  19.25.3.2 HighestPriority()  19.25.3.3 PrioBit()  19.25.3.4 PrioMapWordIndex()  19.25.3.5 PriorityFromBitmap()  19.25.3.6 Set()  19.25.4 Member Data Documentation	203 203 203 203 204 204 204 204 205 205

CONTENTS xix

19.25.4.4 m_uXPrioMapWordShift
19.25.4.5 m_uXPriorityMap
19.26 Mark3::PriorityMapL2 < T, C > Class Template Reference
19.26.1 Detailed Description
19.26.2 Constructor & Destructor Documentation
19.26.2.1 PriorityMapL2()
19.26.3 Member Function Documentation
19.26.3.1 Clear()
19.26.3.2 HighestPriority()
19.26.3.3 PrioBit()
19.26.3.4 PrioMapWordIndex()
19.26.3.5 PriorityFromBitmap()
19.26.3.6 Set()
19.26.4 Member Data Documentation
19.26.4.1 m_auXPriorityMap
19.26.4.2 m_uXPrioMapBitMask
19.26.4.3 m_uXPrioMapBits
19.26.4.4 m_uXPrioMapNumWords
19.26.4.5 m_uXPrioMapShiftLUT
19.26.4.6 m_uXPrioMapWordShift
19.26.4.7 m_uXPriorityMapL2
19.27Mark3::ProfileTimer Class Reference
19.27.1 Detailed Description
19.27.2 Member Function Documentation
19.27.2.1 GetAverage()
19.27.2.2 GetCurrent()
19.27.2.3 Init()
19.27.2.4 Start()
19.27.2.5 Stop()
19.27.3 Member Data Documentation

19.27.3.1 m_bActive	12
19.27.3.2 m_u16Iterations	13
19.27.3.3 m_u32Cumulative	13
19.27.3.4 m_u32CurrentIteration	13
19.27.3.5 m_u32StartTicks	13
19.28Mark3::Quantum Class Reference	13
19.28.1 Detailed Description	14
19.28.2 Member Function Documentation	14
19.28.2.1 Cancel()	14
19.28.2.2 ClearInTimer()	14
19.28.2.3 Init()	15
19.28.2.4 SetInTimer()	15
19.28.2.5 SetTimerThread()	15
19.28.2.6 Update()	15
19.28.3 Member Data Documentation	16
19.28.3.1 m_blnTimer	16
19.28.3.2 m_pclActiveThread	16
19.28.3.3 m_pclTimerThread	16
19.28.3.4 m_u16TicksRemain	16
19.29Mark3::ReaderWriterLock Class Reference	16
19.29.1 Detailed Description	17
19.29.2 Member Function Documentation	18
19.29.2.1 AcquireReader() [1/2]	18
19.29.2.2 AcquireReader() [2/2]	18
19.29.2.3 AcquireReader_i()	18
19.29.2.4 AcquireWriter() [1/2]	19
19.29.2.5 AcquireWriter() [2/2]	19
19.29.2.6 AcquireWriter_i()	19
19.29.2.7 Init()	20
19.29.2.8 operator new()	20

CONTENTS xxi

19.29.2.9 ReleaseReader()	20
19.29.2.10ReleaseWriter()	20
19.29.3 Member Data Documentation	21
19.29.3.1 m_clGlobalMutex	21
19.29.3.2 m_clReaderMutex	21
19.29.3.3 m_u8ReadCount	21
19.30 Mark3::Scheduler Class Reference	21
19.30.1 Detailed Description	22
19.30.2 Member Function Documentation	23
19.30.2.1 Add()	23
19.30.2.2 GetCurrentThread()	23
19.30.2.3 GetNextThread()	23
19.30.2.4 GetStopList()	24
19.30.2.5 GetThreadList()	24
19.30.2.6 Init()	24
19.30.2.7 lsEnabled()	25
19.30.2.8 QueueScheduler()	25
19.30.2.9 Remove()	25
19.30.2.10Schedule()	25
19.30.2.11SetScheduler()	26
19.30.3 Member Data Documentation	26
19.30.3.1 m_aclPriorities	26
19.30.3.2 m_bEnabled	26
19.30.3.3 m_bQueuedSchedule	26
19.30.3.4 m_clPrioMap	27
19.30.3.5 m_clStopList	27
19.30.3.6 m_uNumPriorities	27
19.31 Mark3::SchedulerGuard Class Reference	27
19.31.1 Detailed Description	28
19.31.2 Constructor & Destructor Documentation	28

xxii CONTENTS

19.31.2.1 SchedulerGuard()
19.31.2.2 ~SchedulerGuard()
19.31.3 Member Data Documentation
19.31.3.1 m_bSchedState
19.32Mark3::Semaphore Class Reference
19.32.1 Detailed Description
19.32.2 Constructor & Destructor Documentation
19.32.2.1 ~Semaphore()
19.32.3 Member Function Documentation
19.32.3.1 GetCount()
19.32.3.2 Init()
19.32.3.3 operator new()
19.32.3.4 Pend() [1/2]
19.32.3.5 Pend() [2/2]
19.32.3.6 Pend_i()
19.32.3.7 Post()
19.32.3.8 WakeMe()
19.32.3.9 WakeNext()
19.32.4 Member Data Documentation
19.32.4.1 m_u16MaxValue
19.32.4.2 m_u16Value
19.33Mark3::Streamer Class Reference
19.33.1 Detailed Description
19.33.2 Member Function Documentation
19.33.2.1 CanRead()
19.33.2.2 CanWrite()
19.33.2.3 Claim()
19.33.2.4 GetAvailable()
19.33.2.5 Init()
19.33.2.6 IsEmpty()

CONTENTS xxiii

 238
 238
 239
 239
 239
 240
 240
 240
 240
 241
 241
 241
 241
 242
 245
 245
 245
 246
 247
 247
 247
 248
 248
 248
 248
 249

xxiv CONTENTS

19.34.3.12GetOwner()	249
19.34.3.13GetPriority()	249
19.34.3.14GetQuantum()	250
19.34.3.15GetStack()	250
19.34.3.16GetStackSize()	250
19.34.3.17GetStackSlack()	251
19.34.3.18GetState()	251
19.34.3.19GetTimer()	251
19.34.3.20nheritPriority()	251
19.34.3.21Init() [1/2]	252
<b>19.34.3.22</b> nit() [2/2]	252
19.34.3.23sInitialized()	253
19.34.3.24operator new()	253
19.34.3.25SetCurrent()	253
19.34.3.26SetEventFlagMask()	254
19.34.3.27SetEventFlagMode()	254
19.34.3.2&etExpired()	254
19.34.3.29SetExtendedContext()	255
19.34.3.30SetID()	255
19.34.3.31SetName()	255
19.34.3.32SetOwner()	257
19.34.3.33SetPriority()	257
19.34.3.34SetPriorityBase()	257
19.34.3.35SetQuantum()	258
19.34.3.36SetState()	258
19.34.3.37Sleep()	258
19.34.3.38Start()	259
19.34.3.39Stop()	259
19.34.3.40Yield()	259
19.34.4 Friends And Related Function Documentation	260

CONTENTS xxv

19.34.4.1 ThreadPort	 260
19.34.5 Member Data Documentation	 260
19.34.5.1 m_bExpired	 260
19.34.5.2 m_clTimer	 260
19.34.5.3 m_eFlagMode	 260
19.34.5.4 m_eState	 261
19.34.5.5 m_pclCurrent	 261
19.34.5.6 m_pclOwner	 261
19.34.5.7 m_pfEntryPoint	 261
19.34.5.8 m_pvArg	 261
19.34.5.9 m_pvExtendedContext	 262
19.34.5.10m_pwStack	 262
19.34.5.11m_pwStackTop	 262
19.34.5.12m_szName	 262
19.34.5.13m_u16FlagMask	 262
19.34.5.14m_u16Quantum	 263
19.34.5.15m_u16StackSize	 263
19.34.5.16m_u8ThreadID	 263
19.34.5.17m_uXCurPriority	 263
19.34.5.18m_uXPriority	 263
19.35Mark3::ThreadList Class Reference	 264
19.35.1 Detailed Description	 265
19.35.2 Constructor & Destructor Documentation	 265
19.35.2.1 ThreadList()	 265
19.35.3 Member Function Documentation	 265
19.35.3.1 Add() [1/2]	 265
19.35.3.2 Add() [2/2]	 265
19.35.3.3 AddPriority()	 267
19.35.3.4 HighestWaiter()	 267
19.35.3.5 operator new()	 267

xxvi CONTENTS

19.35.3.6 Remove()
19.35.3.7 SetMapPointer()
19.35.3.8 SetPriority()
19.35.4 Member Data Documentation
19.35.4.1 m_pclMap
19.35.4.2 m_uXPriority
19.36Mark3::ThreadListList Class Reference
19.36.1 Detailed Description
19.36.2 Member Function Documentation
19.36.2.1 Add()
19.36.2.2 GetHead()
19.36.2.3 Remove()
19.36.3 Member Data Documentation
19.36.3.1 m_clThreadListList
19.37Mark3::ThreadPort Class Reference
19.37.1 Detailed Description
19.37.2 Member Function Documentation
19.37.2.1 Init()
19.37.2.2 InitStack()
19.37.2.3 StartThreads()
19.37.3 Friends And Related Function Documentation
19.37.3.1 Thread
19.38Mark3::Timer Class Reference
19.38.1 Detailed Description
19.38.2 Constructor & Destructor Documentation
19.38.2.1 ~Timer()
19.38.2.2 Timer()
19.38.3 Member Function Documentation
19.38.3.1 Init()
19.38.3.2 IsInitialized()

CONTENTS xxvii

	19.38.3.3 MSecondsToTicks()	276
	19.38.3.4 operator new()	276
	19.38.3.5 SecondsToTicks()	277
	19.38.3.6 SetInitialized()	277
	19.38.3.7 Start() [1/2]	277
	19.38.3.8 Start() [2/2]	277
	19.38.3.9 Stop()	278
	19.38.3.1@SecondsToTicks()	278
19.38.4	Friends And Related Function Documentation	278
	19.38.4.1 TimerList	278
19.38.5	Member Data Documentation	278
	19.38.5.1 m_pclOwner	278
	19.38.5.2 m_pfCallback	279
	19.38.5.3 m_pvData	279
	19.38.5.4 m_u32Interval	279
	19.38.5.5 m_u32TimeLeft	279
	19.38.5.6 m_u8Flags	279
	19.38.5.7 m_u8Initialized	280
	19.38.5.8 m_uTimerInitCookie	280
	19.38.5.9 m_uTimerInvalidCookie	280
19.39Mark3::7	FimerList Class Reference	280
19.39.1	Detailed Description	281
19.39.2	Member Function Documentation	281
	19.39.2.1 Add()	281
	19.39.2.2 Init()	282
	19.39.2.3 Process()	282
	19.39.2.4 Remove()	282
19.39.3	Member Data Documentation	282
	19.39.3.1 m_bTimerActive	282
	19.39.3.2 m_clMutex	283

xxviii CONTENTS

19.39.3.3 m_u32NextWakeup	 283
19.40Mark3::TimerScheduler Class Reference	 283
19.40.1 Detailed Description	 284
19.40.2 Member Function Documentation	 284
19.40.2.1 Add()	 284
19.40.2.2 Init()	 284
19.40.2.3 Process()	 284
19.40.2.4 Remove()	 285
19.40.3 Member Data Documentation	 285
19.40.3.1 m_clTimerList	 285
19.41Mark3::Token_t Struct Reference	 285
19.41.1 Detailed Description	 286
19.41.2 Member Data Documentation	 286
19.41.2.1 pcToken	 286
19.41.2.2 u8Len	 286
19.42Mark3::TypedCircularLinkList< T > Class Template Reference	 286
19.42Mark3::TypedCircularLinkList< T > Class Template Reference	
	 287
19.42.1 Detailed Description	 287 287
19.42.1 Detailed Description	 287 287 287
19.42.1 Detailed Description	 287 287 287 287
19.42.1 Detailed Description	 287 287 287 287 287
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation	 287 287 287 287 287 288
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()	 287 287 287 287 287 288 288
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()  19.42.3.3 GetTail()	 287 287 287 287 288 288 288
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()  19.42.3.3 GetTail()  19.42.3.4 InsertNodeBefore()	 287 287 287 287 288 288 288
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()  19.42.3.3 GetTail()  19.42.3.4 InsertNodeBefore()  19.42.3.5 operator new()	287 287 287 287 288 288 288 289
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()  19.42.3.3 GetTail()  19.42.3.4 InsertNodeBefore()  19.42.3.5 operator new()  19.42.3.6 Remove()	287 287 287 287 288 288 288 289 289
19.42.1 Detailed Description  19.42.2 Constructor & Destructor Documentation  19.42.2.1 TypedCircularLinkList()  19.42.3 Member Function Documentation  19.42.3.1 Add()  19.42.3.2 GetHead()  19.42.3.3 GetTail()  19.42.3.4 InsertNodeBefore()  19.42.3.5 operator new()  19.42.3.6 Remove()  19.42.3.7 SetHead()	287 287 287 287 288 288 288 289 289 289

CONTENTS xxix

		19.43.2	2 Constructor & Destructor Documentation	. 291
			19.43.2.1 TypedDoubleLinkList()	. 291
		19.43.3	8 Member Function Documentation	. 291
			19.43.3.1 Add()	. 291
			19.43.3.2 GetHead()	. 292
			19.43.3.3 GetTail()	. 292
			19.43.3.4 operator new()	. 292
			19.43.3.5 Remove()	. 292
			19.43.3.6 SetHead()	. 293
			19.43.3.7 SetTail()	. 293
	19.4	4Mark3:	:TypedLinkListNode < T > Class Template Reference	. 293
		19.44.1	Detailed Description	. 294
		19.44.2	2 Member Function Documentation	. 294
			19.44.2.1 GetNext()	. 294
			19.44.2.2 GetPrev()	. 294
20	File	Docume	entation	295
20			entation moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	
20				. 295
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295 . 299
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295 . 299 . 300
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295 . 299 . 300 . 300
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295 . 299 . 300 . 300 . 300
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()	. 295 . 299 . 300 . 300 . 300
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference	. 295 . 299 . 300 . 300 . 300 . 300
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()  20.1.2.5 Alloc_Message()  20.1.2.6 Alloc_MessagePool()	. 295 . 295 . 300 . 300 . 300 . 301 . 301
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()  20.1.2.5 Alloc_Message()  20.1.2.6 Alloc_MessagePool()  20.1.2.7 Alloc_MessageQueue()	. 295 . 299 . 300 . 300 . 300 . 301 . 301
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()  20.1.2.5 Alloc_Message()  20.1.2.6 Alloc_MessagePool()  20.1.2.7 Alloc_MessageQueue()  20.1.2.8 Alloc_Mutex()	. 295 . 295 . 300 . 300 . 300 . 301 . 301 . 301
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()  20.1.2.5 Alloc_Message()  20.1.2.6 Alloc_MessagePool()  20.1.2.7 Alloc_MessageQueue()  20.1.2.8 Alloc_Mutex()  20.1.2.9 Alloc_Notify()	. 295 . 295 . 300 . 300 . 300 . 301 . 301 . 302 . 302
20		/home/ 20.1.1	moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference  Detailed Description  Function Documentation  20.1.2.1 Alloc_ConditionVariable()  20.1.2.2 Alloc_EventFlag()  20.1.2.3 Alloc_Mailbox()  20.1.2.4 Alloc_Memory()  20.1.2.5 Alloc_Message()  20.1.2.6 Alloc_MessagePool()  20.1.2.7 Alloc_MessageQueue()  20.1.2.8 Alloc_Mutex()	. 295 . 295 . 300 . 300 . 300 . 301 . 301 . 302 . 302

20.1.2.12 Alloc_Thread()
20.1.2.13 Alloc_Timer()
20.1.2.14 Atomic_Add16()
20.1.2.15 Atomic_Add32()
20.1.2.16 Atomic_Add8()
20.1.2.17 Atomic_Set16()
20.1.2.18 Atomic_Set32()
20.1.2.19 Atomic_Set8()
20.1.2.20 Atomic_Sub16()
20.1.2.21 Atomic_Sub32()
20.1.2.22 Atomic_Sub8()
20.1.2.23 Atomic_TestAndSet()
20.1.2.24 ConditionVariable_Broadcast()
20.1.2.25 ConditionVariable_Init()
20.1.2.26 ConditionVariable_Signal()
20.1.2.27 ConditionVariable_TimedWait()
20.1.2.28 ConditionVariable_Wait()
20.1.2.29 Free_ConditionVariable()
20.1.2.30 Free_EventFlag()
20.1.2.31 Free_Mailbox()
20.1.2.32 Free_Memory()
20.1.2.33 Free_Message()
20.1.2.34 Free_MessagePool()
20.1.2.35 Free_MessageQueue()
20.1.2.36 Free_Mutex()
20.1.2.37 Free_Notify()
20.1.2.38 Free_ReaderWriterLock()
20.1.2.39 Free_Semaphore()
20.1.2.40 Free_Thread()
20.1.2.41 Free_Timer()

CONTENTS xxxi

20.1.2.42 Kernel_DebugPrint()
20.1.2.43 Kernel_GetStackGuardThreshold()
20.1.2.44 Kernel_GetTicks()
20.1.2.45 Kernel_Init()
20.1.2.46 Kernel_lsPanic()
20.1.2.47 Kernel_lsStarted()
20.1.2.48 Kernel_Panic()
20.1.2.49 Kernel_SetDebugPrintFunction()
20.1.2.50 Kernel_SetPanic()
20.1.2.51 Kernel_SetStackGuardThreshold()
20.1.2.52 Kernel_Start()
20.1.2.53 Mailbox_GetFreeSlots()
20.1.2.54 Mailbox_Init()
20.1.2.55 Mailbox_lsEmpty()
20.1.2.56 Mailbox_lsFull()
20.1.2.57 Mailbox_Receive()
20.1.2.58 Mailbox_ReceiveTail()
20.1.2.59 Mailbox_Send()
20.1.2.60 Mailbox_SendTail()
20.1.2.61 Mailbox_TimedReceive()
20.1.2.62 Mailbox_TimedReceiveTail()
20.1.2.63 Mailbox_TimedSend()
20.1.2.64 Mailbox_TimedSendTail()
20.1.2.65 Message_GetCode()
20.1.2.66 Message_GetData()
20.1.2.67 Message_Init()
20.1.2.68 Message_SetCode()
20.1.2.69 Message_SetData()
20.1.2.70 MessagePool_Init()
20.1.2.71 MessagePool_Pop()

xxxii CONTENTS

20.1.2.72 MessagePool_Push()
20.1.2.73 MessageQueue_GetCount()
20.1.2.74 MessageQueue_Init()
20.1.2.75 MessageQueue_Receive()
20.1.2.76 MessageQueue_Send()
20.1.2.77 MessageQueue_TimedReceive()
20.1.2.78 Mutex_Claim()
20.1.2.79 Mutex_Init()
20.1.2.80 Mutex_Release()
20.1.2.81 Mutex_TimedClaim()
20.1.2.82 Notify_Init()
20.1.2.83 Notify_Signal()
20.1.2.84 Notify_TimedWait()
20.1.2.85 Notify_Wait()
20.1.2.86 ReaderWriterLock_AcquireReader()
20.1.2.87 ReaderWriterLock_AcquireWriter()
20.1.2.88 ReaderWriterLock_Init()
20.1.2.89 ReaderWriterLock_ReleaseReader()
20.1.2.90 ReaderWriterLock_ReleaseWriter()
20.1.2.91 ReaderWriterLock_TimedAcquireReader()
20.1.2.92 ReaderWriterLock_TimedAcquireWriter()
20.1.2.93 Scheduler_Enable()
20.1.2.94 Scheduler_GetCurrentThread()
20.1.2.95 Scheduler_IsEnabled()
20.1.2.96 Semaphore_Init()
20.1.2.97 Semaphore_Pend()
20.1.2.98 Semaphore_Post()
20.1.2.99 Semaphore_TimedPend()
20.1.2.100Thread_CoopYield()
20.1.2.101Thread_Exit()

CONTENTS xxxiii

20.1.2.102Thread_GetCurPriority()	339
20.1.2.103Thread_GetID()	339
20.1.2.104Thread_GetName()	340
20.1.2.105Thread_GetPriority()	340
20.1.2.106Thread_GetQuantum()	340
20.1.2.107Thread_GetStackSlack()	341
20.1.2.108Thread_GetState()	341
20.1.2.109Thread_Init()	341
20.1.2.110Thread_SetID()	342
20.1.2.111Thread_SetName()	342
20.1.2.112Thread_SetPriority()	343
20.1.2.113Thread_SetQuantum()	343
20.1.2.114Thread_Sleep()	343
20.1.2.115Thread_Start()	344
20.1.2.116Thread_Stop()	344
20.1.2.117Thread_Yield()	344
20.1.2.118Timer_Init()	345
20.1.2.119Timer_Restart()	345
20.1.2.120Timer_Start()	345
20.1.2.121Timer_Stop()	346
20.2 mark3c.cpp	346
20.3 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/fake_types.h File Reference	358
20.3.1 Detailed Description	358
20.4 fake_types.h	358
20.5 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/mark3c.h File Reference	360
20.5.1 Detailed Description	366
20.5.2 Macro Definition Documentation	366
20.5.2.1 CONDITIONVARIABLE_SIZE	366
20.5.2.2 DECLARE_CONDITIONVARIABLE	366
20.5.2.3 DECLARE_MAILBOX	366

	20.5.2.4	DECLARE_MESSAGE	 367
	20.5.2.5	DECLARE_MESSAGEPOOL	 367
	20.5.2.6	DECLARE_MESSAGEQUEUE	 367
	20.5.2.7	DECLARE_MUTEX	 368
	20.5.2.8	DECLARE_NOTIFY	 368
	20.5.2.9	DECLARE_READERWRITERLOCK	 368
	20.5.2.10	DECLARE_SEMAPHORE	 369
	20.5.2.11	DECLARE_THREAD	 369
	20.5.2.12	DECLARE_TIMER	 369
	20.5.2.13	MAILBOX_SIZE	 369
	20.5.2.14	MESSAGE_SIZE	 370
	20.5.2.15	MESSAGEPOOL_SIZE	 370
	20.5.2.16	MESSAGEQUEUE_SIZE	 370
	20.5.2.17	MUTEX_SIZE	 370
	20.5.2.18	NOTIFY_SIZE	 370
	20.5.2.19	READERWRITERLOCK_SIZE	 370
	20.5.2.20	SEMAPHORE_SIZE	 371
	20.5.2.21	THREAD_SIZE	 371
	20.5.2.22	? TIMER_SIZE	 371
	20.5.2.23	TOKEN_1	 371
	20.5.2.24	TOKEN_2	 371
	20.5.2.25	WORD_ROUND	 371
20.5.3	Typedef D	Documentation	 372
	20.5.3.1	ConditionVariable_t	 372
	20.5.3.2	kernel_debug_print_t	 372
	20.5.3.3	Mailbox_t	 372
	20.5.3.4	Message_t	 372
	20.5.3.5	MessagePool_t	 372
	20.5.3.6	MessageQueue_t	 373
	20.5.3.7	Mutex_t	 373

CONTENTS XXXV

	20.5.3.8 Notify_t	373
	20.5.3.9 panic_func_t	373
	20.5.3.10 ReaderWriterLock_t	373
	20.5.3.11 Semaphore_t	374
	20.5.3.12 thread_entry_func_t	374
	20.5.3.13 Thread_t	374
	20.5.3.14 timer_callback_t	374
	20.5.3.15 Timer_t	374
20.5.4	Enumeration Type Documentation	374
	20.5.4.1 thread_state_t	374
20.5.5	Function Documentation	375
	20.5.5.1 Alloc_Mailbox()	375
	20.5.5.2 Alloc_Memory()	375
	20.5.5.3 Alloc_Message()	376
	20.5.5.4 Alloc_MessagePool()	376
	20.5.5.5 Alloc_MessageQueue()	376
	20.5.5.6 Alloc_Mutex()	377
	20.5.5.7 Alloc_Notify()	377
	20.5.5.8 Alloc_Semaphore()	377
	20.5.5.9 Alloc_Thread()	378
	20.5.5.10 Alloc_Timer()	378
	20.5.5.11 Atomic_Add16()	378
	20.5.5.12 Atomic_Add32()	379
	20.5.5.13 Atomic_Add8()	379
	20.5.5.14 Atomic_Set16()	380
	20.5.5.15 Atomic_Set32()	380
	20.5.5.16 Atomic_Set8()	381
	20.5.5.17 Atomic_Sub16()	381
	20.5.5.18 Atomic_Sub32()	382
	20.5.5.19 Atomic_Sub8()	382

xxxvi CONTENTS

20.5.5.20 Atomic_TestAndSet()
20.5.5.21 ConditionVariable_Broadcast()
20.5.5.22 ConditionVariable_Init()
20.5.5.23 ConditionVariable_Signal()
20.5.5.24 ConditionVariable_TimedWait()
20.5.5.25 ConditionVariable_Wait()
20.5.5.26 Free_Mailbox()
20.5.5.27 Free_Memory()
20.5.5.28 Free_Message()
20.5.5.29 Free_MessagePool()
20.5.5.30 Free_MessageQueue()
20.5.5.31 Free_Mutex()
20.5.5.32 Free_Notify()
20.5.5.33 Free_Semaphore()
20.5.5.34 Free_Thread()
20.5.5.35 Free_Timer()
20.5.5.36 Kernel_DebugPrint()
20.5.5.37 Kernel_GetTicks()
20.5.5.38 Kernel_Init()
20.5.5.39 Kernel_IsPanic()
20.5.5.40 Kernel_lsStarted()
20.5.5.41 Kernel_Panic()
20.5.5.42 Kernel_SetDebugPrintFunction()
20.5.5.43 Kernel_SetPanic()
20.5.5.44 Kernel_Start()
20.5.5.45 Mailbox_GetFreeSlots()
20.5.5.46 Mailbox_Init()
20.5.5.47 Mailbox_IsEmpty()
20.5.5.48 Mailbox_lsFull()
20.5.5.49 Mailbox_Receive()

CONTENTS xxxvii

20.5.5.50 Mailbox_ReceiveTail()
20.5.5.51 Mailbox_Send()
20.5.5.52 Mailbox_SendTail()
20.5.5.53 Mailbox_TimedReceive()
20.5.5.54 Mailbox_TimedReceiveTail()
20.5.5.55 Mailbox_TimedSend()
20.5.5.56 Mailbox_TimedSendTail()
20.5.5.57 Message_GetCode()
20.5.5.58 Message_GetData()
20.5.5.59 Message_Init()
20.5.5.60 Message_SetCode()
20.5.5.61 Message_SetData()
20.5.5.62 MessagePool_Init()
20.5.5.63 MessagePool_Pop()
20.5.5.64 MessagePool_Push()
20.5.5.65 MessageQueue_GetCount()
20.5.5.66 MessageQueue_Init()
20.5.5.67 MessageQueue_Receive()
20.5.5.68 MessageQueue_Send()
20.5.5.69 MessageQueue_TimedReceive()
20.5.5.70 Mutex_Claim()
20.5.5.71 Mutex_Init()
20.5.5.72 Mutex_Release()
20.5.5.73 Mutex_TimedClaim()
20.5.5.74 Notify_Init()
20.5.5.75 Notify_Signal()
20.5.5.76 Notify_TimedWait()
20.5.5.77 Notify_Wait()
20.5.5.78 ReaderWriterLock_AcquireReader()
20.5.5.79 ReaderWriterLock_AcquireWriter()

xxxviii CONTENTS

	20.5.5.80 ReaderWriterLock_Init()	407
	20.5.5.81 ReaderWriterLock_ReleaseReader()	407
	20.5.5.82 ReaderWriterLock_ReleaseWriter()	408
	20.5.5.83 ReaderWriterLock_TimedAcquireReader()	408
	20.5.5.84 ReaderWriterLock_TimedAcquireWriter()	408
	20.5.5.85 Scheduler_Enable()	409
	20.5.5.86 Scheduler_GetCurrentThread()	409
	20.5.5.87 Scheduler_IsEnabled()	410
	20.5.5.88 Semaphore_Init()	410
	20.5.5.89 Semaphore_Pend()	411
	20.5.5.90 Semaphore_Post()	411
	20.5.5.91 Semaphore_TimedPend()	411
	20.5.5.92 Thread_CoopYield()	412
	20.5.5.93 Thread_Exit()	412
	20.5.5.94 Thread_GetCurPriority()	413
	20.5.5.95 Thread_GetID()	413
	20.5.5.96 Thread_GetPriority()	413
	20.5.5.97 Thread_GetState()	414
	20.5.5.98 Thread_Init()	414
	20.5.5.99 Thread_SetID()	415
	20.5.5.100Thread_SetPriority()	415
	20.5.5.101Thread_Sleep()	416
	20.5.5.102Thread_Start()	416
	20.5.5.103Thread_Stop()	417
	20.5.5.104Thread_Yield()	417
	20.5.5.105Timer_Init()	417
	20.5.5.106Timer_Restart()	418
	20.5.5.107Timer_Start()	418
	20.5.5.108Timer_Stop()	419
20.6 mark3	c.h	419

CONTENTS xxxix

20.7 /home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp File Reference
20.7.1 Detailed Description
20.8 memutil.cpp
20.9 /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h File Reference
20.9.1 Detailed Description
20.10memutil.h
20.11/home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h File Reference 433
20.11.1 Detailed Description
20.12streamer.h
20.13/home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp File Reference
20.13.1 Detailed Description
20.14streamer.cpp
20.15/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp File Reference 438
20.15.1 Detailed Description
20.16kernelswi.cpp
20.17/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp File Reference
20.17.1 Detailed Description
20.17.2 Macro Definition Documentation
20.17.2.1 TCCR1B_INIT
20.17.2.2 TIMER_IFR
20.17.2.3 TIMER_IMSK
20.17.3 Function Documentation
20.17.3.1 ISR()
20.17.3.1 ISR()
20.18kerneltimer.cpp
20.18kerneltimer.cpp
20.18kerneltimer.cpp

<u>xI</u> CONTENTS

20.19.2.3 K_INT	443
20.19.2.4 K_WORD	443
20.19.2.5 KERNEL_NUM_PRIORITIES	444
20.19.2.6 KERNEL_STACK_GUARD_DEFAULT	444
20.19.2.7 KERNEL_TIMERS_THREAD_PRIORITY	444
20.19.2.8 PORT_COROUTINE_PRIORITIES	444
20.19.2.9 PORT_KERNEL_DEFAULT_STACK_SIZE	444
20.19.2.10PORT_KERNEL_TIMERS_THREAD_STACK	445
20.19.2.11PORT_MIN_TIMER_TICKS	445
20.19.2.12PORT_OVERLOAD_NEW	445
20.19.2.13PORT_PRIO_MAP_WORD_SIZE	445
20.19.2.14PORT_PRIO_TYPE	445
20.19.2.15PORT_STACK_GROWS_DOWN	446
20.19.2.16PORT_SYSTEM_FREQ	446
20.19.2.17PORT_TIMER_COUNT_TYPE	446
20.19.2.18PORT_TIMER_FREQ	446
20.19.2.19PORT_USE_HW_CLZ	447
20.19.2.20THREAD_QUANTUM_DEFAULT	447
20.20portcfg.h	447
20.21/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/threadport.h File Reference	448
20.21.1 Detailed Description	448
20.21.2 Macro Definition Documentation	449
20.21.2.1 ASM	449
20.21.2.2 PORT_PUSH_TO_STACK	449
20.21.2.3 PORT_TOP_OF_STACK	449
20.21.2.4 Thread_RestoreContext	449
20.21.2.5 Thread_SaveContext	450
20.22threadport.h	450
20.23/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp File Reference	452

CONTENTS xli

20.23.1 Detailed Description
20.23.2 Variable Documentation
20.23.2.1 g_kwCriticalCount
20.23.2.2 g_kwSFR
20.24threadport.cpp
20.25/home/moslevin/projects/m3-repo/kernel/src/atomic.cpp File Reference
20.25.1 Detailed Description
20.26atomic.cpp
20.27/home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp File Reference
20.27.1 Detailed Description
20.28autoalloc.cpp
20.29/home/moslevin/projects/m3-repo/kernel/src/blocking.cpp File Reference
20.29.1 Detailed Description
20.30blocking.cpp
20.31/home/moslevin/projects/m3-repo/kernel/src/colist.cpp File Reference
20.31.1 Detailed Description
20.32colist.cpp
20.33/home/moslevin/projects/m3-repo/kernel/src/condvar.cpp File Reference
20.33.1 Detailed Description
20.34condvar.cpp
20.35/home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp File Reference
20.35.1 Detailed Description
20.36coroutine.cpp
20.37/home/moslevin/projects/m3-repo/kernel/src/cosched.cpp File Reference
20.37.1 Detailed Description
20.38cosched.cpp
20.39/home/moslevin/projects/m3-repo/kernel/src/eventflag.cpp File Reference
20.39.1 Detailed Description
20.40 eventflag.cpp
20.41/home/moslevin/projects/m3-repo/kernel/src/kernel.cpp File Reference

xlii CONTENTS

20.41.1 Detailed Description
20.42kernel.cpp
20.43/home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp File Reference
20.43.1 Detailed Description
20.44ksemaphore.cpp
20.45/home/moslevin/projects/m3-repo/kernel/src/II.cpp File Reference
20.45.1 Detailed Description
20.46ll.cpp
20.47/home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp File Reference
20.47.1 Detailed Description
20.48lockguard.cpp
20.49/home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp File Reference
20.49.1 Detailed Description
20.50 mailbox.cpp
20.51/home/moslevin/projects/m3-repo/kernel/src/message.cpp File Reference
20.51.1 Detailed Description
20.52message.cpp
20.53/home/moslevin/projects/m3-repo/kernel/src/mutex.cpp File Reference
20.53.1 Detailed Description
20.54mutex.cpp
20.55/home/moslevin/projects/m3-repo/kernel/src/notify.cpp File Reference
20.55.1 Detailed Description
20.56notify.cpp
20.57/home/moslevin/projects/m3-repo/kernel/src/profile.cpp File Reference
20.57.1 Detailed Description
20.58profile.cpp
20.59/home/moslevin/projects/m3-repo/kernel/src/public/atomic.h File Reference
20.59.1 Detailed Description
20.60 atomic.h
20.61/home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h File Reference

CONTENTS xliii

20.61.1 Detailed Description
20.62autoalloc.h
20.63/home/moslevin/projects/m3-repo/kernel/src/public/blocking.h File Reference
20.63.1 Detailed Description
20.64blocking.h
20.65/home/moslevin/projects/m3-repo/kernel/src/public/colist.h File Reference
20.65.1 Detailed Description
20.66colist.h
20.67/home/moslevin/projects/m3-repo/kernel/src/public/condvar.h File Reference
20.67.1 Detailed Description
20.68condvar.h
20.69/home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h File Reference
20.69.1 Detailed Description
20.70coroutine.h
20.71/home/moslevin/projects/m3-repo/kernel/src/public/cosched.h File Reference
20.71.1 Detailed Description
20.72cosched.h
20.73/home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h File Reference
20.73.1 Detailed Description
20.74criticalguard.h
20.75/home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h File Reference
20.75.1 Detailed Description
20.76criticalsection.h
20.77/home/moslevin/projects/m3-repo/kernel/src/public/eventflag.h File Reference 50
20.77.1 Detailed Description
20.78eventflag.h
20.79/home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h File Reference 50
20.79.1 Detailed Description
20.80ithreadport.h
20.81/home/moslevin/projects/m3-repo/kernel/src/public/kernel.h File Reference

XIIV CONTENTS

20.81.1 Detailed Description
20.81.2 Typedef Documentation
20.81.2.1 DebugPrintFunction
20.82kernel.h
20.83/home/moslevin/projects/m3-repo/kernel/src/public/kerneldebug.h File Reference 504
20.83.1 Detailed Description
20.83.2 Macro Definition Documentation
20.83.2.1 KERNEL_ASSERT
20.84kerneldebug.h
20.85/home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h File Reference 506
20.85.1 Detailed Description
20.86kernelswi.h
20.87/home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h File Reference 507
20.87.1 Detailed Description
20.88kerneltimer.h
20.89/home/moslevin/projects/m3-repo/kernel/src/public/kerneltypes.h File Reference 508
20.89.1 Detailed Description
20.90kerneltypes.h
20.91/home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h File Reference 509
20.91.1 Detailed Description
20.92ksemaphore.h
20.93/home/moslevin/projects/m3-repo/kernel/src/public/II.h File Reference
20.93.1 Detailed Description
20.94ll.h
20.95/home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h File Reference
20.95.1 Detailed Description
20.96lockguard.h
20.97/home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h File Reference
20.97.1 Detailed Description
20.98mailbox.h

CONTENTS xlv

20.99/home/moslevin/projects/m3-repo/kernel/src/public/manual.h File Reference
20.99.1 Detailed Description
20.10@nanual.h
20.10/home/moslevin/projects/m3-repo/kernel/src/public/mark3.h File Reference
20.101. Detailed Description
20.102hark3.h
20.102home/moslevin/projects/m3-repo/kernel/src/public/mark3cfg.h File Reference
20.103. Detailed Description
20.103.2Macro Definition Documentation
20.103.2.1KERNEL_CONTEXT_SWITCH_CALLOUT
20.103.2.2KERNEL_DEBUG
20.103.2.3KERNEL_EVENT_FLAGS
20.103.2.4KERNEL_EXTENDED_CONTEXT
20.103.2.5KERNEL_NAMED_THREADS
20.103.2.6KERNEL_ROUND_ROBIN
20.103.2.7KERNEL_STACK_CHECK
20.103.2.8KERNEL_THREAD_CREATE_CALLOUT
20.103.2.9KERNEL_THREAD_EXIT_CALLOUT
20.104nark3cfg.h
20.10 home/moslevin/projects/m3-repo/kernel/src/public/message.h File Reference
20.105. Detailed Description
20.105.2 sing Messages, Queues, and the Global Message Pool
20.10@nessage.h
20.107/home/moslevin/projects/m3-repo/kernel/src/public/mutex.h File Reference
20.107. Detailed Description
20.107.2nitializing
20.107. Resource protection example
20.10 <b>8</b> nutex.h
20.10@home/moslevin/projects/m3-repo/kernel/src/public/notify.h File Reference
20.109. Detailed Description

XIVI

20.11 <b>0</b> otify.h
20.111/home/moslevin/projects/m3-repo/kernel/src/public/paniccodes.h File Reference
20.111. Detailed Description
20.111.2Macro Definition Documentation
20.111.2.1PANIC_ACTIVE_COROUTINE_DESCOPED
20.111.2.2PANIC_ACTIVE_EVENTFLAG_DESCOPED
20.111.2.3PANIC_ACTIVE_MAILBOX_DESCOPED
20.111.2.4PANIC_ACTIVE_MUTEX_DESCOPED
20.111.2.5PANIC_ACTIVE_NOTIFY_DESCOPED
20.111.2.6PANIC_ACTIVE_SEMAPHORE_DESCOPED
20.111.2.7PANIC_ACTIVE_TIMER_DESCOPED
20.111.2.8PANIC_ASSERT_FAILED
20.111.2.9PANIC_AUTO_HEAP_EXHAUSTED
20.111.2.1MANIC_LIST_UNLINK_FAILED
20.111.2.1PANIC_NO_READY_THREADS
20.111.2.1PANIC_POWERMAN_EXHAUSTED
20.111.2.1BANIC_RUNNING_THREAD_DESCOPED
20.111.2.1AANIC_STACK_SLACK_VIOLATED
20.11 paniccodes.h
20.112home/moslevin/projects/m3-repo/kernel/src/public/priomap.h File Reference
20.113. Detailed Description
20.11 <b>\peri</b> omap.h
20.11 home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h File Reference
20.115. Detailed Description
20.11 <b>6</b> riomapl1.h
20.117/home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h File Reference
20.117. Detailed Description
20.11 <b>8</b> riomapl2.h
20.11 nome/moslevin/projects/m3-repo/kernel/src/public/profile.h File Reference
20.119. Detailed Description

CONTENTS xlvii

20.12 <b>p</b> rofile.h
20.12/lhome/moslevin/projects/m3-repo/kernel/src/public/profiling_results.h File Reference 536
20.12@profiling_results.h
20.122home/moslevin/projects/m3-repo/kernel/src/public/quantum.h File Reference
20.123. Detailed Description
20.12 <b>4</b> uantum.h
20.125home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h File Reference
20.125. Detailed Description
20.126eaderwriter.h
20.127/home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h File Reference
20.127. Detailed Description
20.127.2/ariable Documentation
20.127.2.1g_pclCurrent
20.127.2.2g_pclNext
20.128cheduler.h
20.12@home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h File Reference 54
20.129. Detailed Description
20.136chedulerguard.h
20.13/home/moslevin/projects/m3-repo/kernel/src/public/sizeprofile.h File Reference
20.138izeprofile.h
20.132 home/moslevin/projects/m3-repo/kernel/src/public/thread.h File Reference
20.133. Detailed Description
20.13 <b>t</b> hread.h
20.13#home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h File Reference
20.135. Detailed Description
20.13 <b>6</b> hreadlist.h
20.137/home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h File Reference
20.137.1Detailed Description
20.138hreadlistlist.h
20.13@nome/moslevin/projects/m3-repo/kernel/src/public/timer.h File Reference

xlviii CONTENTS

20.139. Detailed Description
20.14 <b>6</b> mer.h
20.14/home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h File Reference
20.141. Detailed Description
20.14 <b>2</b> merlist.h
20.142home/moslevin/projects/m3-repo/kernel/src/public/timerscheduler.h File Reference
20.143. Detailed Description
20.14#merscheduler.h
20.145 home/moslevin/projects/m3-repo/kernel/src/quantum.cpp File Reference
20.145. Detailed Description
20.14 <b>q</b> uantum.cpp
20.147/home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp File Reference
20.147. Detailed Description
20.14 <b>&amp;</b> eaderwriter.cpp
20.149home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp File Reference
20.149. Detailed Description
20.149.2/ariable Documentation
20.149.2.1g_pclCurrent
20.149.2.2g_pclNext
20.15@cheduler.cpp
20.15/home/moslevin/projects/m3-repo/kernel/src/thread.cpp File Reference
20.151. Detailed Description
20.15 <b>2</b> hread.cpp
20.152 home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp File Reference
20.153. Detailed Description
20.15threadlist.cpp
20.155 home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp File Reference
20.155. Detailed Description
20.15 <b>6</b> hreadlistlist.cpp
20.157/home/moslevin/projects/m3-repo/kernel/src/timer.cpp File Reference
20.157. Detailed Description
20.158mer.cpp
20.15@home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp File Reference
20.159. Detailed Description
20.16 <b>0</b> merlist.cpp

CONTENTS xlix

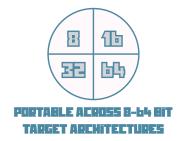
21	Example Documentation	569
	21.1 lab10_notifications/main.cpp	569
	21.2 lab11_mailboxes/main.cpp	570
	21.3 lab1_kernel_setup/main.cpp	572
	21.4 lab2_coroutines/main.cpp	574
	21.5 lab3_round_robin/main.cpp	576
	21.6 lab4_semaphores/main.cpp	578
	21.7 lab5_mutexes/main.cpp	579
	21.8 lab6_timers/main.cpp	581
	21.9 lab7_events/main.cpp	583
	21.10lab8_messages/main.cpp	585
	21.11lab9 dynamic threads/main.cpp	587

# **Chapter 1**

# The Mark3 Realtime Kernel



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







The RTOS is written using a super portable design that scales to many common processor architectures, including a variety of 8, 16, 32, and 64-bit targets. The flexible CMake-based build system facilitates compiling the kernel, tests, examples, and user-application code for any supported target with a consistent interface.

The API is rich and surprisingly simple – with six function calls, you can set up the kernel, initialize two threads, and start the scheduler.

Written in modern C++, Mark3 makes use of modern language features that improve code quality, reduces duplication, and simplifies API usage. C++ isn't your thing? No problem! there's a complete set of C-language bindings available to facilitate the use of Mark3 in a wider variety of environments.

The Mark3 kernel releases contain zero compiler warnings, zero compiler errors, and have zero unit test failures. The build and test process can be automated through the Mark3-docker project, allowing for easy integration with continuous integration environments. The kernel is also run through static analysis tools, automated profiling and documentation tools.

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

# **Chapter 2**

# License

### 2.1 License

Copyright (c) 2012 - 2019, mOslevin 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 names m0slevin, Mark Slevinsky, 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 AND/OR ITS CONTRIBUTORS 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.

4 License

# **Chapter 3**

# **Configuring The Kernel**

#### 3.1 Overview

Configuration is done through setting options in mark3cfg.h , and portcfg.h.

mark3cfg.h contains global kernel configuration options, which determine specific kernel behaviors, and enable certain kernel APIs independent of the any target architecture. Previous to the R7 release, all kernel configuration options were set from mark3cfg.h, and there was an incredible amount of granularity in the configuration options.

One main motivating factor behind removing the granular configuration in mark3cfg.h is that it added a ton of #ifdefs throughout the code, which made it look less clean. It was also difficult to maintain since there were too many permutations and combinations of configuration options to reasonably test.

Another motivation for removing the vast array of configuration options from mark3cfg.h is that there's limited benefit to code size. With the advent of modern compiler optimizations such as section-based garbage collection and link-time optimizations, compilers to a remarkable job of optimizing out unused code. Mark3 supports these optimizations, allowing for nearly the same level of performance benefit as feature specific #ifdefs. In short - you still only pay for what you use, without having to select the features you want ahead of time.

### 3.1.1 Kernel Configuration Options

Kernel configuration is performed by setting #define's in mark3cfg.h to values of 0 or 1.

#### **KERNEL DEBUG**

When enabled, assert statements within the kernel are checked at runtime. This is useful for tracking kernel-breaking changes, memory corruption, etc. in debug builds.

Should be disabled in release builds for performance reasons.

## KERNEL\_STACK\_CHECK

Perform stack-depth checks on threads at each context switch, which is useful in detecting stack overflows / near overflows. Near-overflow detection uses thresholds defined in the target's portcfg.h. Enabling this also adds the Thread::GetStackSlack() method, which allows a thread's stack to be profiled on-demand.

Note: When enabled, the additional stack checks result in a performance hit to context switches and thread initialization.

#### KERNEL\_NAMED\_THREADS

Enabling this provides the Thread::SetName() and Thread::GetName() methods, allowing for each thread to be named with a null-terminated const char\* string.

Note: the string passed to Thread::SetName() must persist for the lifetime of the thread

#### KERNEL EVENT FLAGS

This flag enables the event-flags synchronization object. This feature allows threads to be blocked, waiting on specific condition bits to be set or cleared on an EventFlag object.

While other synchronization objects are enabled by default, this one is configurable because it impacts the Thread object's member data.

### KERNEL\_CONTEXT\_SWITCH\_CALLOUT

When enabled, this feature allows a user to define a callback to be executed whenever a context switch occurs. Enabling this provides a means for a user to track thread statistics, but it does result in additional overhead during a context switch.

#### KERNEL\_THREAD\_CREATE\_CALLOUT

This feature provides a user-defined kernel callback that is executed whenever a thread is started.

#### KERNEL THREAD EXIT CALLOUT

This feature provides a user-defined kernel callback that is executed whenever a thread is terminated.

#### KERNEL ROUND ROBIN

Enable round-robin scheduling within each priority level. When selected, this results in a small performance hit during context switching and in the system tick handler, as a special software timer is used to manage the running thread's quantum. Can be disabled to optimize performance if not required.

#### KERNEL EXTENDED CONTEXT

Provide a special data pointer in the thread object, which may be used to add additional context to a thread. Typically this would be used to implement thread-local-storage.

#### 3.1.2 Port Configuration Options

The bulk of kernel configuration options reside in the target's portcfg.h file. These options determine various sizes, priorities, and default values related to registers, timers, and threads. Some ports may define their own configuration options used locally by its kerneltimer/kernelswi/threadport modules; these are not shown here. The common configuration options are described below.

#### KERNEL\_NUM\_PRIORITIES

Define the number of thread priorities that the kernel's scheduler will support. The number of thread priorities is limited only by the memory of the host CPU, as a ThreadList object is statically-allocated for each thread priority.

In practice, systems rarely need more than 32 priority levels, with the most complex having the capacity for 256.

## KERNEL\_TIMERS\_THREAD\_PRIORITY

Define the priority at which the kernel timer thread runs. Typically, this needs to be one of the highest

3.1 Overview 7

Note: Other threads at or above the timer thread's priority are not permitted to use the kernel's Timer API, as the thread relies on strict priority scheduling to manage the global timer list without requiring additional/excessive critical sections.

#### THREAD\_QUANTUM\_DEFAULT

Number of milliseconds to set as the default epoch for round-robin scheduling when multiple ready threads are active within the same priority.

#### KERNEL\_STACK\_GUARD\_DEFAULT

Set the minimum number of words of margin that each thread's stack must maintain. If a thread's stack grows into their margin, a kernel assert will be generated in debug builds. This is useful for ensuring that threads are not in danger of overflowing their stacks during development/verification.

#### K WORD

Define the size of a data word (in bytes) on the target

#### K ADDR

Define the size of an address (in bytes) on the target. This typically only differs from K\_WORD on Harvard-architecture CPUs.

#### K INT

Define a type to be used as an integer by the kernel.

#### PORT\_PRIO\_TYPE

Set a base datatype used to represent each element of the scheduler's priority bitmap.

#### PORT\_PRIO\_MAP\_WORD\_SIZE

Size of PORT PRIO TYPE in bytes

#### PORT SYSTEM FREQ

Define the running CPU frequency. This may be an integer constant, or an alias for another variable which holds the CPU's current running frequency.

#### PORT\_TIMER\_FREQ

Set the timer tick frequency. This is the frequency at which the fixed-frequency kernel tick interrupt occurs.

### PORT\_KERNEL\_DEFAULT\_STACK\_SIZE

Define the default thread stack size (in bytes)

#### PORT\_KERNEL\_TIMERS\_THREAD\_STACK

Define the Timer thread's stack size (in bytes)

### PORT\_TIMER\_COUNT\_TYPE

Define the native type corresponding to the target timer hardware's counter register.

#### PORT\_MIN\_TIMER\_TICKS

Minimum number of timer ticks for any delay or sleep, required to ensure that a timer cannot be initialized to a negative value.

#### PORT OVERLOAD NEW

Set this to 1 to overload the system's New/Free functions with the kernel's allocator functions. A user must configure the Kernel's allocator functions to point to a real heap implementation backed with real memory in order to use dynamic object creation.

#### PORT STACK GROWS DOWN

Set this to 1 if the stack grows down in the target architecture, or 0 if the stack grows up

#### PORT\_USE\_HW\_CLZ

Set this to 1 if the target CPU/toolchain supports an optimized Count-leading-zeros instruction, or count-leading-zeros intrinsic. If such functionality is not available, a general-purpose implementation will be used.

See also

portcfg.h mark3cfg.h

# **Chapter 4**

# **Building The Kernel**

Mark3 contains its own build system and support scripts, based on CMake and Ninja.

# 4.1 Prerequisites

To build via CMake, a user requires a suitable, supported toolchain (i.e. gcc-avr, arm-none-eabi-gcc), CMake 3.4.2 or higher, and a backend supported by CMake (i.e. Ninja build).

For example, on debian-based distributions, such as Ubuntu, the avr toolchain can be installed using:

```
apt-get install avr-libc gcc-avr cmake ninja-build
```

Once a sane build environment has been created, the kernel, libraries, examples and tests can be built by running  $\min$  from the target's build directory (/out/<target>/). By default, Mark3 builds for the atmega328p target, but the target can be selected by manually configuring environment variables, or by running the ./build/set\_target.sh script as follows:

This script is a thin wrapper for the cmake configuration commands, and clears the output directory before reinitializing cmake for the selected target.

# 4.2 Building

To build the Mark3 kernel and middleware libraries for a generic ARM Cortex-M0 using a pre-configured arm-none-eabi-gcc toolchain, one would run the following commands:

```
./build/set_target.sh cm0 generic gcc ./build/build.sh
```

To perform an incremental build, go into the target's cmake build directory (/out/<target/) and simply run 'ninja'.

To perform a clean build, go into the target's cmake build directory (/out/<target/) and run 'ninja clean', then 'ninja'.

Note that not all libraries/tests/examples will build in all kernel configurations. The default kernel configuration may need adjustment/tweaking to support a specific part. See the documentation for details.

10 Building The Kernel

# 4.3 Supported Targets

Currently, Mark3 supports GCC toolchains for the following targets:

```
atmega328p
atmega644
atmega1284p
atxmega256a3
atmega1280
atmega2560
msp430f2274
ARM Cortex-M0
ARM Cortex-M3 (Note: Also supports Cortex-M4)
ARM Cortex-M4F (Note: Also supports Cortex-M7)
```

## 4.4 More Info

The Mark3 project also has a ready-made docker image to simplify the process. Please see the Mark3 Docker project here for more details:

https://github.com/moslevin/mark3-docker

# **Chapter 5**

# The Mark3 API

## 5.1 Kernel Setup

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

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

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

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

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

This is shown in the example code below:

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

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

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

static K_WORD awAppStack[STACK_SIZE_APP / sizeof(K_WORD)];
static K_WORD awIdleStack[STACK_SIZE_IDLE / sizeof(K_WORD)];

//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.

12 The Mark3 API

```
int main (void)
    //1) Initialize the kernel prior to use
   Kernel::Init();
                               // MUST be before other kernel ops
    //2) Initialize all of the threads we've defined
   AppThread.Init( awAppStack,
                                           // Pointer to the stack
                    awappstack,
sizeof(awAppStack),
                                            // Size of the stack
                                            // Thread priority
                    (void*) AppEntry,
                                            // Entry function
                                            // Entry function argument
                    nullptr );
   IdleThread.Init( awIdleStack,
                                            // Pointer to the stack
                                            // Size of the stack
                     sizeof(awIdleStack),
                     Ο,
                                            // Thread priority
                     (void*)IdleEntry,
                                            // Entry function
                    nullptr );
                                            // Entry function argument
    //3) Add the threads to the scheduler
                               // Actively schedule the threads
   AppThread.Start():
    IdleThread.Start();
   //4) Give control of the system to the kernel
   Kernel::Start();
                                // Start the kernel!
```

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

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

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

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

### 5.2 Threads

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

#### 5.2.1 Thread Setup

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

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

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

5.2 Threads

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.

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.

#### 5.2.2 Entry Functions

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

The most basic Thread loop is shown below:

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

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

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

14 The Mark3 API

#### 5.3 Timers

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

While extremely simple to use, they provide one of the most powerful execution contexts in the system. The timer callbacks execute from within a timer thread, as a result of a semaphore posted in a timer interrupt. Timer callbacks are executed from a high-priority thread – typically at the highest priority thread in the system. 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 lower-priroity threads by way of semaphores or messages.

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

# 5.4 Semaphores

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

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

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

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

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

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

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

5.5 Mutexes 15

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

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

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

#### 5.5 Mutexes

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

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

Prioritiy 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();
    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex
    my_protected_resource.do_something();
    my_protected_resource.do_something_else();
    clMutex.Release();
}
// Some function called from another thread
void Thread2Function()
    clMutex.Claim();
    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex
    my_protected_resource.do_something();
    my_protected_resource.do_different_things();
    clMutex.Release();
```

16 The Mark3 API

## 5.6 Event Flags

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

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

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

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

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

Examples demonstrating the use of event flags are shown below.

```
// Simple example showing a thread blocking on a multiple bits in the
// fields within an event flag.
EventFlag clEventFlag;
int main()
    clEventFlag.Init(); // Initialize event flag prior to use
void MyInterrupt()
    // Some interrupt corresponds to event 0x0020
    clEventFlag.Set(0x0020);
void MyThreadFunc()
    while(1) {
        uint16_t u16WakeCondition;
        // Allow this thread to block on multiple flags
        u16WakeCondition = clEventFlag.Wait(0x00FF, EventFlagOperation::Any_Set);
        // Clear the event condition that caused the thread to wake (in this case,
        // u16WakeCondtion will equal 0x20 when triggered from the interrupt above)
        clEventFlag.Clear(u16WakeCondition);
        // <do something>
```

### 5.7 Messages

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

Sending a message consists of the following operations:

- Obtain a Message object from the global message pool
- · Set the message data and event fields

5.7 Messages 17

Send the message to the destination message queue

While receiving a message consists of the following steps:

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

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

#### 5.7.1 Message Objects

Message objects are used to communicate arbitrary data between threads in a safe and synchronous way.

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

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

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

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

#### 5.7.2 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.

18 The Mark3 API

### 5.7.3 Messaging Example

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

### 5.8 Mailboxes

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

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

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

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

All send/receive APIs support an optional timeout parameter if the KERNEL\_USE\_TIMEOUTS option has been configured in mark3cfg.h

#### 5.8.1 Mailbox Example

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

```
// Initialize our mailbox, telling it to use our defined buffer for envelope
    \ensuremath{//} storage. Pass in the size of the buffer, and set the size of each
    // envelope to 16 bytes. This gives ul6 a mailbox capacity of (128 / 16) = 8
    // envelopes.
    clMbox.Init((void*)aucMBoxBuffer, 128, 16);
void SendThread(void)
    // Define a buffer that we'll eventually send to the
    // mailbox. Note the size is the same as that of an // envelope.
    uint8_t aucTxBuf[16];
    while(1) {
         // Copy some data into aucTxBuf, a 16-byte buffer, the // same size as a mailbox envelope.
         // Deliver the envelope (our buffer) into the mailbox
         clMbox.Send((void*)aucTxBuf);
}
void RecvThred(void)
    uint8_t aucRxBuf[16];
    while(1) {
         // Wait until there's a message in our mailbox. Once
         // there is a message, read it into our local buffer.
         cmMbox.Receive((void*)aucRxBuf);
         // Do something with the contents of aucRxBuf, which now // contains an envelope of data read from the mailbox.
```

# 5.9 Notification Objects

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

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

#### 5.9.1 Notification Example

20 The Mark3 API

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

#### 5.10 Condition Variables

Condition Variables, implemented in Mark3 with the ConditionVariable class, provide an implementation of the classic Monitor pattern. This object allows a thread to wait for a specific condition to occur, claiming a shared lock once the condition is met. Threads may also choose to signal a single blocking thread to indicate a condition has changed, or broadcast condition changes to all waiting threads.

### 5.10.1 Condition Variable Example

```
// Define a condition variable object, a shared lock, and
// a piece of common data shared between threads to represent
// a condition.
// Assume Mutex and ConditionVariable are initialized
// prior to use.
static ConditionVariable clCondVar;
static Mutex clSharedLock;
static volatile int iCondition = 0;
void CondThread1(void *unused_)
    while (1)
        // Wait until
        clCondVar.Wait(&clSharedLock);
        // Only act on a specific condition
        if (iCondition == 1337) {
            // Do something
        clSharedLock.Release();
}
void CondThread2(void *unused_)
    // Assume Mutex and ConditionVariable are initialized
    // prior to use.
    while (1)
        // Wait until
        clCondVar.Wait(&clSharedLock);
        // Act on a \stardifferent\star condition than the other thread
        if (iCondition == 5454) {
            // Do something
        clSharedLock.Release();
void SignalThread(void* unused)
    while (1) {
         // Sleep for a while
        Thread::Sleep(100);
        // Update the condition in a thread-safe manner
        clSharedLock.Claim();
        iCondition = 1337;
```

5.11 Reader-Write Locks 21

```
clSharedLock.Release();

// Wake one thread to check for the updated condition
clCondVar.Signal();

// Sleep for a while
Thread::Sleep(100);

// Update the condition in a thread-safe manner
clSharedLock.Claim();
iCondition = 1337;
clSharedLock.Release();

// Wake all threads to check for the updated condition
clCondVar.Broadcast();
}
```

#### 5.11 Reader-Write Locks

Reader-Writer locks are provided in Mark3 to provide an efficient way for multiple threads to share concurrent, non-destructive access to a resource, while preventing concurrent destructive/non-destructive accesses. A single "writer" may hold the lock, or 1-or-more "readers" may hold the lock. In the case that readers hold the lock, writers will block until all readers have relinquished their access to the resource. In the case that a writer holds the lock, all other readers and writers must wait until the lock is relinquished.

#### 5.11.1 Reader-Write Lock Example

```
void WriterTask(void* param)
{
    auto pclRWLock = static_cast<ReaderWriterLock*>(param);

    pclRWLock->AcquireWriter();
    // All other readers and writers will have to wait until
    // the lock is released.
    iNumWrites++;
    ...
    pclRWLock->ReleaseWriter();
}

void ReaderTask(void* param)
{
    auto pclRWLock = static_cast<ReaderWriterLock*>(param);
    pclRWLock->AcquireReader();
    // Any number of reader threads can also acquire the lock
    // without having to block, waiting for this task to release it.
    // Writers must block until all readers have released their references
    // to the lock.
    iNumReads++;
    ...
    pclRWLock->ReleaseReader();
}
```

# **5.12** 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.

22 The Mark3 API

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

# 5.13 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.

# 5.14 Coroutines

Mark3 implements a coroutine scheduler, capable of managing a set of prioritized run-to-completion tasks. This is a simple and lightweight cooperative scheduling mechanism, that trades the preemption and synchonization capabilities of threads for simplicity. It is an ideal mechanism to use for background processes in a system, or for performing a coordinating a group of tasks where the relative priority of task execution is important, but the duration of individual tasks is less important.

Like the Mark3 thread scheduler, the coroutine scheduler supports multiple priorities of tasks. Multiple coroutines activated at the same priority level are executed in first-in first-out order.

Coroutines are activated by interrupts, threads, or from within other co-routines. Once activated, the co-routine is able to be scheuduled.

The coroutine scheduler is called by the application from a thread priority. So long as there are activated tasks to be scheduled, the scheduler will return a pointer to the highest priority active coroutine to be run.

Running a co-routine de-activates the co-routine, meaning that coroutines must be re-activated every time they are run.

```
Coroutine cr1;
Coroutine cr2;
Coroutine cr3;

void CoRoutineInit()
{
    CoScheduler::Init();
    cr1.Init(<priority>, < handler function>, <data passed to handler function>);
    cr2.Init(<priority>, <handler function>, <data passed to handler function>);
    cr3.Init(<priority>, <handler function>, <data passed to handler function>);
}

void AsyncEvent1()
{
    // Some event occurred requiring us to run cr1. Activating a task does not
    // cause the coroutine to be run immediately, but schedules it to be returned by
```

5.15 Critical Guards 23

```
// CoScheduler::Schedule() when there are no higher-priroity active tasks to
    cr1.Activate();
}
void AsvncEvent2()
    // Some event occurred requiring us to run cr2.
    cr2.Activate();
void AsvncEvent3()
    // Some event occurred requiring us to run cr3.
    cr3.Activate();
void CoRoutineSchedulerTask()
    while (1) {
        auto* coroutineToRun = CoScheduler::Schedule();
        if (coroutineToRun) {
            // Scheduled task is removed from the scheduler once run. It must
            // be reactivated to be run again
            coroutineToRun->Run();
        // If run from idle thread, could go into sleep mode instead of running in
        // a tight loop.
```

#### 5.15 Critical Guards

Often times, it is useful in a real-time multi-threaded system to place a critical section around a block of code to protect it against concurrent access, or to protect global data from access from interrupts. In Mark3 there are a few different ways of implementing critical sections. Historically, the CS\_ENTER() and CS\_EXIT() macros were used for this purpose; however, the usage of matching entry/exit macros can often-times be cumbersome and error-prone.

The CriticalGuard object allows a user to wrap a block of code in a critical section, where the critical section is entered when the critical guard object is declared, and the critical section is exited when the object goes out-of-scope.

It is essentially an RAII-style critical section object, that provides the benefit of critical sections without the hassle of having to carefully match enter/exit statements.

```
void MyFunc()
{
    // operations outside of critical section
    {
        auto cg = CriticalGuard{};
        // operations protected by critical section
        // critical section ends when CriticalGuard object goes out of scope
    }
    // Operations outside of critical section
}
```

#### 5.16 Lock Guards

The modern C++ standard library provides RAII-style mutex locking. Unfortunately such an implementation is not usable in the context of Mark3, because the STL implementation supplied for an embedded C++ toolchain would not be aware of Mark3's threading model or synchronization primatives. Mark3 provides its own RAII mutex locking mechanism in the form of LockGuard objects. When a LockGuard object is declared (referencing a valid and initialized mutex object at construction), the lock is claimed upon declaration, and released when the object goes out-of-scope.

24 The Mark3 API

```
Mutex clMutex;
void MyFunc()
{
    // operations outside of mutex-locked context
    {
        auto lg = LockGuard{&clMutex};
        // operations inside mutex-locked context
        // mutex automatically unlocked when LockGuard goes out of scope
    }
    // Operations outside of mutex-locked context
}
```

# 5.17 Scheduler Guards

Similar to the LockGuard and CriticalGuard objects, the SchedulerGuard object provides a coped scheduler-disabled blocks. This essentially gives the executing thread exclusive control of the CPU - except for interrupts - for the duration of the block wrapped in the SchedulerGuard. The scheduler is disabled when the object is declared, and scheduler state is restored when the SchedulerGuard object goes out-of-scope. This is yet another form of RAII-based resource locking in Mark3.

```
void MyFunc()
{
    // thread scheduler enabled
    {
        auto sg = SchedulerGuard{};
        // thread-safety guaranteed - scheduler disabled
        // thread scheduler re-enabled when SchedulerGuard object goes out of scope
    }
    // thread scheduler enabled
```

# **Chapter 6**

# Why Mark3?

My first job after graduating from university in 2005 was with a small company that had a very old-school, low-budget philosophy when it came to software development. Every make-or-buy decision ended with "make" when it came to tools. It was the kind of environment where vendors cost us money, but manpower was free. In retrospect, we didn't have a ton of business during the time that I worked there, and that may have had something to do with the fact that we were constantly short on ready cash for things we could code ourselves.

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

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

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

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

26 Why Mark3?

tools worked gave me a better understanding of how to use them - making me more productive at the jobs that I've had since.

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

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

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

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

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

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

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

# **Chapter 7**

# When to use an RTOS?

# 7.1 The reality of system code

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

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

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

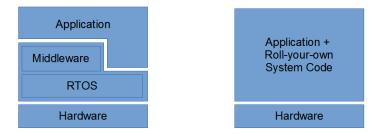


Figure 7.1 Arch

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

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

28 When to use an RTOS?

Conversely, an RTOS-based system incurs a slightly higher fixed cost up-front, but scales infinitely better than a bare-metal system as application's complexity increases. Using an RTOS for simple systems reduces application development time, but may cause an application not to fit into some extremely size-constrained microcontroller. An RTOS can also cause the size of an application to grow more slowly relative to a bare-metal system – especially as a result of applying synchronization mechanisms and judicious IPC. As a result, an RTOS makes it significantly easier to "go agile" with an application – iteratively adding features and functionality, without having to consider refactoring the underlying system at each turn.

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

Consider the following questions when making that decision:

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

# 7.2 Superloops, and their limitations

# 7.2.1 Intro to Superloops

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

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

Nonetheless, a variety of examples are presented here- from the extremely simple, to cooperative and 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.

#### 7.2.2 The simplest loop

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

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

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

Despite its simplicity we can see the beginnings of some core OS concepts. Here, the while(1) statement can be logically seen as the 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 sequence in the main while-loop. Note that in this example the CPU is always busy running tasks - at no time is the CPU idle, meaning that it is likely burning a lot of power.

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

#### 7.2.3 Interrupt-Driven Super-loop

In the previous example, we had a system without any way to control the execution of the task- it just runs forever. There's no way to control when the task can (or more importantly can't) run, which greatly limits the usefulness of the system. Say you only want your task to run every 100 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:

30 When to use an RTOS?

```
volatile K_BOOL something_to_do = false;
__interrupt__ My_Interrupt_Source(void)
{
    something_to_do = true;
}
int main()
{
    while (1)
    {
        if (something_to_do)
        {
            Do_something();
            something_to_do = false;
        }
        else
        {
            Idle();
        }
}
```

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

The first thing to notice in the source is that the global variable, something\_to\_do, is used as a synchronization object. When an interrupt occurs from some external event, triggering the My\_Interrupt\_Source() ISR, program flow in main() is interrupted, the interrupt handler is run, and something\_to\_do is set to true, letting us know that when we get back to main(), that we should run our Do\_something() task.

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

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

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

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

# 7.2.4 Cooperative multi-tasking

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

```
// Bitfield values used to represent three distinct tasks
#define TASK_1_EVENT (0x01)
#define TASK_2_EVENT (0x02)
#define TASK_3_EVENT (0x04)
volatile K_UCHAR event_flags = 0;
// Interrupt sources used to trigger event execution
 _interrupt__ My_Interrupt_1(void)
    event flags |= TASK 1 EVENT;
 _interrupt__ My_Interrupt_2(void)
    event_flags |= TASK_2_EVENT;
 _interrupt__ My_Interrupt_3(void)
    event_flags |= TASK_3_EVENT;
// Main tasks
int main(void)
    while(1)
       while (event_flags)
            if( event_flags & TASK_1_EVENT)
                Do_Task_1();
                event_flags &= ~TASK_1_EVENT;
            } else if( event_flags & TASK_2_EVENT) {
                Do Task 2();
                event_flags &= ~TASK_2_EVENT;
            } else if( event_flags & TASK_3_EVENT) {
                Do_Task_3();
                event_flags &= ~TASK_3_EVENT;
        Idle();
```

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

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

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

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

32 When to use an RTOS?

# 7.2.5 Hybrid cooperative/preemptive multi-tasking

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

```
// Bitfields used to represent high-priority tasks. Tasks in this group
// can preempt tasks in the group below - but not eachother.
#define HP_TASK_1(0x01)
#define HP_TASK_2(0x02)
volatile K_UCHAR hp_tasks = 0;
// Bitfields used to represent low-priority tasks.
#define LP_TASK_1(0x01)
#define LP_TASK_2(0x02)
volatile K_UCHAR lp_tasks = 0;
// Interrupt sources, used to trigger both high and low priority tasks.
__interrupt__ System_Interrupt_1(void)
    // Set any of the other tasks from here...
hp_tasks |= HP_TASK_1;
// Trigger the SWI that calls the High_Priority_Tasks interrupt handler
  _interrupt__ System_Interrupt_n...(void)
// Set any of the other tasks from here...
// Interrupt handler that is used to implement the high-priority event context
 _interrupt__ High_Priority_Tasks(void)
    // Enabled every interrupt except this one
    Disable_My_Interrupt();
    Enable_Interrupts();
    while( hp_tasks)
        if( hp_tasks & HP_TASK_1)
        {
            HP_Task1();
            hp_tasks &= ~HP_TASK_1;
        else if (hp_tasks & HP_TASK_2)
            HP_Task2();
            hp_tasks &= ~HP_TASK_2;
    Restore_Interrupts();
    Enable_My_Interrupt();
// Main loop, used to implement the low-priority events
int main(void)
    // Set the function to run when a SWI is triggered
    Set_SWI(High_Priority_Tasks);
    // Run our super-loop
    while(1)
        while (lp_tasks)
             if (lp_tasks & LP_TASK_1)
                 LP_Task1();
                 lp_tasks &= ~LP_TASK_1;
            else if (lp_tasks & LP_TASK_2)
                 LP_Task2();
                 lp_tasks &= ~LP_TASK_2;
        Idle();
}
```

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

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

# 7.3 Problems with superloops

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

#### 7.3.1 Hidden Costs

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

# 7.3.2 Tightly-coupled code

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

## 7.3.3 No blocking Calls

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

# 7.3.4 Difficult to guarantee responsiveness

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

When to use an RTOS?

# 7.3.5 Limited preemption capability

As shown in the example code, the way to gain preemption in a superloop is through the use of nested interrupts. While this isn't 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 8**

# Can you afford an RTOS?

#### 8.1 Intro

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

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

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

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

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

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

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

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

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

RTOS overhead can be broken into three distinct areas:

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

While there are other notable reasons to include or avoid the use of an RTOS in certain applications (determinism, responsiveness, and interrupt latency among others), these are not considered in this discussion - as they are difficult to consider for the scope of our "canned" application.

# 8.2 Application description

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

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

Code Space and Memory Overhead:

The application is compiled for an ATMega328p processor which contains 32kB of code space in flash, and 2kB of RAM, which is a lower-mid-range microcontroller in Atmel's 8-bit AVR line of microcontrollers. Using the AVR GCC compiler with -Os level optimizations, an executable is produced with the following code/RAM utilization:

Program: 27914 bytes Data: 1313 bytes

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

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

Program: 24886 bytes Data: 750 bytes

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

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

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

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

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

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

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

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

Fixed kernel data costs:

8.3 Runtime Overhead 37

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

#### Application (Variable) data costs:

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

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

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

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

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

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

#### 8.3 Runtime Overhead

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

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

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

Preemptive mode:

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

Cooperative mode:

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

coop - 438 cycles preempt - 764 cycles

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

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

# 8.4 Analysis

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

On the one hand, if these tests had been performed on a higher-end microcontroller such as the ATMega1284p (containing 128kB of code space and 16kB of RAM), the overhead would be in the noise. For this type of resource-rich microcontroller, there would be no reason to avoid using the Mark3 preemptive kernel.

Conversely, using a lower-end microcontroller like an ATMega88pa (which has only 8kB of code space and 1kB of RAM), the added overhead would likely be prohibitive for including a preemptive kernel. In this case, the cooperative-mode kernel would be a better choice.

As a rule of thumb, if one budgets 25% of a microcontroller's code space/RAM for system code, you should only require at minimum a microcontroller with 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 9**

# **Mark3 Design Goals**

#### 9.1 Overview

#### 9.1.1 Services Provided by an RTOS Kernel

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

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

## 9.1.2 Guiding Principles of Mark3

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

#### 9.1.3 Be feature competitive

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

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

## 9.1.4 No external dependencies, no non-language features

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

40 Mark3 Design Goals

# 9.1.5 Target the most popular hobbyist platforms available

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

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

#### 9.1.6 Maximize determinism – but be pragmatic

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

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

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

#### 9.1.7 Apply engineering principles – and that means discipline, measurement and verification

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

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

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

9.1 Overview 41

#### 9.1.8 Use Virtualization For Verification

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

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

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

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

- · On linux, it requires only pthreads.
- On Windows, it rquires pthreads and ws2\_32, both satisfied via MinGW.
- Optional SDL builds for both targets (featuring graphics and simulated joystick input) can be built, and rely on libSDL.

42 Mark3 Design Goals

# **Chapter 10**

# **Mark3 Kernel Architecture**

#### 10.1 Overview

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

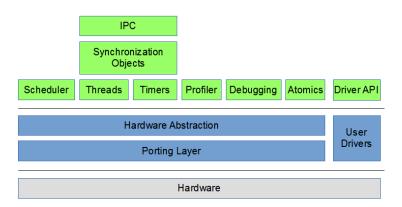


Figure 10.1 Overview

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

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

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

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

**IPC:** Inter-process-communications. Message-passing and Mailbox interfaces used to communicate between threads synchronously or asynchronously.

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

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

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

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

**Atomics:** Support for UN-interruptble arithmatic operations.

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

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

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

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

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

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

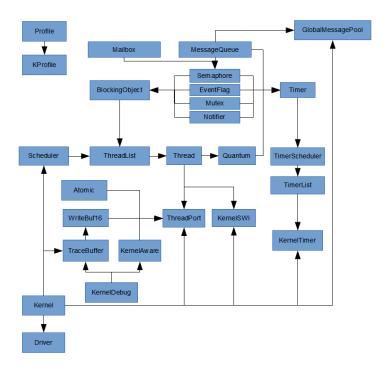


Figure 10.2 Overview

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

## **Atomic Operations**

Atomic - atomic.cpp/.h

#### **Memory Allocators**

AutoAlloc - autoalloc.cpp/.h

10.1 Overview 45

#### **CoRoutines**

CoScheduler - cosched.cpp/.h CoList - colist.cpp/.h Coroutine - coroutine.cpp/.h

#### **Data Structures**

LinkList - II.cpp/.h PriorityMapL1 - priomapl1.h PriorityMapL2 - priomapl2.h

## Threads + Scheduling

Thread - thread.cpp/.h Scheduler - scheduler.cpp/.h Quantum - quantum.cpp/.h ThreadPort - threadport.cpp/.h \*\* KernelSWI - kernelswi.cpp/.h \*\* ThreadList - threadlist.cpp/.h ThreadListList - threadlistlist.cpp/.h

#### **Profiling**

ProfileTimer - profile.cpp/.h

#### **Timers**

Timer - timer.h/timer.cpp
TimerScheduler - timerscheduler.h
TimerList - timerlist.h/cpp
KernelTimer - kerneltimer.cpp/.h \*\*

# **Synchronization**

BlockingObject - blocking.cpp/.h
Semaphore - ksemaphore.cpp/.h
EventFlag - eventflag.cpp/.h
Mutex - mutex.cpp/.h
Notify - notify.cpp/.h
ConditionVariable - condvar.cpp/.h
ReaderWriterLock - readerwriter.cpp/.h
CriticalGuard - criticalguard.h
CriticalSection - criticalsection.h
LockGuard - lockguard.h
SchedGuard - schedguard.h

#### IPC/Message-passing

Mailbox - mailbox.cpp/.h MessageQueue - message.cpp/.h

#### Debugging

Miscellaneous Macros - kerneldebug.h

#### Kernel

Kernel - kernel.cpp/.h

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

# 10.2 Threads and Scheduling

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

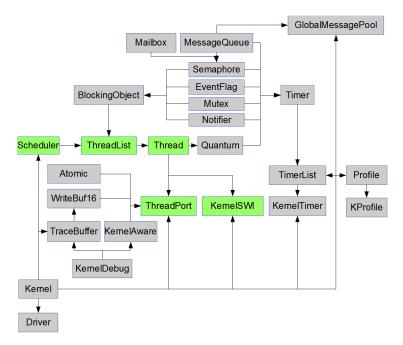


Figure 10.3 Threads and Scheduling

#### 10.2.1 A Bit About Threads

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

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

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

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

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

#### 10.2.2 Thread States and ThreadLists

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

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

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

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

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

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

#### 10.2.3 Blocking and Unblocking

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

#### 10.2.4 Blocking Objects

Blocking objects and primatives provided by Mark3 include:

- · Semaphores (binary and counting)
- Mutexes
- · Event Flags
- · Thread Notification Objects
- · Thread Sleep
- · Message Queues

· Mailboxes

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

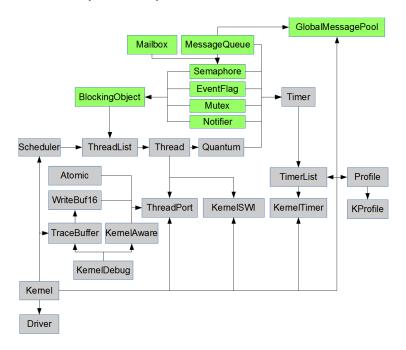


Figure 10.4 Blocking Objects

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

## 10.3 Inside the Mark3 Scheduler

At this point we've covered the following concepts:

- · Threads
- · Thread States and Thread Lists
- · Blocking and Un-Blocking Threads

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

```
Find the highest-priority "Ready" list that has at least one Thread. Select the next thread to run as the first thread in that list
```

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

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

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

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

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

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

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

Now, to implement the leading-zeros check, this can once again be performed iteratively using bitshifts and compares (which isn't any more efficient than the raw list traversal), but it can also be evaluated using either a lookup table, or via a special CPU instruction to count the leading zeros in a value. In Mark3, we use all approaches. In the event a target architecture or toolchain has intrinsic support for a count-leading-zeroes (PORT\_CLZ) instruction, that implementation is used. Otherwise, a software-based implementation is provided – either using a lookup table, or a bitshift-and-compare algorithm.

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

For the lookup-table approach - a 4-bit lookup table can be used with an 8-bit priority-level bitmap would look something like this:

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

// priority is non-zero if we found something there
if( priority )
{
    // Add 4 because we were looking at the higher levels
    priority += 4;
}
else
{
    // Nothing in the upper 4, look at the lowest 4 priority levels
    // represented by the lowest 4 bits in the bitmap
    priority = priority_lookup_table[priority_bitmap & 0x0F];
}
```

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

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

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

next\_thread = thread\_list[prio].get\_head();

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

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

#### 10.3.1 Considerations for Round-Robin Scheduling

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

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

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

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

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

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

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

10.4 Timers 51

# 10.3.2 Context Switching

There's really not much to say about the actual context switch operation at a high level. Context switches are triggered whenever it has been determined that a new thread needs to be swapped into the CPU core when the scheduler is run. Mark3 implements also context switches as a call to a software interrupt - on AVR platforms, we typically use INT0 or INT2 for this (although any pin-change GPIO interrupt can be used), and on ARM we achieve this by triggering a PendSV exception.

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

Save the current Thread's context to the current Thread stack Make the "next to run" thread the "currently running" thread Restore the context of the next Thread from the Thread stack

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

# 10.3.3 Putting It All Together

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

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

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

# 10.4 Timers

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

Timers are implemented using the following components in the Mark3 Kernel:

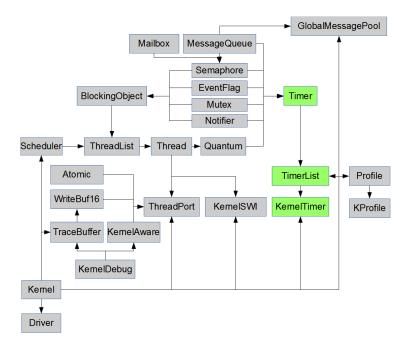


Figure 10.5 Timers

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

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

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

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

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

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

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

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

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

10.4 Timers 53

# 10.4.1 Tick-based Timers

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

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

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

#### 10.4.2 Tickless Timers

Note: Tickless timers are removed as of the R7 release. The below documentation is preserved for historical information only. The reason for removing tickless timers include the overhead associated with managing those timers (significantly more math and management is involved). In practice, there are few scenarios where purely tickless timers add benefit - beyond the most constrained devices. Also, it is entirely possible to disable software timers from the idle task when lower power/fewer interrupts are desired in most cases where a tickless timer would be of value.

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

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

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

# 10.4.3 Timer Processing Algorithm

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

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

In the first pass, the active timers are decremented by either 1 tick (tick-based), or by the duration of the last elapsed timer interval (tickless). Timers that have zero (or less-than-zero) time remaining have a "callback" flag set, telling

the algorithm to call the timer's callback function in the second pass of the loop. In the event of a periodic timer, the timer's interval is reset to its starting value.

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

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

In a tickless system, once the second pass of the loop has been completed, the hardware timer is checked to see if the next timer interval has expired while processing the expired timer callbacks. In that event, the complete algorithm is re-run to ensure that no expired timers are missed. Once the algorithm has completed without the next timer expiring during processing, the expiry time is programmed into the hardware timer. Round-robin scheduling is re-enabled, and if a new thread has been scheduled as a result of action taken during a timer callback, a context switch takes place on return from the timer interrupt.

# 10.5 Synchronization and IPC

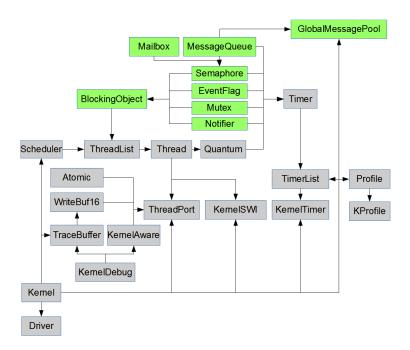


Figure 10.6 Synchronization and IPC

# 10.6 Blocking Objects

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

10.6 Blocking Objects 55

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.

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

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

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

## 10.6.1 Semaphores

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

#### 10.6.2 Mutex

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

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

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

Prioritiy inheritence is provided with these objects as a means to avoid prioritiy 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.

# 10.6.3 Event Flags

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

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

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

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

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

## 10.6.4 Notification Objects

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

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

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

# 10.7 Messages and Message Queues

## 10.7.1 Messages

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

Sending a message consists of the following operations:

- · Obtain a Message object from a source 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 source message pool

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

# 10.7.2 Message Objects

Message objects are used to communicate arbitrary data between threads in a safe and synchronous way.

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

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

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

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

# 10.7.3 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.

#### 10.7.4 Mailboxes

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

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

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

All send/receive APIs support an optional timeout parameter if the KERNEL\_USE\_TIMEOUTS option has been configured in mark3cfg.h

# 10.7.5 Atomic Operations

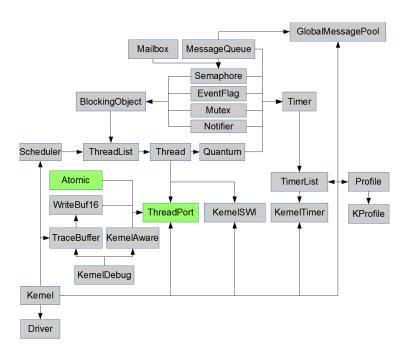


Figure 10.7 Atomic operations

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

### 10.7.6 Drivers

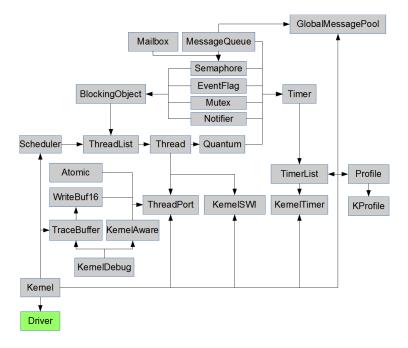


Figure 10.8 Drivers

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

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

### **Driver Design**

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

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

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

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

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

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

### **Driver API**

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

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

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

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

Once a driver has been added to the table, drivers are opened by NAME using DriverList::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!");
```

# 10.8 Kernel Proper and Porting

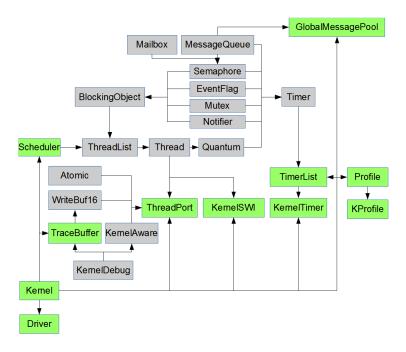


Figure 10.9 Kernel Proper and Porting

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

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

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

#### **Harware Abstraction Layer**

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

These interfaces generally cover four features:

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

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

# Real-world Porting Example - Cortex M0

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

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

/embedded/kernel/CPU/VARIANT/TOOLCHAIN/,

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

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

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

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

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

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

#### Thread Stack Initialization

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

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

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

### a) Exception Stack Frame

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

```
[ XPSR ] <-- Highest address in context
[ PC     ]
[ LR     ]
[ R12     ]
[ R3     ]
[ R2     ]
[ R1     ]
[ R0     ]</pre>
```

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

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

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

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

b) Complimentary CPU Register Context

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

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

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

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

```
void ThreadPort::InitStack(Thread *pclThread_)
    K_ULONG *pulStack;
    K_ULONG *pulTemp;
    K ULONG ulAddr:
    K USHORT i:
    // Get the entrypoint for the thread
    ulAddr = (K_ULONG) (pclThread_->m_pfEntryPoint);
    // Get the top-of-stack pointer for the thread
    pulStack = (K_ULONG*)pclThread_->m_pwStackTop;
    // Initialize the stack to all FF's to aid in stack depth checking
    pulTemp = (K_ULONG*)pclThread_->m_pwStack;
    for (i = 0; i < pclThread_->m_usStackSize / sizeof(K_ULONG); i++)
       pulTemp[i] = 0xFFFFFFF;
    PORT_PUSH_TO_STACK(pulStack, 0);
                                                 // Apply one word of padding
    //-- Simulated Exception Stack Frame --
                                                 // XSPR; set "T" bit for thumb-mode
    PORT_PUSH_TO_STACK(pulStack, 0x01000000);
    PORT_PUSH_TO_STACK(pulStack, ulAddr);
    PORT_PUSH_TO_STACK(pulStack, 0);
    PORT_PUSH_TO_STACK(pulStack, 0x12);
    PORT_PUSH_TO_STACK(pulStack, 0x3);
    PORT_PUSH_TO_STACK(pulStack, 0x2);
    PORT_PUSH_TO_STACK(pulStack, 0x1);
    PORT_PUSH_TO_STACK(pulStack, (K_ULONG)pclThread_->m_pvArg);
                                                                   // R0 = argument
    //-- Simulated Manually-Stacked Registers --
    PORT_PUSH_TO_STACK(pulStack, 0x11);
    PORT_PUSH_TO_STACK(pulStack, 0x10);
    PORT_PUSH_TO_STACK(pulStack, 0x09);
    PORT_PUSH_TO_STACK(pulStack, 0x08);
    PORT_PUSH_TO_STACK(pulStack, 0x07);
    PORT_PUSH_TO_STACK(pulStack, 0x06);
    PORT_PUSH_TO_STACK(pulStack, 0x05);
    PORT_PUSH_TO_STACK(pulStack, 0x04);
    pulStack++;
    pclThread_->m_pwStackTop = pulStack;
```

### **Kernel Startup**

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

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

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

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

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

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

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

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

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

#### **First Thread Entry**

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

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

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

```
get thread stack:
    ; Get the stack pointer for the current thread
    ldr r0, g_pstCurrent
    ldr r1, [r0]
    add r1, #8
   ldr r2, [r1]
                         : r2 contains the current stack-top
load_manually_placed_context_r11_r8:
   ; Handle the bottom 32-bytes of the stack frame
    ; Start with r11-r8, because only r0-r7 can be used
    ; with ldmia on CMO.
    add r2. #16
    ldmia r2!, {r4-r7}
    mov r11, r7
    mov r10, r6
    mov r9, r5
   mov r8, r4
set psp:
   ; Since r2 is coincidentally back to where the stack pointer should be,
    ; Set the program stack pointer such that returning from the exception handler
load_manually_placed_context_r7_r4:
    ; Get back to the bottom of the manually stacked registers and pop.
    sub r2, #32
    ldmia r2!, {r4-r7} ; Register r4-r11 are restored.
set_thread_and_privilege_modes:
    ; Also modify the control register to force use of thread mode as well
    ; For CM3 forward-compatibility, also set user mode.
    mrs r0, control
   mov r1, #0x03
    orr r0, r1
    control, r0
    ; Set up the link register such that on return, the code operates
    ; in thread mode using the PSP. To do this, we or 0x0D to the value stored
    ; in the lr by the exception hardware EXC_RETURN. Alternately, we could
    ; just force lr to be 0xFFFFFFFD (we know that's what we want from the
    ; hardware, anyway)
   mov r0, #0x0D
mov r1, lr
    orr r0, r1
   ; Return from the exception handler.
    ; The CPU will automagically unstack R0-R3, R12, PC, LR, and xPSR \,
              If all goes well, our thread will start execution at the
    : for us.
    ; entrypoint, with the us-specified argument.
    bx r0
```

66 Mark3 Kernel Architecture

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

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

#### 1) Saving the context.

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

### 2) Swap threads

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

#### 3) Restore Context

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

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

```
void PendSV_Handler(void)
     // Thread_SaveContext()
" ldr r1, CURR_ \n"
" ldr r1, [r1] \n "
" mov r3, r1 \n "
" add r3, #8 \n "
     // Grab the psp and adjust it by 32 based on extra registers we're going
     // to be manually stacking. " mrs r2, psp \n " sub r2, #32 \n "
     // While we're here, store the new top-of-stack value
     " str r2, [r3] \n
     // And, while r2 is at the bottom of the stack frame, stack r7-r4
     " stmia r2!, {r4-r7} n "
     // Stack r11-r8
     " mov r7, r11 \n "
     " mov r6, r10 \n "
     " mov r5, r9 \n "
     " mov r4, r8 \n "
     " stmia r2!, {r4-r7} \n "
     // Equivalent of Thread_Swap() - performs g_pstCurrent = g_pstNext
     " ldr r1, CURR_ \n"
" ldr r0, NEXT_ \n"
" ldr r0, [r0] \n"
" str r0, [r1] \n"
     // Thread_RestoreContext()
     // Get the pointer to the next thread's stack
     " add r0, #8 \n "
" ldr r2, [r0] \n "
     // Stack pointer is in r2, start loading registers from // the "manually-stacked" set \ensuremath{\mbox{\sc holimstate}}
     // Start with r\bar{1}1-r8, since these can't be accessed directly.
     " add r2, #16 \n "
     " ldmia r2!, {r4-r7} \n "
" mov r11, r7 \n "
" mov r10, r6 \n "
     " mov r9, r5 \n "
     " mov r8, r4 \n "
```

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

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

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

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

### **Kernel Timers**

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

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

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

```
void KernelTimer::Start(void)
    SysTick Config(PORT SYSTEM FREO / 1000); // 1KHz fixed clock...
    NVIC_EnableIRQ(SysTick_IRQn);
In this instance, the call to SysTick_Config() generates a 1kHz system-tick
signal, and the NVIC\_EnableIRQ() call ensures that a SysTick exception is
generated for each tick. All other functions in the Cortex version of the
KernelTimer class are essentially stubbed out (see the source for more details).
Note that the functions used in this call are part of the ARM Cortex
\hbox{\tt Microcontroller Software Interface Standard (cmsis), and are supplied by all}\\
parts vendors selling Cortex hardware. This greatly simplifies the design
of our port-code, since we can be reasonably assured that these APIs will
work the same on all devices.
The handler code called when a SysTick exception occurs is basically the
same as on other platforms (such as AVR), except that we explicitly clear the
"exception pending" bit before returning. This is implemented in the
following code:
@code{.cpp}
void SysTick_Handler(void)
#if KERNEL USE TIMERS
   TimerScheduler::Process();
#endif
#if KERNEL_USE_QUANTUM
    Quantum::UpdateTimer();
#endif
    // Clear the systick interrupt pending bit.
    SCB->ICSR |= SCB_ICSR_PENDSTCLR_Msk;
```

# **Critical Sections**

68 Mark3 Kernel Architecture

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

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

In general, CriticalSection::Enter() performs the following tasks:

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

Conversely, CriticalSection::Exit() performs the following tasks:

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

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

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

### **Summary**

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

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

# **C-language bindings**

Mark3 now includes an optional additional library with C language bindings for all core kernel APIs, known as Mark3C. This library alllows applications to be written in C, while still enjoying all of the benefits of the clean, modular design of the core RTOS kernel.

The C-language Mark3C APIs map directly to their Mark3 counterparts using a simple set of conventions, documented below. As a result, explicit API documentation for Mark3C is not necessary, as the functions map 1-1 to their C++ counterparts.

# 11.1 API Conventions

### 1) Static Methods:

### 2) Kernel Object Methods:

In short, any class instance is represented using an object handle, and is always passed into the relevant APIs as the first argument. Further, any method that returns a pointer to an object in the C++ implementation now returns a handle to that object.

#### 3) Overloaded Methods:

a) Methods overloaded with a Timeout parameter:

b) Methods overloaded based on number of arguments:

70 C-language bindings

```
<Object>.<MethodName>()
                                          Becomes
                                                      <ClassName>_<MethodName>(<ObjectHandle>)
<Object>.<MethodName>(<arg1>)
                                          Becomes
                                                      <ClassName>_<MethodName>1(<ObjectHandle>, <arg1>)
<Object>.<MethodName>(<arg1>, <arg2>)
                                                      <ClassName>_<MethodName>2(<ObjectHandle>, <arg1>, <arg2>
                                          Becomes
                                                      <ClassName>_<MethodName>(<ObjectHandle>)
<ClassName>::<MethodName>()
                                          Becomes
<ClassName>::<MethodName>(<arg1>)
                                                      <ClassName>_<MethodName>1(<ObjectHandle>, <argl>)
                                          Becomes
<ClassName>::<MethodName>(<arg1>, <arg2>) Becomes
                                                      <ClassName>_<MethodName>2(<ObjectHandle>, <arg1>, <arg2>
```

#### c) Methods overloaded base on parameter types:

#### d) Allocate-once memory allocation APIs

```
AutoAlloc::New<ObjectName> Becomes Alloc_<ObjectName> AutoAlloc::Allocate(uint16_t u16Size_) Becomes AutoAlloc(uint16_t u16Size_)
```

# 11.2 Allocating Objects

Aside from the API name translations, the object allocation scheme is the major different between Mark3C and Mark3. Instead of instantiating objects of the various kernel types, kernel objects must be declared using Declaration macros, which serve the purpose of reserving memory for the kernel object, and provide an opaque handle to that object memory. This is the case for statically-allocated objects, and objects allocated on the stack.

# Example: Declaring a thread

```
#include "mark3c.h"

// Statically-allocated
DECLARE_THREAD(hMyThread1);
...

// On stack
int main(void)
{
    DECLARE_THREAD(hMyThread2);
    ...
}

Where:

hMyThread1 - is a handle to a statically-allocated thread
hMyThread2 - is a handle to a thread allocated from the main stack.
```

Alternatively, the AutoAlloc APIs can be used to dynamically allocate objects, as demonstrated in the following example.

```
void Allocate_Example(void)
{
    Thread_t hMyThread = AutoAlloc_Thread();
    Thread_Init(hMyThread, awMyStack, sizeof(awMyStack), 1, MyFunction, 0);
}
```

Note that the relevant kernel-object Init() function *must* be called prior to using any kernel object, whether or not they have been allocated statically, or dynamically.

# **Release Notes**

# 12.1 R10 Release

- · New: Coroutines + Cooperative scheduler
- · New: Critical section APIs defined in kernel lib
- · New: RAII critical section (CriticalGuard object)
- · New: RAII scheduler-disabled context (SchedulerGuard object)
- Kernel code updated to use RAII critical sections instead of CS\_ENTER/CS\_EXIT macros
- · Updated documentation

# 12.2 R9 Release

- New: templated linked-lists to avoid explicit casting used in list traversal
- · New: ThreadListList class to efficiently track all threads in the system
- · Remove use of C-style casts in kernel
- · Remove use of 0 as nullptr in kernel
- · Refactor code to use constexpr instead of C-style preprocessor defines where possible
- · Refactor priority-map class as a set of template classes, reducing use of macros and defines
- Fix a "disappearing thread" bug where an inopportune context switch could cause a thread to get lost
- · Docs no longer build by default

# 12.3 R8 Release

- Structural changes to separate the kernel from the rest of Mark3-repo
- Cleanup and reformatting

72 Release Notes

# 12.4 R7 (Full Throttle) Release

- · Re-focusing project on kernel, integrating with 3rd party code instead of 1st party middleware
- · Re-focusing on atmega1284p and cortex-m as default targets
- New: Refactored codebase to C++14 standard
- · New: Moved non-kernel code, drivers, libs, and BSPs to separate repos from kernel
- · New: Modular repository-based structure, managed via Android's Repo tool
- · New: ConditionVariable kernel API
- · New: ReaderWriterLock kernel API
- · New: AutoAlloc redirects to user-defined allocators
- New: Global new() and delete() overrides redirect to AutoAlloc APIs
- · New: RAII Mutex Locking APIs
- · New: Support for cortex-a53 (aarch64) targets
- · New: Doxygen builds as part of cmake process
- · Updated Mark3c for new APIs
- · Moved driver layer out of the kernel
- · Moved all non-essential libraries out of the kernel (into other repos)
- · Build system supports modular BSP architecture
- · Removed fake idle-thread feature, since it doesn't support all targets
- · Unit tests and examples will run on any target with a BSP
- · Moved AVR-specific code out of the kernel (kernelaware debugging support)
- Remove most build-time configuration flags from mark3cfg.h, and remove associated ifdefs throughout the code
- · Support qemu-system-arm's Im3s6965 evb target, with semihosting
- · Support qemu-system-arm's rasbpi3 evb target, with semihosting
- · Incrase test coverage
- · Various bugfixes and improvements

# 12.5 R6 Release

- · New: Replace recursive-make build system with CMake and Ninja
- · New: Transitioned version control to Git from Subversion.
- · New: Socket library, implementing named "domain-socket" style IPC
- · New: State Machine framework library
- New: Software I2C library completed, with demo app
- · New: Kernel Timer loop can optionally be run within its own thread instead of a nested interrupt

12.6 R5 Release 73

- · New: UART drivers are all now abstracted throught UartDriver base class for portability
- · Experimental: Process library, allowing for the creation of resource-isolated processes
- Removed: Bare-metal support for Atmel SAMD20 (generic port still works)
- Cleanup all compiler warnings on atmega328p
- · Various Bugfixes and optimizations
- · Various Script changes related to automating the build + release process

# 12.6 R5 Release

- New: Shell library for creating responsive CLIs for embedded applications (M3Shell)
- · New: Stream library for creating thread-safe buffered streams (streamer)
- New: Blocking UART implementation for AVR (drvUARTplus)
- · New: "Extended context" kernel feature, which is used to implement thread-local storage
- · New: "Extra Checks" kernel feature, which enforces safe API usage under pain of Kernel Panic
- · New: Realtime clock library
- New: Example application + bsp for the open-hardware Mark3no development board (mark3no)
- New: Kernel objects descoped/destroyed while still in active use will now cause kernel panic
- New: Kernel callouts for thread creation/destruction/context switching, used for time tracking
- · New: Simple power management class
- · New: WIP software-based I2C + SPI drivers
- · Optimized thread scheduling via target-optimized "count-leading-zero" macros
- Expanded memutil library
- · Various optimizations of ARM Cortex-M assembly code
- · Various bugfixes to Timer code
- Improved stack overflow checking + warning (stack guard kernel feature)
- · AVR bootloader now supports targets with more than 64K of flash
- · Moved some port configuration out of platform.mak into header files in the kernel port code
- The usual minor bugfixes and "gentle refactoring"

74 Release Notes

## 12.7 R4 Release

- New: C-language bindings for Mark3 kernel (mark3c library)
- · New: Support for ARM Cortex-M3 and Cortex-M4 (floating point) targets
- · New: Support for Atmel AVR atmega2560 and arduino pro mega
- · New: Full-featured, lightweight heap implementation
- · New: Mailbox IPC class
- New: Notification object class
- · New: lighweight tracelogger/instrumentation implementation (buffalogger), with sample parser
- New: High-performance AVR Software UART implementation
- · New: Allocate-once "AutoAlloc" memory allocator
- · New: Fixed-time blocking/unblocking operations added to ThreadList/Blocking class
- · Placement-new supported for all kernel objects
- · Scheduler now supports up to 1024 levels of thread priority, up from 8 (configurable at build-time)
- · Kernel now uses stdint.h types for standard integers (instead of K CHAR, K ULONG, etc.)
- · Greatly expanded documentation, with many new examples covering all key kernel features
- · Expanded unit test coverage on AVR
- · Updated build system and scripts for easier kernel configuration
- · Updated builds to only attempt to build tests for supported platforms

## 12.8 R3 Release

- · New: Added support for MSP430 microcontrollers
- · New: Added Kernel Idle-Function hook to eliminate the need for a dedicated idle-thread (where supported)
- · New: Support for kernel-aware simulation and testing via flAVR AVR simulator
- Updated AVR driver selection
- · General bugfixes and maintenance
- · Expanded documentation and test coverage

# 12.9 R2

- Experimental release, using a "kernel transaction queue" for serializing kernel calls
- Works as a proof-of-concept, but abandoned due to overhead of the transaction mechanism in the general case.

# 12.10 R1 - 2nd Release Candidate

- New: Added support for ARM Cortex-M0 targets
- New: Added support for various AVR targets
- New: Timers now support a "tolerance" parameter for grouping timers with close expiry times
- Expanded scripts and auotmation used in build/test
- · Updated and expanded graphics APIs
- Large number of bugfixes

# 12.11 R1 - 1st Release Candidate

• Initial release, with support for AVR microcontrollers

76 Release Notes

# **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 and portcfg.h at compile time. Note that these sizes represent the echo 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 atmega1284p-based targets using gcc. Results are not necessarily indicative of relative or absolute performance on other platforms or toolchains.

# 13.1 Information

Date Profiled: Wed Jun 12 21:37:23 EDT 2019

# 13.2 Compiler Version

avr-gcc (GCC) 5.4.0 Copyright (C) 2015 Free Software Foundation, Inc. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

# 13.3 Profiling Results

# Mark3 Module Size Report:

```
- Atomic Operations. : 56 Bytes
- Allocate-once Heap. : 120 Bytes
- Synchronization Objects - Base Class : 126 Bytes
- Condition Variables (Synchronization Object) : 286 Bytes
- Coroutine task-list management : 122 Bytes
- Main coroutine task object : 336 Bytes
- Coroutine task scheduler : 318 Bytes
- Synchronization Object - Event Flag : 872 Bytes
- Synchronization Object - Event Flag : 872 Bytes
- Semaphore (Synchronization Object) : 568 Bytes
- Fundamental Kernel Linked-List Classes : 560 Bytes
- RAII Locking Support based on Mark3 Mutex class : 62 Bytes
- Mailbox IPC Support : 1064 Bytes
- Message-based IPC : 288 Bytes
- Mutex (Synchronization Object) : 658 Bytes
```

78 Code Size Profiling

# Mark3 Kernel Size Summary:

```
- Kernel : 3359 Bytes
- Synchronization Objects : 4630 Bytes
- Port : 1344 Bytes
- Features : 1460 Bytes
- Coroutines : 776 Bytes
- Untracked Objects : 0 Bytes
- Total Size : 11569 Bytes
```

# Namespace Index

# 14.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Mark3	91
Mark3::Atomic	
The Atomic namespace This utility module provides primatives for atomic operations - that is,	
operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here	
include Set/Add/Subtract, as well as an atomic test-and-set	102

80 Namespace Index

# **Hierarchical Index**

# 15.1 Class Hierarchy

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

Mark3::AutoAlloc
Mark3::BlockingObject
Mark3::EventFlag
Mark3::Mutex
Mark3::Notify
Mark3::Semaphore
Mark3::ConditionVariable
Mark3::CoScheduler
Mark3::CriticalGuard
Mark3::CriticalSection
Mark3::Kernel
Mark3::KernelSWI
Mark3::KernelTimer
Mark3::LinkList
Mark3::CircularLinkList
Mark3::TypedCircularLinkList< T >
Mark3::TypedCircularLinkList < Coroutine >
Mark3::CoList
Mark3::TypedCircularLinkList< Thread >
Mark3::ThreadList
Mark3::DoubleLinkList
Mark3::TypedDoubleLinkList< T >
Mark3::TypedDoubleLinkList< Mark3::Message >
Mark3::TypedDoubleLinkList< Mark3::ThreadList >
Mark3::TypedDoubleLinkList< Timer >
Mark3::TimerList
Mark3::LinkListNode
Mark3::ThreadList
$Mark 3:: Typed Link List Node < T > \dots \dots$
Mark3::TypedLinkListNode < Coroutine >
Mark3::Coroutine
Mark3::TypedLinkListNode < Message >
Mark3::Message
Mark3::TypedLinkListNode < Thread >

82 Hierarchical Index

Mark3::Thread
Mark3::TypedLinkListNode < Timer >
Mark3::Timer
Mark3::LockGuard
Mark3::Mailbox
Mark3::MemUtil
Mark3::MessagePool
Mark3::MessageQueue
$Mark 3:: Priority Map L1 < T, C> \dots $
$Mark 3:: Priority Map L2 < T, C> \dots $
Mark3::ProfileTimer
Mark3::Quantum
Mark3::ReaderWriterLock
Mark3::Scheduler
Mark3::SchedulerGuard
Mark3::Streamer
Mark3::ThreadListList
Mark3::ThreadPort
Mark3::TimerScheduler
$Mark 3:: Token\_t  \dots  \dots  \dots  285$

# **Class Index**

# 16.1 Class List

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

Mark3::AutoAlloc	
, , , , , , , , , , , , , , , , , , , ,	105
Mark3::BlockingObject	
The BlockingObject class. 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	110
Mark3::CircularLinkList	
3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,	115
Mark3::CoList	
The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list	117
Mark3::ConditionVariable	
The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel	120
Mark3::Coroutine	
The Coroutine class implements a lightweight, run-to-completion task that forms the basis for cooperative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines	124
Mark3::CoScheduler	
The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr	127

84 Class Index

Mark3::CriticalGuard	
The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released	131
Mark3::CriticalSection	
The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port	132
Mark3::DoubleLinkList The DoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type	133
Mark3::EventFlag  The EventFlag class. This class implements 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	135
Mark3::Kernel	
The Kernel Class encapsulates all of the kernel startup, configuration and management functions  Mark3::KernelSWI	140
The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer	150
Mark3::KernelTimer	
Timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer	151
Mark3::LinkList The LinkList Class Abstract-data-type from which all other linked-lists are derived	152
Mark3::LinkListNode  The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them	155
Mark3::LockGuard  The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object.  Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met	158
Mark3::Mailbox	
The Mailbox class. This class implements an IPC mechnism based on sending/receiving envelopes containing data of a fixed size, configured at initialization) that reside within a buffer of memory provided by the user	161
Mark3::MemUtil	
String and Memory manipu32ation class	173
Mark3::Message  Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver .	182
Mark3::MessagePool	
The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool	186
Mark3::MessageQueue  The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts	189
Mark3::Mutex	
The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject	193

16.1 Class List 85

Mark3::Notify	
The Notify class. This class provides a blocking object type that allows one or more threads to wait for an event to occur before resuming operation	199
Mark3::PriorityMapL1 < T, C >	
The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in	
the objects internal storage represents a priority. When a bit is set, it indicates that something	
is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities	
are scheduled at that priority. This object provides the fundamental logic required to implement	
efficient priority-based scheduling for the thread + coroutine schedulers in the kernel	
Mark3::PriorityMapL2< T, C >	
The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in	
the objects internal storage represents a priority. When a bit is set, it indicates that something	
is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities	
are scheduled at that priority. This object provides the fundamental logic required to implement	
efficient priority-based scheduling for the thread + coroutine schedulers in the kernel	200
Mark3::ProfileTimer	
Profiling timer. This class is used to perform high-performance profiling of code to see how	
int32_t certain operations take. useful in instrumenting the performance of key algorithms and	
time-critical operations to ensure real-timer behavior	210
Mark3::Quantum	
The Quantum Class. Static-class used to implement Thread quantum functionality, which is	
fundamental to round-robin thread scheduling	213
Mark3::ReaderWriterLock	
The ReaderWriterLock class. This class implements an object that marshalls access to a re-	
source based on the intended usage of the resource. A reader-writer lock permits multiple con-	
current read access, or single-writer access to a resource. If the object holds a write lock, other	
writers, and all readers will block until the writer is finished. If the object holds reader locks, all	
writers will block until all readers are finished before the first writer can take ownership of the	
resource. This is based upon lower-level synchronization primatives, and is somewhat more	
heavyweight than primative synchronization types	216
Mark3::Scheduler	
The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all	
active threads managed by the kernel	
Mark3::SchedulerGuard	
The SchedulerGuard class This class implements RAII-based control of the scheduler's global	
state. Upon object construction, the scheduler's state is cached locally and the scheduler is	
disabled (if not already disabled). Upon object destruction, the scheduler's previous state is	
restored. This object is interrupt-safe, although it has no effect when called from an interrupt	
given that interrupts are inherently higher-priority than threads	
Mark3::Semaphore	
Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base	
class	
Mark3::Streamer	
The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe	
·	
methods for writing-to and reading-from the buffer. Objects of this class type are designed to be	
shared between threads, or between threads and interrupts	234
Mark3::Thread	
The Thread Class. This object providing the fundamental thread control data structures and func-	
tions that define a single thread of execution in the Mark3 operating system. It is the fundamental	
data type used to provide multitasking support in the kernel	242
Mark3::ThreadList	
The ThreadList Class. This class is used for building thread-management facilities, such as	
schedulers, and blocking objects	264
Mark3::ThreadListList	
The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in	
time, the list can be traversed to get a complete view of all running, blocked, or stopped threads	
in the system	270

86 Class Index

Mark3::ThreadPort	
The ThreadPort Class defines the target-specific functions required by the kernel for threading	272
Mark3::Timer	
The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler	273
Mark3::TimerList	
TimerList class. This class implements a doubly-linked-list of timer objects	280
Mark3::TimerScheduler	
The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system	283
Mark3::Token_t	
Token descriptor struct format	285
Mark3::TypedCircularLinkList< T >	
The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types	286
Mark3::TypedDoubleLinkList< T >	
The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types	290
Mark3::TypedLinkListNode < T >	
The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class	293

# File Index

# 17.1 File List

Here is a list of all files with brief descriptions:

/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp	
Implementation of C-language wrappers for the Mark3 kernel	295
/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/fake_types.h	
C-struct definitions that mirror	358
/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/mark3c.h	
Header providing C-language API bindings for the Mark3 kernel	360
/home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp	
Implementation of memory, string, and conversion routines	424
/home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h	
Utility class containing memory, string, and conversion routines	431
/home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp	
Thread/Interrupt-safe byte-based data streaming	434
/home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h	
Thread/Interrupt-safe byte-based data streaming	433
/home/moslevin/projects/m3-repo/kernel/src/atomic.cpp	
Basic Atomic Operations	455
/home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp	
Automatic memory allocation for kernel objects	456
/home/moslevin/projects/m3-repo/kernel/src/blocking.cpp	
Implementation of base class for blocking objects	457
/home/moslevin/projects/m3-repo/kernel/src/colist.cpp	
CoRoutine List structure implementation	459
/home/moslevin/projects/m3-repo/kernel/src/condvar.cpp	
Condition Variable implementation	460
/home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp	
Coroutine object implementation	461
/home/moslevin/projects/m3-repo/kernel/src/cosched.cpp	
CoRoutine Scheduler implementation	463
/home/moslevin/projects/m3-repo/kernel/src/eventflag.cpp	
Event Flag Blocking Object/IPC-Object implementation	464
/home/moslevin/projects/m3-repo/kernel/src/kernel.cpp	
Kernel initialization and startup code	468
/home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp	
	469
/home/moslevin/projects/m3-repo/kernel/src/ll.cpp	
Core Linked-List implementation, from which all kernel objects are derived	472

88 File Index

/home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp	
Mutex RAII helper class	474
/home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp	
Mailbox + Envelope IPC mechanism	475
/home/moslevin/projects/m3-repo/kernel/src/message.cpp	
Inter-thread communications via message passing	478
/home/moslevin/projects/m3-repo/kernel/src/mutex.cpp	
Mutual-exclusion object	480
/home/moslevin/projects/m3-repo/kernel/src/notify.cpp	
Lightweight thread notification - blocking object	483
/home/moslevin/projects/m3-repo/kernel/src/profile.cpp	
Code profiling utilities	486
/home/moslevin/projects/m3-repo/kernel/src/quantum.cpp	
Thread Quantum Implementation for Round-Robin Scheduling	552
/home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp	
Reader-writer lock implementation	554
/home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp	• • • • • • • • • • • • • • • • • • • •
Strict-Priority + Round-Robin thread scheduler implementation	555
/home/moslevin/projects/m3-repo/kernel/src/thread.cpp	555
Platform-Independent thread class Definition	557
/home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp	337
Thread linked-list definitions	563
/home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp	505
	565
· · · · · · · · · · · · · · · · · · ·	363
/home/moslevin/projects/m3-repo/kernel/src/timer.cpp	ECE
Timer implementations	565
/home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp	<b>507</b>
Implements timer list processing algorithms, responsible for all timer tick and expiry logic	567
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp	400
Kernel Software interrupt implementation for ATMega1284p	438
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp	
Kernel Timer Implementation for ATMega1284p	439
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp	
ATMega1284p Multithreading	452
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/portcfg.h	
Mark3 Port Configuration	442
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/threadport.h	
ATMega1284p Multithreading support	448
/home/moslevin/projects/m3-repo/kernel/src/public/atomic.h	
Basic Atomic Operations	487
/home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h	
Automatic memory allocation for kernel objects	489
/home/moslevin/projects/m3-repo/kernel/src/public/blocking.h	
Blocking object base class declarations	491
/home/moslevin/projects/m3-repo/kernel/src/public/colist.h	
CoRoutine List structure implementation	493
/home/moslevin/projects/m3-repo/kernel/src/public/condvar.h	
Condition Variable implementation	494
/home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h	
CoRoutine implementation	495
/home/moslevin/projects/m3-repo/kernel/src/public/cosched.h	
CoRoutine Scheduler implementation	497
/home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h	
RAII Critical Section Implementation	498
/home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h	
Critical Section Support	499
/home/moslevin/projects/m3-repo/kernel/src/public/eventflag.h	
Event Flag Blocking Object/IPC-Object definition	500

17.1 File List 89

/home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h	
	501
/home/moslevin/projects/m3-repo/kernel/src/public/kernel.h	
Kernel initialization and startup class	502
/home/moslevin/projects/m3-repo/kernel/src/public/kerneldebug.h	
Macros and functions used for assertions, kernel traces, etc	504
/home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h	
· ·	506
/home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h	
	507
/home/moslevin/projects/m3-repo/kernel/src/public/kerneltypes.h	
• • • • • • • • • • • • • • • • • • • •	508
/home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h	
· · · · · · · · · · · · · · · · · · ·	509
/home/moslevin/projects/m3-repo/kernel/src/public/ll.h	
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 u16 to keep our RTOS code efficient and logically-	
separated	510
/home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h	
	513
/home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h	
Mailbox + Envelope IPC Mechanism	514
/home/moslevin/projects/m3-repo/kernel/src/public/manual.h	516
/home/moslevin/projects/m3-repo/kernel/src/public/mark3.h	
Single include file given to users of the Mark3 Kernel API	517
/home/moslevin/projects/m3-repo/kernel/src/public/mark3cfg.h	
Mark3 Kernel Configuration This file is used to configure the kernel for your specific application	
	518
/home/moslevin/projects/m3-repo/kernel/src/public/message.h	
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.	
1	522
/home/moslevin/projects/m3-repo/kernel/src/public/mutex.h	
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 re-	
source, 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	
	524
/home/moslevin/projects/m3-repo/kernel/src/public/notify.h	<i>52</i> 4
· · · · · · · · · · · · · · · · · · ·	526
/home/moslevin/projects/m3-repo/kernel/src/public/paniccodes.h	320
	527
/home/moslevin/projects/m3-repo/kernel/src/public/priomap.h	027
	530
/home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h	
	531
/home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h	
	533
/home/moslevin/projects/m3-repo/kernel/src/public/profile.h	200
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	
· · · · · · · · · · · · · · · · · · ·	535
	536
· · · · · · · · · · · · · · · · · · ·	

90 File Index

/home/moslevin/projects/m3-repo/kernel/src/public/quantum.h	
Thread Quantum declarations for Round-Robin Scheduling	537
/home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h	
Reader-Writer lock implementation	538
/home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h	
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 op-	
erating systems. This scheduling mechanism is referred to as priority round-robin	539
/home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h	
RAII Scheduler Locking	541
$/home/moslevin/projects/m3-repo/kernel/src/public/sizeprofile.h \\ \ldots \\ $	542
/home/moslevin/projects/m3-repo/kernel/src/public/thread.h	
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	543
/home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h	
Thread linked-list declarations	546
/home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h	
Class implementing a doubly-linked list of thread lists	547
/home/moslevin/projects/m3-repo/kernel/src/public/timer.h	
Timer object declarations	548
/home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h	
Timer list declarations	550
/home/moslevin/projects/m3-repo/kernel/src/public/timerscheduler.h	
Timer echeduler declarations	551

# **Namespace Documentation**

# 18.1 Mark3 Namespace Reference

# **Namespaces**

#### Atomic

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

### **Classes**

## class AutoAlloc

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

#### · class BlockingObject

The BlockingObject class. 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.

# · class CircularLinkList

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

#### class CoList

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

## · class ConditionVariable

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

#### class Coroutine

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

### · class CoScheduler

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

### · class CriticalGuard

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

#### · class CriticalSection

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

#### class DoubleLinkList

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

## class EventFlag

The EventFlag class. This class implements 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.

#### · class Kernel

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

# class KernelSWI

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

#### class KernelTimer

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

#### · class LinkList

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

#### · class LinkListNode

The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

## class LockGuard

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

## class Mailbox

The Mailbox class. This class implements an IPC mechnism based on sending/receiving envelopes containing data of a fixed size, configured at initialization) that reside within a buffer of memory provided by the user.

# • class MemUtil

String and Memory manipu32ation class.

#### · class Message

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

### class MessagePool

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

#### • class MessageQueue

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

#### · class Mutex

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

· class Notify

The Notify class. This class provides a blocking object type that allows one or more threads to wait for an event to occur before resuming operation.

#### class PriorityMapL1

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

#### class PriorityMapL2

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

#### class ProfileTimer

Profiling timer. This class is used to perform high-performance profiling of code to see how int32\_t certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

#### · class Quantum

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

#### class ReaderWriterLock

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

#### · class Scheduler

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

### · class SchedulerGuard

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

#### class Semaphore

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

#### · class Streamer

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

#### class Thread

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

### · class ThreadList

The ThreadList Class. This class is used for building thread-management facilities, such as schedulers, and blocking objects.

#### · class ThreadListList

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

### class ThreadPort

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

#### · class Timer

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

· class TimerList

the TimerList class. This class implements a doubly-linked-list of timer objects.

class TimerScheduler

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

· struct Token t

Token descriptor struct format.

· class TypedCircularLinkList

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

class TypedDoubleLinkList

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

class TypedLinkListNode

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

## **Typedefs**

```
    using AutoAllocAllocator_t = void *(*)(AutoAllocType eType_, size_t sSize_)
```

```
    using AutoAllocFree_t = void(*)(AutoAllocType eType_, void *pvObj_)
```

- using CoPrioMap = PriorityMapL1 < PORT\_PRIO\_TYPE, PORT\_COROUTINE\_PRIORITIES >
- using CoroutineHandler = void(\*)(Coroutine \*pclCaller\_, void \*pvContext\_)
- using PanicFunc = void(\*)(uint16\_t u16PanicCode\_)
- using IdleFunc = void(\*)()
- using ThreadEntryFunc = void(\*)(void \*pvArg\_)
- using PriorityMap = PriorityMapL1 < PORT\_PRIO\_TYPE, KERNEL\_NUM\_PRIORITIES >
- using ThreadCreateCallout = void(\*)(Thread \*pclThread\_)
- using ThreadExitCallout = void(\*)(Thread \*pclThread )
- using ThreadContextCallout = void(\*)(Thread \*pclThread\_)
- using TimerCallback = void(\*)(Thread \*pclOwner\_, void \*pvData\_)

#### **Enumerations**

```
    enum AutoAllocType::uint8_t {
        AutoAllocType::EventFlag, AutoAllocType::MailBox, AutoAllocType::Message, AutoAllocType::MessagePool,
        AutoAllocType::MessageQueue, AutoAllocType::Mutex, AutoAllocType::Notify, AutoAllocType::Semaphore,
        AutoAllocType::Thread, AutoAllocType::Timer, AutoAllocType::ConditionVariable, AutoAllocType::Reader
        WriterLock,
        AutoAllocType::User, AutoAllocType::Raw = 0xFF }
```

enum EventFlagOperation::uint8\_t {
 EventFlagOperation::All\_Set = 0, EventFlagOperation::Any\_Set, EventFlagOperation::All\_Clear, EventFlagOperation::Any\_Clear,
 EventFlagOperation::Pending Unblock }

```
    enum ThreadState::uint8_t {
        ThreadState::Exit = 0, ThreadState::Blocked, ThreadState::Stop,
        ThreadState::Invalid }
```

#### **Functions**

```
• static void KernelTimer Task (void *unused)
```

- static void Thread\_Switch (void)
- ISR (INT2\_vect) \_\_attribute\_\_((signal

ISR(INT2\_vect) SWI using INT2 - used to trigger a context switch.

- uint8\_t PORT\_CLZ (uint8\_t in\_)
- void PORT\_IRQ\_ENABLE ()
- void PORT\_IRQ\_DISABLE ()
- void PORT\_CS\_ENTER ()
- void PORT CS EXIT ()
- K\_WORD PORT\_CS\_NESTING ()

### **Variables**

- static constexpr auto uMaxTimerTicks = uint32\_t { 0x7FFFFFFF }
  - Maximum value to set.
- static constexpr auto uTimerTicksInvalid = uint32\_t { 0 }
- static constexpr auto uTimerFlagOneShot = uint8\_t { 0x01 }

Timer is one-shot.

static constexpr auto uTimerFlagActive = uint8\_t { 0x02 }

Timer is currently active.

static constexpr auto uTimerFlagCallback = uint8\_t { 0x04 }

Timer is pending a callback.

static constexpr auto uTimerFlagExpired = uint8\_t { 0x08 }

Timer is actually expired.

- naked
- static constexpr auto SR\_ = uint8\_t{0x3F}
- K\_WORD g\_kwSFR
- K\_WORD g\_kwCriticalCount

### 18.1.1 Typedef Documentation

```
18.1.1.1 AutoAllocAllocator_t
```

```
using Mark3::AutoAllocAllocator_t = typedef void* (*) (AutoAllocType eType_, size_t sSize_)
```

Definition at line 53 of file autoalloc.h.

```
18.1.1.2 AutoAllocFree_t
```

```
using Mark3::AutoAllocFree_t = typedef void (*)(AutoAllocType eType_, void* pvObj_)
```

Definition at line 54 of file autoalloc.h.

#### 18.1.1.3 CoPrioMap

```
using Mark3::CoPrioMap = typedef PriorityMapL1<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
```

Definition at line 30 of file coroutine.h.

#### 18.1.1.4 CoroutineHandler

```
using Mark3::CoroutineHandler = typedef void (*)(Coroutine* pclCaller_, void* pvContext_)
```

Definition at line 40 of file coroutine.h.

#### 18.1.1.5 IdleFunc

```
using Mark3::IdleFunc = typedef void (*)()
```

Function pointer type used to implement the idle function, where support for an idle function (as opposed to an idle thread) exists.

Definition at line 37 of file kerneltypes.h.

### 18.1.1.6 PanicFunc

```
using Mark3::PanicFunc = typedef void (*)(uint16_t u16PanicCode_)
```

Function pointer type used to implement kernel-panic handlers.

Definition at line 30 of file kerneltypes.h.

### 18.1.1.7 PriorityMap

```
using Mark3::PriorityMap = typedef PriorityMapL1<PRIO_TYPE, KERNEL_NUM_PRIORITIES>
```

Definition at line 29 of file priomap.h.

### 18.1.1.8 ThreadContextCallout

```
using Mark3::ThreadContextCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 54 of file thread.h.

#### 18.1.1.9 ThreadCreateCallout

```
using Mark3::ThreadCreateCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 52 of file thread.h.

#### 18.1.1.10 ThreadEntryFunc

```
using Mark3::ThreadEntryFunc = typedef void (*)(void* pvArg_)
```

Function pointer type used for thread entrypoint functions

Definition at line 43 of file kerneltypes.h.

#### 18.1.1.11 ThreadExitCallout

```
using Mark3::ThreadExitCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 53 of file thread.h.

### 18.1.1.12 TimerCallback

```
using Mark3::TimerCallback = typedef void (*)(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 50 of file timer.h.

### 18.1.2 Enumeration Type Documentation

#### 18.1.2.1 AutoAllocType

```
enum Mark3::AutoAllocType : uint8_t [strong]
```

### Enumerator

EventFlag	
MailBox	
Message	
MessagePool	
MessageQueue	
Mutex	
Notify	
Semaphore	
Thread	
Timer	
ConditionVariable	
ReaderWriterLock	
User	
Raw	

Definition at line 32 of file autoalloc.h.

### 18.1.2.2 EventFlagOperation

```
enum Mark3::EventFlagOperation : uint8_t [strong]
```

This enumeration describes the different operations supported by the event flag blocking object.

# Enumerator

All_Set	Block until all bits in the specified bitmask are set.
Any_Set	Block until any bits in the specified bitmask are set.
All_Clear	Block until all bits in the specified bitmask are cleared.
Any_Clear	Block until any bits in the specified bitmask are cleared.
Pending_Unblock	Special code. Not used by user

Definition at line 50 of file kerneltypes.h.

### 18.1.2.3 ThreadState

```
enum Mark3::ThreadState : uint8_t [strong]
```

Enumeration representing the different states a thread can exist in

### Enumerator

Exit		
Ready	$\prod$	
Blocked		
Stop	П	Generated by Doxyge
Invalid		

Definition at line 62 of file kerneltypes.h.

### 18.1.3 Function Documentation

ISR(INT2 \_vect) SWI using INT2 - used to trigger a context switch.

Definition at line 137 of file threadport.cpp.

### 18.1.3.2 KernelTimer\_Task()

Definition at line 44 of file kerneltimer.cpp.

# 18.1.3.3 PORT\_CLZ()

Definition at line 144 of file threadport.h.

### 18.1.3.4 PORT\_CS\_ENTER()

```
void Mark3::PORT_CS_ENTER ( ) [inline]
```

Definition at line 167 of file threadport.h.

### 18.1.3.5 PORT\_CS\_EXIT()

```
void Mark3::PORT_CS_EXIT ( ) [inline]
```

Definition at line 178 of file threadport.h.

### 18.1.3.6 PORT\_CS\_NESTING()

```
K_WORD Mark3::PORT_CS_NESTING ( ) [inline]
```

Definition at line 187 of file threadport.h.

### 18.1.3.7 PORT\_IRQ\_DISABLE()

```
void Mark3::PORT_IRQ_DISABLE ( ) [inline]
```

Definition at line 161 of file threadport.h.

### 18.1.3.8 PORT\_IRQ\_ENABLE()

```
void Mark3::PORT_IRQ_ENABLE ( ) [inline]
```

Definition at line 155 of file threadport.h.

### 18.1.3.9 Thread\_Switch()

Definition at line 94 of file threadport.cpp.

### 18.1.4 Variable Documentation

### 18.1.4.1 g\_kwCriticalCount

```
K_WORD Mark3::g_kwCriticalCount
```

Definition at line 36 of file threadport.cpp.

### 18.1.4.2 g\_kwSFR

```
K_WORD Mark3::g_kwSFR
```

Definition at line 35 of file threadport.cpp.

18.1.4.3 naked

Mark3::naked

Definition at line 136 of file threadport.cpp.

```
18.1.4.4 SR
```

```
constexpr auto Mark3::SR_ = uint8_t{0x3F} [static]
```

Definition at line 137 of file threadport.h.

#### 18.1.4.5 uMaxTimerTicks

```
constexpr auto Mark3::uMaxTimerTicks = uint32_t { 0x7FFFFFFF } [static]
```

Maximum value to set.

Definition at line 32 of file timer.h.

#### 18.1.4.6 uTimerFlagActive

```
constexpr auto Mark3::uTimerFlagActive = uint8_t { 0x02 } [static]
```

Timer is currently active.

Definition at line 35 of file timer.h.

### 18.1.4.7 uTimerFlagCallback

```
constexpr auto Mark3::uTimerFlagCallback = uint8_t { 0x04 } [static]
```

Timer is pending a callback.

Definition at line 36 of file timer.h.

#### 18.1.4.8 uTimerFlagExpired

```
constexpr auto Mark3::uTimerFlagExpired = uint8_t { 0x08 } [static]
```

Timer is actually expired.

Definition at line 37 of file timer.h.

#### 18.1.4.9 uTimerFlagOneShot

```
constexpr auto Mark3::uTimerFlagOneShot = uint8_t { 0x01 } [static]
```

Timer is one-shot.

Definition at line 34 of file timer.h.

#### 18.1.4.10 uTimerTicksInvalid

```
constexpr auto Mark3::uTimerTicksInvalid = uint32_t { 0 } [static]
```

Definition at line 33 of file timer.h.

# 18.2 Mark3::Atomic Namespace Reference

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

### **Functions**

```
template<typename T >
T Set (T *pSource_, T val_)
```

Set Set a variable to a given value in an uninterruptable operation.

• template<typename T >

```
T Add (T *pSource_, T val_)
```

Add Add a value to a variable in an uninterruptable operation.

 $\bullet \ \ template {<} typename \ T >$ 

```
T Sub (T *pSource_, T val_)
```

Sub Subtract a value from a variable in an uninterruptable operation.

bool TestAndSet (bool \*pbLock)

TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

### 18.2.1 Detailed Description

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

# 18.2.2 Function Documentation

### 18.2.2.1 Add()

Add Add a value to a variable in an uninterruptable operation.

#### **Parameters**

p⇔ Source⇔	Pointer to a variable
_	
val_	Value to add to the variable

#### Returns

Previously-held value in pSource\_

Definition at line 64 of file atomic.h.

### 18.2.2.2 Set()

Set Set a variable to a given value in an uninterruptable operation.

#### **Parameters**

p⊷ Source⊷	Pointer to a variable to set the value of
_	
val_	New value to set in the variable

#### Returns

Previously-set value

Definition at line 47 of file atomic.h.

### 18.2.2.3 Sub()

Sub Subtract a value from a variable in an uninterruptable operation.

#### **Parameters**

p <i>⊷</i> Source <i>⊷</i>	Pointer to a variable
_	
val_	Value to subtract from the variable

#### Returns

Previously-held value in pSource\_

Definition at line 81 of file atomic.h.

### 18.2.2.4 TestAndSet()

TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

If the value is false, set the variable to true, and return the previously-held value.

If the value is already true, return true.

### **Parameters**

pbLoci	Pointer to a value to test against. This will always be set to "true" at the end of a call to TestAndSet.

### Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 26 of file atomic.cpp.

# **Chapter 19**

# **Class Documentation**

### 19.1 Mark3::AutoAlloc Class Reference

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

```
#include <autoalloc.h>
```

### **Static Public Member Functions**

static void Init (void)

Init Initialize the AutoAllocator before use. Called by Kernel::Init().

- static void SetAllocatorFunctions (AutoAllocAllocator\_t pfAllocator\_, AutoAllocFree\_t pfFree\_)
  - SetAllocatorFunctions Set the functions used by this class to allocate/free memory used in the kernel.
- template<typename T , AutoAllocType e> static T \* NewObject ()
- template<typename T , AutoAllocType e> static void DestroyObject (T \*pObj\_)
- static void \* NewUserTypeAllocation (uint8 t eUserType )

NewUserTypeAllocation Attempt to allocate a user-defined object type from the heap.

static void DestroyUserTypeAllocation (uint8\_t eUserType\_, void \*pvObj\_)

DestroyUserTypeAllocation Free a previously allocated user-defined object.

• static void \* NewRawData (size t sSize )

NewRawData Attempt to allocate a blob of raw data from the heap.

static void DestroyRawData (void \*pvData\_)

DestroyRawData Free a previously allocated blob of data allocated via NewRawData()

### **Static Private Member Functions**

- static void \* Allocate (AutoAllocType eType\_, size\_t sSize\_)
- static void Free (AutoAllocType eType\_, void \*pvObj\_)

#### **Static Private Attributes**

static AutoAllocAllocator\_t m\_pfAllocator

Function used to allocate objects.

static AutoAllocFree\_t m\_pfFree

Funciton used to free objectss.

#### 19.1.1 Detailed Description

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

Definition at line 82 of file autoalloc.h.

#### 19.1.2 Member Function Documentation

#### 19.1.2.1 Allocate()

Definition at line 58 of file autoalloc.cpp.

#### 19.1.2.2 DestroyObject()

Template function used to manage the destruction and de-allocation of predefined kernel objects

Definition at line 116 of file autoalloc.h.

#### 19.1.2.3 DestroyRawData()

DestroyRawData Free a previously allocated blob of data allocated via NewRawData()

#### **Parameters**

pv⊷	pointer to previously-created data object
Data_	

Definition at line 105 of file autoalloc.cpp.

### 19.1.2.4 DestroyUserTypeAllocation()

DestroyUserTypeAllocation Free a previously allocated user-defined object.

### **Parameters**

pvObj_	Pointer to previously-allocated object, allocated through NewUserTypeAllocation ()
eUser⊷	User defined object type, interpreted by the allocator function
Type_	

Definition at line 95 of file autoalloc.cpp.

### 19.1.2.5 Free()

Definition at line 67 of file autoalloc.cpp.

#### 19.1.2.6 Init()

Init Initialize the AutoAllocator before use. Called by Kernel::Init().

Definition at line 83 of file autoalloc.cpp.

### 19.1.2.7 NewObject()

```
template<typename T , AutoAllocType e>
static T* Mark3::AutoAlloc::NewObject ( ) [inline], [static]
```

Template function used to manage the allocation of predefined kernel object types

Definition at line 103 of file autoalloc.h.

#### 19.1.2.8 NewRawData()

NewRawData Attempt to allocate a blob of raw data from the heap.

#### **Parameters**

<i>s</i> ⊷	Size of the data blob (in bytes)
Size⊷	

### Returns

pointer to newly-allocated blob of data, or nullptr on error.

Definition at line 100 of file autoalloc.cpp.

### 19.1.2.9 NewUserTypeAllocation()

NewUserTypeAllocation Attempt to allocate a user-defined object type from the heap.

### **Parameters**

eUser⊷	User defined object type, interpreted by the allocator function
Type_	

#### Returns

pointer to a newly-created object, or nullptr on error.

Definition at line 90 of file autoalloc.cpp.

#### 19.1.2.10 SetAllocatorFunctions()

SetAllocatorFunctions Set the functions used by this class to allocate/free memory used in the kernel.

#### **Parameters**

pf↔ Allocator↔ -	Function to allocate an object based on its type and/or size
pfFree_	Function to free a previously-allocated object

Definition at line 76 of file autoalloc.cpp.

#### 19.1.3 Member Data Documentation

### 19.1.3.1 m\_pfAllocator

```
AutoAllocAllocator_t Mark3::AutoAlloc::m_pfAllocator [static], [private]
```

Function used to allocate objects.

Definition at line 157 of file autoalloc.h.

### 19.1.3.2 m\_pfFree

```
AutoAllocFree_t Mark3::AutoAlloc::m_pfFree [static], [private]
```

Funciton used to free objectss.

Funciton used to free objects.

Definition at line 158 of file autoalloc.h.

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

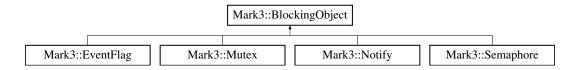
- /home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h
- /home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp

# 19.2 Mark3::BlockingObject Class Reference

The BlockingObject class. 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.

#include <blocking.h>

Inheritance diagram for Mark3::BlockingObject:



#### **Public Member Functions**

- · BlockingObject ()
- ∼BlockingObject ()

#### **Protected Member Functions**

void Block (Thread \*pclThread\_)

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

void BlockPriority (Thread \*pclThread\_)

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

void UnBlock (Thread \*pclThread\_)

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

void SetInitialized (void)

SetInitialized.

• bool IsInitialized (void)

IsInitialized.

### **Protected Attributes**

- ThreadList m\_clBlockList
- uint8\_t m\_u8Initialized

### **Static Protected Attributes**

- static constexpr auto m uBlockingInvalidCookie = uint8 t { 0x3C }
- static constexpr auto m\_uBlockingInitCookie = uint8\_t { 0xC3 }

### 19.2.1 Detailed Description

The BlockingObject class. 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.

#### 19.2.2 Constructor & Destructor Documentation

#### 19.2.2.1 BlockingObject()

```
Mark3::BlockingObject::BlockingObject ( ) [inline]
```

Definition at line 68 of file blocking.h.

#### 19.2.2.2 ∼BlockingObject()

```
Mark3::BlockingObject::~BlockingObject ( ) [inline]
```

Definition at line 69 of file blocking.h.

#### 19.2.3 Member Function Documentation

### 19.2.3.1 Block()

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

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

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

#### **Parameters**

pcl←	Pointer to the thread object that will be blocked.
Thread_	

Definition at line 26 of file blocking.cpp.

### 19.2.3.2 BlockPriority()

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

### **Parameters**

pcl←	Pointer to the Thread to Block.
Thread_	

Definition at line 41 of file blocking.cpp.

### 19.2.3.3 IsInitialized()

IsInitialized.

Returns

true if initialized, false if object uninitialized

Definition at line 123 of file blocking.h.

### 19.2.3.4 SetInitialized()

SetInitialized.

Definition at line 117 of file blocking.h.

# 19.2.3.5 UnBlock()

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

#### **Parameters**

pcl←	Pointer to the thread to unblock.
Thread_	

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

Definition at line 56 of file blocking.cpp.

#### 19.2.4 Member Data Documentation

#### 19.2.4.1 m\_clBlockList

```
ThreadList Mark3::BlockingObject::m_clBlockList [protected]
```

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

Definition at line 133 of file blocking.h.

### 19.2.4.2 m\_u8Initialized

```
uint8_t Mark3::BlockingObject::m_u8Initialized [protected]
```

Token used to check whether or not the object has been initialized prior to use.

Definition at line 139 of file blocking.h.

#### 19.2.4.3 m\_uBlockingInitCookie

```
constexpr auto Mark3::BlockingObject::m_uBlockingInitCookie = uint8_t { 0xC3 } [static],
[protected]
```

Definition at line 127 of file blocking.h.

#### 19.2.4.4 m\_uBlockingInvalidCookie

```
constexpr auto Mark3::BlockingObject::m_uBlockingInvalidCookie = uint8_t { 0x3C } [static],
[protected]
```

Definition at line 126 of file blocking.h.

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

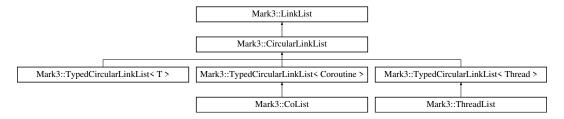
- /home/moslevin/projects/m3-repo/kernel/src/public/blocking.h
- /home/moslevin/projects/m3-repo/kernel/src/blocking.cpp

### 19.3 Mark3::CircularLinkList Class Reference

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

#include <ll.h>

Inheritance diagram for Mark3::CircularLinkList:



#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- CircularLinkList ()
- void Add (LinkListNode \*node\_)

Add the linked list node to this linked list.

void Remove (LinkListNode \*node\_)

Remove Add the linked list node to this linked list.

void PivotForward ()

 $\textit{PivotForward Pivot the head of the circularly linked list forward ( \textit{Head} = \textit{Head-}>\textit{next}, \textit{Tail} = \textit{Tail-}>\textit{next})}$ 

void PivotBackward ()

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

void InsertNodeBefore (LinkListNode \*node\_, LinkListNode \*insert\_)

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

#### **Additional Inherited Members**

### 19.3.1 Detailed Description

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

Definition at line 218 of file II.h.

## 19.3.2 Constructor & Destructor Documentation

#### 19.3.2.1 CircularLinkList()

Mark3::CircularLinkList::CircularLinkList ( ) [inline]

Definition at line 222 of file II.h.

### 19.3.3 Member Function Documentation

### 19.3.3.1 Add()

Add the linked list node to this linked list.

#### **Parameters**

node⊷	Pointer to the node to add
_	

Definition at line 81 of file II.cpp.

### 19.3.3.2 InsertNodeBefore()

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

#### **Parameters**

node⊷	Node to insert into the list
_	
insert←	Insert point.
_	

Definition at line 153 of file II.cpp.

### 19.3.3.3 operator new()

Definition at line 221 of file II.h.

#### 19.3.3.4 PivotBackward()

```
void Mark3::CircularLinkList::PivotBackward ( )
```

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

Definition at line 144 of file II.cpp.

#### 19.3.3.5 PivotForward()

```
void Mark3::CircularLinkList::PivotForward ( )
```

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

Definition at line 135 of file II.cpp.

#### 19.3.3.6 Remove()

Remove Add the linked list node to this linked list.

### Parameters

node⊷	Pointer to the node to remove

Definition at line 103 of file II.cpp.

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

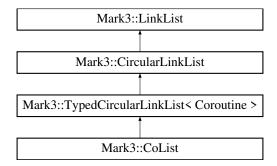
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

### 19.4 Mark3::CoList Class Reference

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

```
#include <colist.h>
```

Inheritance diagram for Mark3::CoList:



#### **Public Member Functions**

void SetPrioMap (CoPrioMap \*pclPrioMap\_)

SetPrioMap Assign a priority map object to this co-routine list.

void SetPriority (PORT\_PRIO\_TYPE uPriority\_)

SetPriority Set the scheduling priority of this coroutine liss; has no effect unless a SetPrioMap has been called with a valid coroutine priority map object.

void Add (Coroutine \*pclCoroutine\_)

Add Add a coroutine object to this list.

void Remove (Coroutine \*pclCoroutine\_)

Remove Remove a given coroutine object from this list.

### **Private Attributes**

- CoPrioMap \* m\_pclPrioMap
- uint8\_t m\_uPriority

#### **Additional Inherited Members**

### 19.4.1 Detailed Description

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

Definition at line 35 of file colist.h.

#### 19.4.2 Member Function Documentation

Add Add a coroutine object to this list.

#### **Parameters**

pcl←	Pointer to the coroutine object to add
Coroutine_	

Definition at line 36 of file colist.cpp.

### 19.4.2.2 Remove()

Remove Remove a given coroutine object from this list.

#### **Parameters**

pcl⊷	Pointer to the coroutine object to remove
Coroutine_	

Definition at line 46 of file colist.cpp.

### 19.4.2.3 SetPrioMap()

SetPrioMap Assign a priority map object to this co-routine list.

#### **Parameters**

pclPrio←	priority map object to assign
Мар_	

Definition at line 24 of file colist.cpp.

# 19.4.2.4 SetPriority()

SetPriority Set the scheduling priority of this coroutine liss; has no effect unless a SetPrioMap has been called with a valid coroutine priority map object.

#### **Parameters**

U←	Priority of coroutines associated with this list
Priority←	
_	

Definition at line 30 of file colist.cpp.

#### 19.4.3 Member Data Documentation

```
19.4.3.1 m_pclPrioMap
```

```
CoPrioMap* Mark3::CoList::m_pclPrioMap [private]
```

Definition at line 73 of file colist.h.

19.4.3.2 m\_uPriority

```
uint8_t Mark3::CoList::m_uPriority [private]
```

Definition at line 74 of file colist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/colist.h
- /home/moslevin/projects/m3-repo/kernel/src/colist.cpp

#### 19.5 Mark3::ConditionVariable Class Reference

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

#include <condvar.h>

#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- void Init ()

Init Initialize the condition variable prior to use. Must be called before the object can be used.

void Wait (Mutex \*pclMutex\_)

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

bool Wait (Mutex \*pclMutex\_, uint32\_t u32WaitTimeMS\_)

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

· void Signal ()

Signal Signal/Unblock the next thread currently blocked on this condition variable.

· void Broadcast ()

Broadcast Unblock all threads currently blocked on this condition variable.

#### **Private Attributes**

- Mutex m clMutex
- · Semaphore m clSemaphore
- uint8\_t m\_u8Waiters

#### 19.5.1 Detailed Description

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

Definition at line 39 of file condvar.h.

# 19.5.2 Member Function Documentation

#### 19.5.2.1 Broadcast()

```
void Mark3::ConditionVariable::Broadcast ( )
```

Broadcast Unblock all threads currently blocked on this condition variable.

Definition at line 77 of file condvar.cpp.

### 19.5.2.2 Init()

Init Initialize the condition variable prior to use. Must be called before the object can be used.

Definition at line 25 of file condvar.cpp.

#### 19.5.2.3 operator new()

```
void* Mark3::ConditionVariable::operator new (  \label{eq:size_tsz}  \mbox{size\_t $sz$,}  void * pv ) [inline]
```

Definition at line 42 of file condvar.h.

### 19.5.2.4 Signal()

```
void Mark3::ConditionVariable::Signal ( )
```

Signal Signal/Unblock the next thread currently blocked on this condition variable.

Definition at line 66 of file condvar.cpp.

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

#### **Parameters**

pcl←	Mutex to claim once the calling thread has access to the condvar
Mutex_	

Definition at line 32 of file condvar.cpp.

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

#### **Parameters**

pclMutex_	Mutex to claim once the calling thread has access to the condvar
u32WaitTimeM⊷	Maximum time in ms to wait before abandoning the operation
S_	

#### Returns

true on success, false on timeout

Definition at line 48 of file condvar.cpp.

#### 19.5.3 Member Data Documentation

#### 19.5.3.1 m\_clMutex

```
Mutex Mark3::ConditionVariable::m_clMutex [private]
```

Definition at line 82 of file condvar.h.

### 19.5.3.2 m\_clSemaphore

```
Semaphore Mark3::ConditionVariable::m_clSemaphore [private]
```

Definition at line 83 of file condvar.h.

### 19.5.3.3 m\_u8Waiters

```
uint8_t Mark3::ConditionVariable::m_u8Waiters [private]
```

Definition at line 84 of file condvar.h.

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

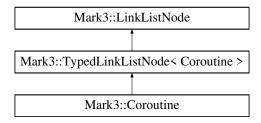
- /home/moslevin/projects/m3-repo/kernel/src/public/condvar.h
- /home/moslevin/projects/m3-repo/kernel/src/condvar.cpp

#### 19.6 Mark3::Coroutine Class Reference

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

```
#include <coroutine.h>
```

Inheritance diagram for Mark3::Coroutine:



#### **Public Member Functions**

- ∼Coroutine ()
- void Init (PORT\_PRIO\_TYPE uPriority\_, CoroutineHandler pfHandler\_, void \*pvContext\_)

Init Initialize the coroutine object prior to use. Must be called before using the other methods in the class.

• void Run ()

Run Clear the co-routine's pending execution flag, and execute the coroutine's handler function.

· void Activate ()

Activate Tag the co-routine as pending execution. Has no effect if the co-routine is already pending execution.

void SetPriority (PORT\_PRIO\_TYPE uPriority\_)

SetPriority Update the scheduling priority of the co-routine. Can be called from within the co-routine, or from any other context aware of the co-routine object.

• PORT\_PRIO\_TYPE GetPriority ()

GetPriority Retrieve the current scheduling priority of the co-routine.

#### **Private Attributes**

- CoList \* m\_pclOwner
- CoroutineHandler m pfHandler
- void \* m pvContext
- PORT\_PRIO\_TYPE m\_uPriority
- bool m bQueued

### **Additional Inherited Members**

### 19.6.1 Detailed Description

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

Co-routines differ from Threads in that they cannot block, and must run to completion before other (potentially higher-priority) co-routines block.

### **Examples:**

lab2\_coroutines/main.cpp.

Definition at line 53 of file coroutine.h.

### 19.6.2 Constructor & Destructor Documentation

```
19.6.2.1 \simCoroutine()
```

Mark3::Coroutine::~Coroutine ( )

Definition at line 28 of file coroutine.cpp.

### 19.6.3 Member Function Documentation

```
19.6.3.1 Activate()
```

```
void Mark3::Coroutine::Activate ( )
```

Activate Tag the co-routine as pending execution. Has no effect if the co-routine is already pending execution.

### **Examples:**

lab2\_coroutines/main.cpp.

Definition at line 67 of file coroutine.cpp.

#### 19.6.3.2 GetPriority()

```
PORT_PRIO_TYPE Mark3::Coroutine::GetPriority ( )
```

GetPriority Retrieve the current scheduling priority of the co-routine.

### Returns

current scheduling priority of the co-routine.

Definition at line 97 of file coroutine.cpp.

### 19.6.3.3 Init()

Init Initialize the coroutine object prior to use. Must be called before using the other methods in the class.

#### **Parameters**

uPriority←	The scheduling priority of this coroutine as configured by the port.
 pf⊷ Handler⊷	Function to run when the coroutine is executed
_	
pv⊷ Context_	User-defined value passed into the handler function on execution

Definition at line 39 of file coroutine.cpp.

```
19.6.3.4 Run()
```

```
void Mark3::Coroutine::Run ( )
```

Run Clear the co-routine's pending execution flag, and execute the coroutine's handler function.

Definition at line 53 of file coroutine.cpp.

### 19.6.3.5 SetPriority()

SetPriority Update the scheduling priority of the co-routine. Can be called from within the co-routine, or from any other context aware of the co-routine object.

### **Parameters**

u⇔ Priority⇔	New priority of the co-routine
Priority←	, ,

Definition at line 82 of file coroutine.cpp.

#### 19.6.4 Member Data Documentation

# 19.6.4.1 m\_bQueued

```
bool Mark3::Coroutine::m_bQueued [private]
```

Definition at line 108 of file coroutine.h.

```
19.6.4.2 m_pclOwner
```

```
CoList* Mark3::Coroutine::m_pclOwner [private]
```

Definition at line 104 of file coroutine.h.

### 19.6.4.3 m\_pfHandler

```
CoroutineHandler Mark3::Coroutine::m_pfHandler [private]
```

Definition at line 105 of file coroutine.h.

### 19.6.4.4 m\_pvContext

```
void* Mark3::Coroutine::m_pvContext [private]
```

Definition at line 106 of file coroutine.h.

### 19.6.4.5 m\_uPriority

```
PORT_PRIO_TYPE Mark3::Coroutine::m_uPriority [private]
```

Definition at line 107 of file coroutine.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h
- /home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp

### 19.7 Mark3::CoScheduler Class Reference

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

```
#include <cosched.h>
```

#### **Static Public Member Functions**

· static void Init ()

Init Initialize the coroutine scheduler prior to use. Must be called prior to using any other functions in the coroutine scheduler.

static CoPrioMap \* GetPrioMap ()

GetPrioMap Get the pointer to the priority map object used by the scheduler.

static CoList \* GetStopList ()

GetStopList Get the pointer to the coroutine list managing initialized coroutines that are not awaiting execution.

static CoList \* GetCoList (PORT\_PRIO\_TYPE uPriority\_)

GetCoList Retrieve the coroutine list associated with a given priority.

• static Coroutine \* Schedule ()

Schedule Determine what coroutine (if any) is the next to be executed.

#### **Static Private Attributes**

- static CoList m\_aclPriorities [PORT\_COROUTINE\_PRIORITIES]
- static CoList m\_clStopList
- static CoPrioMap m\_clPrioMap

### 19.7.1 Detailed Description

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

Definition at line 33 of file cosched.h.

#### 19.7.2 Member Function Documentation

#### 19.7.2.1 GetCoList()

GetCoList Retrieve the coroutine list associated with a given priority.

### **Parameters**

U←	Priority to get the coroutine list of.
Priority⊷	

#### Returns

coroutine list pointer or nullptr on invalid priority.

Definition at line 51 of file cosched.cpp.

### 19.7.2.2 GetPrioMap()

```
CoPrioMap * Mark3::CoScheduler::GetPrioMap ( ) [static]
```

GetPrioMap Get the pointer to the priority map object used by the scheduler.

#### Returns

Return the priority map object owned by the schedule

Definition at line 39 of file cosched.cpp.

## 19.7.2.3 GetStopList()

```
CoList * Mark3::CoScheduler::GetStopList ( ) [static]
```

GetStopList Get the pointer to the coroutine list managing initialized coroutines that are not awaiting execution.

### Returns

Pointer to the coroutine stop list

Definition at line 45 of file cosched.cpp.

# 19.7.2.4 Init()

Init Initialize the coroutine scheduler prior to use. Must be called prior to using any other functions in the coroutine scheduler.

# **Examples:**

lab2\_coroutines/main.cpp.

Definition at line 29 of file cosched.cpp.

## 19.7.2.5 Schedule()

```
Coroutine * Mark3::CoScheduler::Schedule ( ) [static]
```

Schedule Determine what coroutine (if any) is the next to be executed.

Returns

next coroutine to execute, or nullptr if no coroutines are ready to be scheduled.

## **Examples:**

lab2\_coroutines/main.cpp.

Definition at line 60 of file cosched.cpp.

# 19.7.3 Member Data Documentation

```
19.7.3.1 m_aclPriorities
```

```
CoList Mark3::CoScheduler::m_aclPriorities [static], [private]
```

Definition at line 80 of file cosched.h.

```
19.7.3.2 m_clPrioMap
```

```
CoPrioMap Mark3::CoScheduler::m_clPrioMap [static], [private]
```

Definition at line 82 of file cosched.h.

```
19.7.3.3 m_clStopList
```

```
CoList Mark3::CoScheduler::m_clStopList [static], [private]
```

Definition at line 81 of file cosched.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/cosched.h
- /home/moslevin/projects/m3-repo/kernel/src/cosched.cpp

# 19.8 Mark3::CriticalGuard Class Reference

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

```
#include <criticalguard.h>
```

## **Public Member Functions**

- · CriticalGuard ()
- ∼CriticalGuard ()

## 19.8.1 Detailed Description

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

This is similar to the LockGuard class, except that class operates on a single Mutex, and this class operates on a global interrupt-disabled lock.

Definition at line 38 of file criticalguard.h.

## 19.8.2 Constructor & Destructor Documentation

### 19.8.2.1 CriticalGuard()

```
Mark3::CriticalGuard::CriticalGuard ( ) [inline]
```

Definition at line 40 of file criticalguard.h.

## 19.8.2.2 ∼CriticalGuard()

```
Mark3::CriticalGuard::~CriticalGuard ( ) [inline]
```

Definition at line 44 of file criticalguard.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h

# 19.9 Mark3::CriticalSection Class Reference

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

```
#include <criticalsection.h>
```

#### Static Public Member Functions

• static void Enter ()

Enter Enter a critical section, disabling all kernel-aware interrupts, and giving exclusive control of the CPU to the curreninttly running task.

static void Exit ()

Exit Exit a critical section, re-enabling kernel-aware interrupts, and releasing exclusive control of the CPU.

• static K\_WORD NestingCount ()

NestingCount.

# 19.9.1 Detailed Description

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

Critical sections *can* be safely nested, that is, multiple calls to Enter() may be called before matching calls to Exit() are called. In such cases, only the *final* call to Exit() will cause the caller to relinquish control of the CPU.

Critical sections can be safely used within interrupts; although they have no effect in the general case. The exception is in the case where a system has multiple levels of interrupt nesting; at which point calling a critical section from an interrupt temporarily disables subsequent levels of nesting.

Care, however, must be taken to ensure that the currently executing thread does *not* block when a critical section is active. This condition is guaranteed to break the system.

Definition at line 48 of file critical section.h.

# 19.9.2 Member Function Documentation

```
19.9.2.1 Enter()
```

```
static void Mark3::CriticalSection::Enter ( ) [inline], [static]
```

Enter Enter a critical section, disabling all kernel-aware interrupts, and giving exclusive control of the CPU to the curreninttly running task.

# **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 56 of file criticalsection.h.

## 19.9.2.2 Exit()

```
static void Mark3::CriticalSection::Exit ( ) [inline], [static]
```

Exit Exit a critical section, re-enabling kernel-aware interrupts, and releasing exclusive control of the CPU.

### **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 65 of file criticalsection.h.

#### 19.9.2.3 NestingCount()

```
static K_WORD Mark3::CriticalSection::NestingCount ( ) [inline], [static]
```

NestingCount.

#### Returns

Number of Enter() calls awaiting an Exit() call.

Definition at line 73 of file criticalsection.h.

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

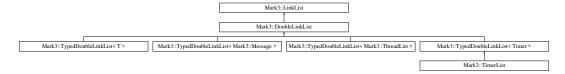
• /home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h

# 19.10 Mark3::DoubleLinkList Class Reference

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

```
#include <11.h>
```

Inheritance diagram for Mark3::DoubleLinkList:



## **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- DoubleLinkList ()

DoubleLinkList.

void Add (LinkListNode \*node\_)

Add

void Remove (LinkListNode \*node\_)

Remove.

# **Additional Inherited Members**

# 19.10.1 Detailed Description

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

Definition at line 179 of file II.h.

# 19.10.2 Constructor & Destructor Documentation

# 19.10.2.1 DoubleLinkList()

```
Mark3::DoubleLinkList::DoubleLinkList ( ) [inline]
```

# DoubleLinkList.

Default constructor - initializes the head/tail nodes to nullptr

Definition at line 188 of file II.h.

# 19.10.3 Member Function Documentation

# 19.10.3.1 Add()

Add.

Add the linked list node to this linked list

# **Parameters**

node⊷	Pointer to the node to add

Definition at line 34 of file II.cpp.

## 19.10.3.2 operator new()

```
void* Mark3::DoubleLinkList::operator new (
```

```
size_t sz,
void * pv ) [inline]
```

Definition at line 182 of file II.h.

## 19.10.3.3 Remove()

Remove.

Add the linked list node to this linked list

#### **Parameters**

node←	Pointer to the node to remove

Definition at line 55 of file II.cpp.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

# 19.11 Mark3::EventFlag Class Reference

The EventFlag class. This class implements 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.

```
#include <eventflag.h>
```

Inheritance diagram for Mark3::EventFlag:



## **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ∼EventFlag ()
- void Init ()

Init Initializes the EventFlag object prior to use.

uint16\_t Wait (uint16\_t u16Mask\_, EventFlagOperation eMode\_)

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

• uint16 t Wait (uint16 t u16Mask , EventFlagOperation eMode , uint32 t u32TimeMS )

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

void WakeMe (Thread \*pclChosenOne )

WakeMe Wake the given thread, currently blocking on this object.

void Set (uint16\_t u16Mask\_)

Set Set additional flags in this object (logical OR). This API can potentially result in threads blocked on Wait() to be unblocked.

void Clear (uint16\_t u16Mask\_)

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

uint16\_t GetMask ()

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

#### **Private Member Functions**

• uint16\_t Wait\_i (uint16\_t u16Mask\_, EventFlagOperation eMode\_, uint32\_t u32TimeMS\_)

Wait\_i Interal abstraction used to manage both timed and untimed wait operations.

## **Private Attributes**

uint16\_t m\_u16SetMask

Event flags currently set in this object.

## **Additional Inherited Members**

# 19.11.1 Detailed Description

The EventFlag class. This class implements a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system. Each EventFlag object contains a 16-bit bitmask, which is used to trigger events on associated threads. Threads wishing to block, waiting for a specific event to occur can wait on any pattern within this 16-bit bitmask to be set. Here, we provide the ability for a thread to block, waiting for ANY bits in a specified mask to be set, or for ALL bits within a specific mask to be set. Depending on how the object is configured, the bits that triggered the wakeup can be automatically cleared once a match has occurred.

## **Examples:**

lab7\_events/main.cpp.

Definition at line 45 of file eventflag.h.

# 19.11.2 Constructor & Destructor Documentation

```
19.11.2.1 ~EventFlag()

Mark3::EventFlag::~EventFlag ( )
```

# 19.11.3 Member Function Documentation

# 19.11.3.1 Clear()

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

## **Parameters**

u16⇔	- Bitmask of flags to clear
Mask_	

# Examples:

lab7\_events/main.cpp.

## 19.11.3.2 GetMask()

```
uint16_t Mark3::EventFlag::GetMask ( )
```

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

# Returns

The state of the 16-bit bitmask

# 19.11.3.3 Init()

```
void Mark3::EventFlag::Init ( )
```

Init Initializes the EventFlag object prior to use.

# 19.11.3.4 operator new()

Definition at line 48 of file eventflag.h.

# 19.11.3.5 Set()

Set Set additional flags in this object (logical OR). This API can potentially result in threads blocked on Wait() to be unblocked.

## **Parameters**

<i>u</i> 16⇔	- Bitmask of flags to set.
Mask_	

# **Examples:**

lab7\_events/main.cpp.

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

EventFlagOperation eMode\_ )

## **Parameters**

u16⇔	- 16-bit bitmask to block on
Mask_	
eMode_	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask
	EventFlagOperation::All_Set: Thread will block on all of the bits in the mask

## Returns

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

# **Examples:**

lab7\_events/main.cpp.

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

## **Parameters**

u16Mask_	- 16-bit bitmask to block on
eMode_	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask
	EventFlagOperation::All_Set: Thread will block on all of the bits in the mask
u32TimeM⊷ S_	- Time to block (in ms)

## Returns

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

# 19.11.3.8 Wait\_i()

Wait\_i Interal abstraction used to manage both timed and untimed wait operations.

# **Parameters**

u16Mask_	- 16-bit bitmask to block on
eMode_	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask
	EventFlagOperation::All_Set: Thread will block on all of the bits in the mask
u32TimeM⊷ S_	- Time to block (in ms)

## Returns

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

# 19.11.3.9 WakeMe()

WakeMe Wake the given thread, currently blocking on this object.

## **Parameters**

pclChosen←	Pointer to the owner thread to unblock.
One_	

# 19.11.4 Member Data Documentation

## 19.11.4.1 m\_u16SetMask

```
uint16_t Mark3::EventFlag::m_u16SetMask [private]
```

Event flags currently set in this object.

Definition at line 119 of file eventflag.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/eventflag.h

# 19.12 Mark3::Kernel Class Reference

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

```
#include <kernel.h>
```

### **Static Public Member Functions**

· static void Init ()

Kernel Initialization Function, call before any other OS function.

static void Start ()

Start the operating system kernel - the current execution context is cancelled, all kernel services are started, and the processor resumes execution at the entrypoint for the highest-priority thread.

static void CompleteStart ()

CompleteStart Call this from the thread initialization code at the point that the scheduler is to be run for the first time.

• static bool IsStarted ()

IsStarted.

static void SetPanic (PanicFunc pfPanic\_)

SetPanic Set a function to be called when a kernel panic occurs, giving the user to determine the behavior when a catastrophic failure is observed.

• static bool IsPanic ()

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

static void Panic (uint16\_t u16Cause\_)

Panic Cause the kernel to enter its panic state.

static void SetThreadCreateCallout (ThreadCreateCallout pfCreate )

SetThreadCreateCallout Set a function to be called on creation of a new thread. This callout is executed on the successful completion of a Thread::Init() call. A callout is only executed if this method has been called to set a valid handler function.

static void SetThreadExitCallout (ThreadExitCallout pfExit\_)

SetThreadExitCallout Set a function to be called on thread exit. This callout is executed from the beginning of Thread::Exit().

static void SetThreadContextSwitchCallout (ThreadContextCallout pfContext\_)

SetThreadContextSwitchCallout Set a function to be called on each context switch.

• static void SetDebugPrintFunction (DebugPrintFunction pfPrintFunction\_)

SetDebugPrintFunction Set the function to be used when printing kernel debug information.

static void DebugPrint (const char \*szString\_)

DebugPrint Print a string to the configured output interface. Has no effect if Kernel::SetDebugPrintFunction() has not been called with a valid print handler.

static ThreadCreateCallout GetThreadCreateCallout ()

GetThreadCreateCallout Return the current function called on every Thread::Init();.

static ThreadExitCallout GetThreadExitCallout ()

GetThreadExitCallout Return the current function called on every Thread::Exit();.

static ThreadContextCallout GetThreadContextSwitchCallout ()

GetThreadContextSwitchCallout Return the current function called on every Thread::ContextSwitchSWI()

- static void SetStackGuardThreshold (uint16\_t u16Threshold\_)
- static uint16\_t GetStackGuardThreshold ()
- · static void Tick ()
- static uint32\_t GetTicks ()

## **Static Private Attributes**

static bool m blsStarted

true if kernel is running, false otherwise

static bool m\_blsPanic

true if kernel is in panic state, false otherwise

· static PanicFunc m pfPanic

set panic function

• static ThreadCreateCallout m\_pfThreadCreateCallout

Function to call on thread creation.

static ThreadExitCallout m\_pfThreadExitCallout

Function to call on thread exit.

static ThreadContextCallout m\_pfThreadContextCallout

Function to call on context switch.

• static DebugPrintFunction m\_pfDebugPrintFunction

Function to call to print debug info.

- static uint16 t m u16GuardThreshold
- static uint32\_t m\_u32Ticks

# 19.12.1 Detailed Description

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

Definition at line 48 of file kernel.h.

### 19.12.2 Member Function Documentation

## 19.12.2.1 CompleteStart()

```
void Mark3::Kernel::CompleteStart ( ) [static]
```

CompleteStart Call this from the thread initialization code at the point that the scheduler is to be run for the first time.

Definition at line 64 of file kernel.cpp.

## 19.12.2.2 DebugPrint()

DebugPrint Print a string to the configured output interface. Has no effect if Kernel::SetDebugPrintFunction() has not been called with a valid print handler.

### **Parameters**

SZ⊷	string to print
String_	

#### **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 81 of file kernel.cpp.

# 19.12.2.3 GetStackGuardThreshold()

```
static uint16_t Mark3::Kernel::GetStackGuardThreshold ( ) [inline], [static]
```

Definition at line 202 of file kernel.h.

## 19.12.2.4 GetThreadContextSwitchCallout()

```
static ThreadContextCallout Mark3::Kernel::GetThreadContextSwitchCallout ( ) [inline], [static]
```

GetThreadContextSwitchCallout Return the current function called on every Thread::ContextSwitchSWI()

## Returns

Pointer to the currently-installed callout function, or nullptr if not set.

Definition at line 198 of file kernel.h.

# 19.12.2.5 GetThreadCreateCallout()

```
static ThreadCreateCallout Mark3::Kernel::GetThreadCreateCallout ( ) [inline], [static]
```

GetThreadCreateCallout Return the current function called on every Thread::Init();.

## Returns

Pointer to the currently-installed callout function, or nullptr if not set.

Definition at line 178 of file kernel.h.

## 19.12.2.6 GetThreadExitCallout()

```
static ThreadExitCallout Mark3::Kernel::GetThreadExitCallout ( ) [inline], [static]
```

GetThreadExitCallout Return the current function called on every Thread::Exit();.

# Returns

Pointer to the currently-installed callout function, or nullptr if not set.

Definition at line 188 of file kernel.h.

```
19.12.2.7 GetTicks()
uint32_t Mark3::Kernel::GetTicks ( ) [static]
```

lab9\_dynamic\_threads/main.cpp.

Definition at line 90 of file kernel.cpp.

Kernel Initialization Function, call before any other OS function.

Initializes all global resources used by the operating system. This must be called before any other kernel function is invoked.

**Examples:** 

**Examples:** 

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.⇔ cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.⇔ cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 44 of file kernel.cpp.

```
19.12.2.9 IsPanic()
static bool Mark3::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 99 of file kernel.h.

```
19.12.2.10 IsStarted()
static bool Mark3::Kernel::IsStarted ( ) [inline], [static]
IsStarted.
Returns
```

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

Definition at line 86 of file kernel.h.

Panic Cause the kernel to enter its panic state.

#### **Parameters**

u16 <b></b> ⇔	Reason for the kernel panic
Cause_	

Definition at line 70 of file kernel.cpp.

## 19.12.2.12 SetDebugPrintFunction()

SetDebugPrintFunction Set the function to be used when printing kernel debug information.

## **Parameters**

pfPrint←	Function used to print kernel debug message strings
Function_	

## **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 155 of file kernel.h.

## 19.12.2.13 SetPanic()

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**

pf←	Panic function pointer
Panic⊷	

Definition at line 94 of file kernel.h.

## 19.12.2.14 SetStackGuardThreshold()

Definition at line 201 of file kernel.h.

# 19.12.2.15 SetThreadContextSwitchCallout()

SetThreadContextSwitchCallout Set a function to be called on each context switch.

A callout is only executed if this method has been called to set a valid handler function.

#### **Parameters**

pf⇔	Pointer to a function to call on context switch
Context←	

## **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 143 of file kernel.h.

# 19.12.2.16 SetThreadCreateCallout()

SetThreadCreateCallout Set a function to be called on creation of a new thread. This callout is executed on the successful completion of a Thread::Init() call. A callout is only executed if this method has been called to set a valid handler function.

#### **Parameters**

pf⇔	Pointer to a function to call on thread creation
Create←	
1_	

# Examples:

lab9\_dynamic\_threads/main.cpp.

Definition at line 116 of file kernel.h.

#### 19.12.2.17 SetThreadExitCallout()

SetThreadExitCallout Set a function to be called on thread exit. This callout is executed from the beginning of Thread::Exit().

A callout is only executed if this method has been called to set a valid handler function.

#### **Parameters**

pf⇔	Pointer to a function to call on thread exit
Exit←	
_	

#### **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 130 of file kernel.h.

## 19.12.2.18 Start()

```
void Mark3::Kernel::Start ( ) [static]
```

Start the operating system kernel - the current execution context is cancelled, all kernel services are started, and the processor resumes execution at the entrypoint for the highest-priority thread.

You must have at least one thread added to the kernel before calling this function, otherwise the behavior is undefined. The exception to this is if the system is configured to use the threadless idle hook, in which case the kernel is allowed to run without any ready threads.

## **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 58 of file kernel.cpp.

```
19.12.2.19 Tick()
static void Mark3::Kernel::Tick ( ) [inline], [static]
Definition at line 205 of file kernel.h.
19.12.3 Member Data Documentation
19.12.3.1 m_blsPanic
bool Mark3::Kernel::m_bIsPanic [static], [private]
true if kernel is in panic state, false otherwise
Definition at line 210 of file kernel.h.
19.12.3.2 m_blsStarted
bool Mark3::Kernel::m_bIsStarted [static], [private]
true if kernel is running, false otherwise
Definition at line 209 of file kernel.h.
19.12.3.3 m_pfDebugPrintFunction
DebugPrintFunction Mark3::Kernel::m_pfDebugPrintFunction [static], [private]
Function to call to print debug info.
Function to call when printing debug info.
Definition at line 222 of file kernel.h.
19.12.3.4 m_pfPanic
PanicFunc Mark3::Kernel::m_pfPanic [static], [private]
set panic function
```

Definition at line 211 of file kernel.h.

## 19.12.3.5 m\_pfThreadContextCallout

ThreadContextCallout Mark3::Kernel::m\_pfThreadContextCallout [static], [private]

Function to call on context switch.

Definition at line 220 of file kernel.h.

# 19.12.3.6 m\_pfThreadCreateCallout

ThreadCreateCallout Mark3::Kernel::m\_pfThreadCreateCallout [static], [private]

Function to call on thread creation.

Definition at line 214 of file kernel.h.

# 19.12.3.7 m\_pfThreadExitCallout

ThreadExitCallout Mark3::Kernel::m\_pfThreadExitCallout [static], [private]

Function to call on thread exit.

Definition at line 217 of file kernel.h.

# 19.12.3.8 m\_u16GuardThreshold

```
uint16_t Mark3::Kernel::m_u16GuardThreshold [static], [private]
```

Definition at line 224 of file kernel.h.

# 19.12.3.9 m\_u32Ticks

```
uint32_t Mark3::Kernel::m_u32Ticks [static], [private]
```

Definition at line 226 of file kernel.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/kernel.h
- /home/moslevin/projects/m3-repo/kernel/src/kernel.cpp

# 19.13 Mark3::KernelSWI Class Reference

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

```
#include <kernelswi.h>
```

#### **Static Public Member Functions**

static void Config (void)

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

static void Start (void)

Start Enable ("Start") the software interrupt functionality.

• static void Trigger (void)

Trigger Call the software interrupt.

# 19.13.1 Detailed Description

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

Definition at line 32 of file kernelswi.h.

# 19.13.2 Member Function Documentation

# 19.13.2.1 Config()

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

Definition at line 31 of file kernelswi.cpp.

```
19.13.2.2 Start()
```

Start Enable ("Start") the software interrupt functionality.

Definition at line 39 of file kernelswi.cpp.

## 19.13.2.3 Trigger()

Trigger Call the software interrupt.

Definition at line 46 of file kernelswi.cpp.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp

# 19.14 Mark3::KernelTimer Class Reference

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

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

# **Static Public Member Functions**

• static void Config (void)

Config Initializes the kernel timer before use.

• static void Start (void)

Start Starts the kernel time (must be configured first)

static void Stop (void)

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

# 19.14.1 Detailed Description

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

Definition at line 34 of file kerneltimer.h.

# 19.14.2 Member Function Documentation

# 19.14.2.1 Config()

Config Initializes the kernel timer before use.

Definition at line 60 of file kerneltimer.cpp.

## 19.14.2.2 Start()

Start Starts the kernel time (must be configured first)

Definition at line 76 of file kerneltimer.cpp.

#### 19.14.2.3 Stop()

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

Definition at line 86 of file kerneltimer.cpp.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp

# 19.15 Mark3::LinkList Class Reference

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

```
#include <11.h>
```

Inheritance diagram for Mark3::LinkList:



# **Public Member Functions**

```
• void Init ()
```

Init.

LinkListNode \* GetHead ()

GetHead.

void SetHead (LinkListNode \*pclNode\_)

SetHead.

• LinkListNode \* GetTail ()

GetTail

void SetTail (LinkListNode \*pclNode\_)

SetTail.

# **Protected Attributes**

LinkListNode \* m\_pclHead

Pointer to the head node in the list.

• LinkListNode \* m\_pclTail

Pointer to the tail node in the list.

# 19.15.1 Detailed Description

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

Definition at line 119 of file II.h.

## 19.15.2 Member Function Documentation

```
19.15.2.1 GetHead()
```

```
LinkListNode* Mark3::LinkList::GetHead ( ) [inline]
```

GetHead.

Get the head node in the linked list

Returns

Pointer to the head node in the list

Definition at line 144 of file II.h.

```
19.15.2.2 GetTail()
```

```
LinkListNode* Mark3::LinkList::GetTail ( ) [inline]
```

GetTail.

Get the tail node of the linked list

Returns

Pointer to the tail node in the list

Definition at line 162 of file II.h.

# 19.15.2.3 Init()

Init.

Clear the linked list.

Definition at line 131 of file II.h.

# 19.15.2.4 SetHead()

SetHead.

Set the head node of a linked list

## **Parameters**

pcl⇔	Pointer to node to set as the head of the linked list	
Node_		

Definition at line 153 of file II.h.

# 19.15.2.5 SetTail()

SetTail.

Set the tail node of the linked list

## **Parameters**

pcl←	Pointer to the node to set as the tail of the linked list
Node_	

Definition at line 171 of file II.h.

# 19.15.3 Member Data Documentation

```
19.15.3.1 m_pclHead
```

```
LinkListNode* Mark3::LinkList::m_pclHead [protected]
```

Pointer to the head node in the list.

Definition at line 122 of file II.h.

19.15.3.2 m\_pclTail

```
LinkListNode* Mark3::LinkList::m_pclTail [protected]
```

Pointer to the tail node in the list.

Definition at line 123 of file II.h.

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

/home/moslevin/projects/m3-repo/kernel/src/public/II.h

# 19.16 Mark3::LinkListNode Class Reference

The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

```
#include <11.h>
```

Inheritance diagram for Mark3::LinkListNode:



# **Public Member Functions**

LinkListNode \* GetNext (void)

GetNext.

LinkListNode \* GetPrev (void)

GetPrev.

# **Protected Member Functions**

- · LinkListNode ()
- void ClearNode ()

ClearNode.

# **Protected Attributes**

LinkListNode \* next

Pointer to the next node in the list.

LinkListNode \* prev

Pointer to the previous node in the list.

# **Friends**

- class LinkList
- class DoubleLinkList
- · class CircularLinkList

# 19.16.1 Detailed Description

The LinkListNode Class 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 62 of file II.h.

# 19.16.2 Constructor & Destructor Documentation

# 19.16.2.1 LinkListNode()

```
Mark3::LinkListNode::LinkListNode ( ) [inline], [protected]
```

Definition at line 68 of file II.h.

# 19.16.3 Member Function Documentation

# 19.16.3.1 ClearNode()

```
void Mark3::LinkListNode::ClearNode ( ) [protected]
```

ClearNode.

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

Definition at line 27 of file II.cpp.

```
19.16.3.2 GetNext()
```

GetNext.

Returns a pointer to the next node in the list.

Returns

a pointer to the next node in the list.

Definition at line 85 of file II.h.

# 19.16.3.3 GetPrev()

GetPrev.

Returns a pointer to the previous node in the list.

Returns

a pointer to the previous node in the list.

Definition at line 93 of file II.h.

# 19.16.4 Friends And Related Function Documentation

# 19.16.4.1 CircularLinkList

```
friend class CircularLinkList [friend]
```

Definition at line 96 of file II.h.

# 19.16.4.2 DoubleLinkList

```
friend class DoubleLinkList [friend]
```

Definition at line 95 of file II.h.

## 19.16.4.3 LinkList

```
friend class LinkList [friend]
```

Definition at line 94 of file II.h.

## 19.16.5 Member Data Documentation

```
19.16.5.1 next
```

```
LinkListNode* Mark3::LinkListNode::next [protected]
```

Pointer to the next node in the list.

Definition at line 65 of file II.h.

19.16.5.2 prev

```
LinkListNode* Mark3::LinkListNode::prev [protected]
```

Pointer to the previous node in the list.

Definition at line 66 of file II.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

# 19.17 Mark3::LockGuard Class Reference

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

```
#include <lockguard.h>
```

## **Public Member Functions**

- LockGuard (Mutex \*pclMutex)
- LockGuard (Mutex \*pclMutex, uint32\_t u32TimeoutMs\_)
- ∼LockGuard ()
- bool isAcquired ()

# **Private Attributes**

- bool m\_blsAcquired
- Mutex \* m\_pclMutex

# 19.17.1 Detailed Description

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

Definition at line 32 of file lockguard.h.

# 19.17.2 Constructor & Destructor Documentation

#### **Parameters**

	pclMutex	mutex to lock during construction
--	----------	-----------------------------------

Definition at line 25 of file lockguard.cpp.

```
19.17.2.2 LockGuard() [2/2]
```

# **Parameters**

pclMutex	mutex to lock during construction	
u32Timeout⊷	timeout (in ms) to wait before bailng	
Ms_		

Definition at line 34 of file lockguard.cpp.

## 19.17.2.3 $\sim$ LockGuard()

```
Mark3::LockGuard::~LockGuard ( )
```

Definition at line 42 of file lockguard.cpp.

## 19.17.3 Member Function Documentation

## 19.17.3.1 isAcquired()

```
bool Mark3::LockGuard::isAcquired ( ) [inline]
```

Verify that lock was correctly initialized and locked during acquisition. This is used to provide error-checking for timed RAII locks, where Mark3 does not use exceptions, and a kernel-panic is too heavy-handed.

#### Returns

true if the lock was initialed correctly, false on error

Definition at line 54 of file lockguard.h.

# 19.17.4 Member Data Documentation

# 19.17.4.1 m\_blsAcquired

```
bool Mark3::LockGuard::m_bIsAcquired [private]
```

Definition at line 57 of file lockguard.h.

# 19.17.4.2 m\_pclMutex

```
Mutex* Mark3::LockGuard::m_pclMutex [private]
```

Definition at line 58 of file lockguard.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h
- /home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp

## 19.18 Mark3::Mailbox Class Reference

The Mailbox class. This class implements an IPC mechnism based on sending/receiving envelopes containing data of a fixed size, configured at initialization) that reside within a buffer of memory provided by the user.

```
#include <mailbox.h>
```

## **Public Member Functions**

- void \* operator new (size t sz, void \*pv)
- ∼Mailbox ()
- void Init (void \*pvBuffer\_, uint16\_t u16BufferSize\_, uint16\_t u16ElementSize\_)

Init Initialize the mailbox object prior to its use. This must be called before any calls can be made to the object.

bool Send (void \*pvData )

Send Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

bool SendTail (void \*pvData\_)

SendTail Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

bool Send (void \*pvData\_, uint32\_t u32TimeoutMS\_)

Send Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

bool SendTail (void \*pvData\_, uint32\_t u32TimeoutMS\_)

SendTail Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

void Receive (void \*pvData )

Receive Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

void ReceiveTail (void \*pvData )

ReceiveTail Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

bool Receive (void \*pvData\_, uint32\_t u32TimeoutMS\_)

Receive Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

bool ReceiveTail (void \*pvData\_, uint32\_t u32TimeoutMS\_)

ReceiveTail Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

- uint16\_t GetFreeSlots (void)
- bool IsFull (void)
- bool IsEmpty (void)

## **Static Public Member Functions**

static Mailbox \* Init (uint16\_t u16BufferSize\_, uint16\_t u16ElementSize\_)

Init Create and initialize the mailbox object prior to its use. This must be called before any calls can be made to the object. This version of the API alloctes the buffer space from the kernel's Auto-Allocation heap, which cannot be returned back. As a result, this is only suitable for cases where the mailbox will be created once on startup, and persist for the duration of the system.

#### **Private Member Functions**

void \* GetHeadPointer (void)

GetHeadPointer Return a pointer to the current head of the mailbox's internal circular buffer.

void \* GetTailPointer (void)

GetTailPointer Return a pointer to the current tail of the mailbox's internal circular buffer.

void CopyData (const void \*src\_, void \*dst\_, uint16\_t len\_)

CopyData Perform a direct byte-copy from a source to a destination object.

void MoveTailForward (void)

Move Tail Forward Move the tail index forward one element.

void MoveHeadForward (void)

MoveHeadForward Move the head index forward one element.

void MoveTailBackward (void)

MoveTailBackward Move the tail index backward one element.

void MoveHeadBackward (void)

MoveHeadBackward Move the head index backward one element.

bool Send\_i (const void \*pvData\_, bool bTail\_, uint32\_t u32TimeoutMS\_)

Send\_i Internal method which implements all Send() methods in the class.

bool Receive\_i (void \*pvData\_, bool bTail\_, uint32\_t u32WaitTimeMS\_)

Receive\_i Internal method which implements all Read() methods in the class.

#### **Private Attributes**

· uint16\_t m\_u16Head

Current head index.

• uint16\_t m\_u16Tail

Current tail index.

• uint16 t m u16Count

Count of items in the mailbox.

• volatile uint16 t m u16Free

Current number of free slots in the mailbox.

• uint16\_t m\_u16ElementSize

Size of the objects tracked in this mailbox.

• const void \* m\_pvBuffer

Pointer to the data-buffer managed by this mailbox.

• Semaphore m\_clRecvSem

Counting semaphore used to synchronize threads on the object.

· Semaphore m clSendSem

Binary semaphore for send-blocked threads.

# 19.18.1 Detailed Description

The Mailbox class. This class implements an IPC mechnism based on sending/receiving envelopes containing data of a fixed size, configured at initialization) that reside within a buffer of memory provided by the user.

### **Examples:**

lab11\_mailboxes/main.cpp.

Definition at line 35 of file mailbox.h.

# 19.18.2 Constructor & Destructor Documentation

```
19.18.2.1 \sim Mailbox()

Mark3::Mailbox::\sim Mailbox ( )
```

Definition at line 25 of file mailbox.cpp.

# 19.18.3 Member Function Documentation

# 19.18.3.1 CopyData()

CopyData Perform a direct byte-copy from a source to a destination object.

## **Parameters**

<i>src</i> ⊷ –	Pointer to an object to read from
dst⊷ –	Pointer to an object to write to
len⊷ _	Length to copy (in bytes)

Definition at line 216 of file mailbox.h.

# 19.18.3.2 GetFreeSlots()

Definition at line 170 of file mailbox.h.

## 19.18.3.3 GetHeadPointer()

GetHeadPointer Return a pointer to the current head of the mailbox's internal circular buffer.

## Returns

pointer to the head element in the mailbox

Definition at line 187 of file mailbox.h.

# 19.18.3.4 GetTailPointer()

GetTailPointer Return a pointer to the current tail of the mailbox's internal circular buffer.

## Returns

pointer to the tail element in the mailbox

Definition at line 201 of file mailbox.h.

Init Initialize the mailbox object prior to its use. This must be called before any calls can be made to the object.

## **Parameters**

pvBuffer_	Pointer to the static buffer to use for the mailbox
u16BufferSize_	Size of the mailbox buffer, in bytes
u16Element⊷	Size of each envelope, in bytes
Size_	

Definition at line 34 of file mailbox.cpp.

Init Create and initialize the mailbox object prior to its use. This must be called before any calls can be made to the object. This version of the API alloctes the buffer space from the kernel's Auto-Allocation heap, which cannot be returned back. As a result, this is only suitable for cases where the mailbox will be created once on startup, and persist for the duration of the system.

#### **Parameters**

u16BufferSize_	Size of the mailbox buffer, in bytes
u16Element←	Size of each envelope, in bytes
Size_	

Definition at line 59 of file mailbox.cpp.

### 19.18.3.7 IsEmpty()

Definition at line 177 of file mailbox.h.

# 19.18.3.8 IsFull()

Definition at line 176 of file mailbox.h.

### 19.18.3.9 MoveHeadBackward()

MoveHeadBackward Move the head index backward one element.

Definition at line 263 of file mailbox.h.

### 19.18.3.10 MoveHeadForward()

MoveHeadForward Move the head index forward one element.

Definition at line 239 of file mailbox.h.

### 19.18.3.11 MoveTailBackward()

Move TailBackward Move the tail index backward one element.

Definition at line 251 of file mailbox.h.

### 19.18.3.12 MoveTailForward()

MoveTailForward Move the tail index forward one element.

Definition at line 227 of file mailbox.h.

### 19.18.3.13 operator new()

Definition at line 38 of file mailbox.h.

```
19.18.3.14 Receive() [1/2]
```

Receive Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

### **Parameters**

pv⊷	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
Data_	

### **Examples:**

lab11\_mailboxes/main.cpp.

Definition at line 83 of file mailbox.cpp.

```
19.18.3.15 Receive() [2/2]
```

Receive Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

### **Parameters**

pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM⊷	Maximum time to wait for delivery.
S	

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 90 of file mailbox.cpp.

### 19.18.3.16 Receive\_i()

Receive\_i Internal method which implements all Read() methods in the class.

#### **Parameters**

- read from tail, false - read from head
e to wait before timeout (in ms).

Generated by Doxygen

### Returns

true - read successfully, false - timeout.

Definition at line 197 of file mailbox.cpp.

ReceiveTail Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

#### **Parameters**

pv⊷	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
Data_	

Definition at line 97 of file mailbox.cpp.

ReceiveTail Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

### **Parameters**

pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM⊷	Maximum time to wait for delivery.
<i>S</i> _	

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 104 of file mailbox.cpp.

Send Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

### **Parameters**

pv⊷	Pointer to the data object to send to the mailbox.
Data_	

### Returns

true - envelope was delivered, false - mailbox is full.

#### **Examples:**

lab11\_mailboxes/main.cpp.

Definition at line 111 of file mailbox.cpp.

Send Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

uint32\_t u32TimeoutMS\_ )

#### **Parameters**

pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM↔	Maximum time to wait for a free transmit slot
<u>S_</u>	

## Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 125 of file mailbox.cpp.

### 19.18.3.21 Send\_i()

Send\_i Internal method which implements all Send() methods in the class.

### **Parameters**

pvData_	Pointer to the envelope data
bTail_	true - write to tail, false - write to head
u32TimeoutM⊷	Time to wait before timeout (in ms).
S_	

### Returns

true - data successfully written, false - buffer full

Definition at line 139 of file mailbox.cpp.

SendTail Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

## **Parameters**

pv⊷	Pointer to the data object to send to the mailbox.
Data_	

### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 118 of file mailbox.cpp.

### 19.18.3.23 SendTail() [2/2]

SendTail Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

#### **Parameters**

pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM⊷	Maximum time to wait for a free transmit slot
S_	

### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 132 of file mailbox.cpp.

## 19.18.4 Member Data Documentation

## 19.18.4.1 m\_clRecvSem

```
Semaphore Mark3::Mailbox::m_clRecvSem [private]
```

Counting semaphore used to synchronize threads on the object.

Definition at line 302 of file mailbox.h.

## 19.18.4.2 m\_clSendSem

```
Semaphore Mark3::Mailbox::m_clSendSem [private]
```

Binary semaphore for send-blocked threads.

Definition at line 303 of file mailbox.h.

```
19.18.4.3 m_pvBuffer
const void* Mark3::Mailbox::m_pvBuffer [private]
Pointer to the data-buffer managed by this mailbox.
Definition at line 300 of file mailbox.h.
19.18.4.4 m_u16Count
uint16_t Mark3::Mailbox::m_u16Count [private]
Count of items in the mailbox.
Definition at line 296 of file mailbox.h.
19.18.4.5 m_u16ElementSize
uint16_t Mark3::Mailbox::m_u16ElementSize [private]
Size of the objects tracked in this mailbox.
Definition at line 299 of file mailbox.h.
19.18.4.6 m_u16Free
volatile uint16_t Mark3::Mailbox::m_u16Free [private]
Current number of free slots in the mailbox.
Definition at line 297 of file mailbox.h.
19.18.4.7 m_u16Head
```

uint16\_t Mark3::Mailbox::m\_u16Head [private]

Current head index.

Definition at line 293 of file mailbox.h.

Generated by Doxygen

```
19.18.4.8 m_u16Tail
```

```
uint16_t Mark3::Mailbox::m_u16Tail [private]
```

Current tail index.

Definition at line 294 of file mailbox.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h
- /home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp

## 19.19 Mark3::MemUtil Class Reference

String and Memory manipu32ation class.

```
#include <memutil.h>
```

### **Static Public Member Functions**

- static void DecimalToHex (uint8\_t u8Data\_, char \*szText\_)
   DecimalToHex.
- static void DecimalToHex (uint16\_t u16Data\_, char \*szText\_)
- static void DecimalToHex (uint32\_t u32Data\_, char \*szText\_)
- static void DecimalToHex (uint64\_t u64Data\_, char \*szText\_)
- static void DecimalToString (uint8\_t u8Data\_, char \*szText\_)

DecimalToString.

- static void DecimalToString (uint16\_t u16Data\_, char \*szText\_)
- static void DecimalToString (uint32\_t u32Data\_, char \*szText\_)
- static void DecimalToString (uint64 t u64Data , char \*szText )
- static bool StringToDecimal8 (const char \*szText\_, uint8\_t \*pu8Out\_)

StringToDecimial8.

- static bool StringToDecimal16 (const char \*szText\_, uint16\_t \*pu16Out\_)
- static bool StringToDecimal32 (const char \*szText\_, uint32\_t \*pu32Out\_)
- static bool StringToDecimal64 (const char \*szText\_, uint64\_t \*pu64Out\_)
- static uint8\_t Checksum8 (const void \*pvSrc\_, uint16\_t u16Len\_)

Checksum8.

static uint16\_t Checksum16 (const void \*pvSrc\_, uint16\_t u16Len\_)

Checksum16.

static uint16\_t StringLength (const char \*szStr\_)

StringLength.

static bool CompareStrings (const char \*szStr1\_, const char \*szStr2\_)

CompareStrings

- static bool CompareStrings (const char \*szStr1\_, const char \*szStr2\_, uint16\_t u16Length\_)
- static void CopyMemory (void \*pvDst\_, const void \*pvSrc\_, uint16\_t u16Len\_)

CopyMemory.

static void CopyString (char \*szDst , const char \*szSrc )

CopyString.

• static int16\_t StringSearch (const char \*szBuffer\_, const char \*szPattern\_)

StringSearch.

static bool CompareMemory (const void \*pvMem1\_, const void \*pvMem2\_, uint16\_t u16Len\_)
 CompareMemory.

- static void SetMemory (void \*pvDst\_, uint8\_t u8Val\_, uint16\_t u16Len\_)
   SetMemory.
- static uint8\_t Tokenize (const char \*szBuffer\_, Token\_t \*pastTokens\_, uint8\_t u8MaxTokens\_)

Tokenize Function to tokenize a string based on a space delimeter. This is a non-destructive function, which popu32ates a Token\_t descriptor array.

## 19.19.1 Detailed Description

String and Memory manipu32ation class.

Utility method class implementing common memory and string manipu32ation functions, without relying on an external standard library implementation which might not be available on some toolchains, may be closed source, or may not be thread-safe.

Definition at line 46 of file memutil.h.

### 19.19.2 Member Function Documentation

### 19.19.2.1 Checksum16()

#### Checksum16.

Compute the 16-bit addative checksum of a memory buffer.

## **Parameters**

pvSrc⊷	Memory buffer to compute a 16-bit checksum of.
_	
u16⇔	Length of the buffer in bytes.
Len_	

### Returns

16-bit checksum of the memory block.

Definition at line 346 of file memutil.cpp.

### 19.19.2.2 Checksum8()

# Checksum8.

Compute the 8-bit addative checksum of a memory buffer.

#### **Parameters**

pvSrc←	Memory buffer to compute a 8-bit checksum of.
 u16⇔	Length of the buffer in bytes.
Len_	

#### Returns

8-bit checksum of the memory block.

Definition at line 333 of file memutil.cpp.

## 19.19.2.3 CompareMemory()

## CompareMemory.

Compare the contents of two memory buffers to eachother

#### **Parameters**

pv⊷	First buffer to compare
pv⊷ Mem1_	
pv⊷	Second buffer to compare
pv⊷ Mem2_	
u16Len←	Length of buffer (in bytes) to compare

### Returns

true if the buffers match, false if they do not.

Definition at line 467 of file memutil.cpp.

### 19.19.2.4 CompareStrings() [1/2]

CompareStrings.

Compare the contents of two zero-terminated string buffers to eachother.

### **Parameters**

sz⊷ Str1_	First string to compare
SZ←	Second string to compare
Str2	

#### Returns

true if strings match, false otherwise.

Definition at line 372 of file memutil.cpp.

# 19.19.2.5 CompareStrings() [2/2]

Definition at line 391 of file memutil.cpp.

### 19.19.2.6 CopyMemory()

CopyMemory.

Copy one buffer in memory into another.

### **Parameters**

pvDst⊷	Pointer to the destination buffer
_	
pvSrc⊷	Pointer to the source buffer
_	
u16⇔	Number of bytes to copy from source to destination
Len_	

Generated by Doxygen

Definition at line 408 of file memutil.cpp.

### 19.19.2.7 CopyString()

# CopyString.

Copy a string from one buffer into another.

### **Parameters**

SZ←⊃	Pointer to the buffer to copy into
Dst_	
SZ←	Pointer to the buffer to copy data from
Src_	

Definition at line 422 of file memutil.cpp.

## 19.19.2.8 DecimalToHex() [1/4]

DecimalToHex.

Convert an 8-bit unsigned binary value as a hexadecimal string.

#### **Parameters**

<i>u8</i> ⇔	Value to convert into a string
Data_	
SZ←	Destination string buffer (3 bytes minimum)
Text	

### **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 29 of file memutil.cpp.

### 19.19.2.9 DecimalToHex() [2/4]

Definition at line 55 of file memutil.cpp.

# **19.19.2.10 DecimalToHex()** [3/4]

Definition at line 81 of file memutil.cpp.

# 19.19.2.11 DecimalToHex() [4/4]

Definition at line 107 of file memutil.cpp.

# **19.19.2.12 DecimalToString()** [1/4]

### DecimalToString.

Convert an 8-bit unsigned binary value as a decimal string.

#### **Parameters**

<i>u8</i> ⇔	Value to convert into a string
Data_	
SZ⊷	Destination string buffer (4 bytes minimum)
Text	

# **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 133 of file memutil.cpp.

```
19.19.2.13 DecimalToString() [2/4]
```

Definition at line 157 of file memutil.cpp.

### 19.19.2.14 DecimalToString() [3/4]

Definition at line 178 of file memutil.cpp.

### 19.19.2.15 **DecimalToString()** [4/4]

Definition at line 199 of file memutil.cpp.

### 19.19.2.16 SetMemory()

SetMemory.

Initialize a buffer of memory to a specified 8-bit pattern

# **Parameters**

pvDst⊷	Destination buffer to set
_	
u8Val⊷	8-bit pattern to initialize each byte of destination with
_	
u16 Generated by D Len_	Length of the buffer (in bytes) to initialize

Definition at line 486 of file memutil.cpp.

### 19.19.2.17 StringLength()

StringLength.

Compute the length of a string in bytes.

### **Parameters**

<i>SZ</i> ←	Pointer to the zero-terminated string to calculate the length of	l
Str_		

### Returns

length of the string (in bytes), not including the 0-terminator.

Definition at line 360 of file memutil.cpp.

# 19.19.2.18 StringSearch()

StringSearch.

Search for the presence of one string as a substring within another.

## Parameters

szBuffer⊷	Buffer to search for pattern within
_	
SZ⊷	Pattern to search for in the buffer
Pattern	

## Returns

Index of the first instance of the pattern in the buffer, or -1 on no match.

Definition at line 436 of file memutil.cpp.

### 19.19.2.19 StringToDecimal16()

Definition at line 248 of file memutil.cpp.

# 19.19.2.20 StringToDecimal32()

Definition at line 276 of file memutil.cpp.

### 19.19.2.21 StringToDecimal64()

Definition at line 304 of file memutil.cpp.

# 19.19.2.22 StringToDecimal8()

StringToDecimial8.

Convert a string to an unsigned integer value.

#### **Parameters**

szText⇔	String to convert
_	
pu8⊷	Pointer to a uint8_t that will contain the result
Out_	

## Returns

true on success, false on invalid parameters or failure to convert the input string to an unsigned integer value

Definition at line 220 of file memutil.cpp.

#### 19.19.2.23 Tokenize()

Tokenize Function to tokenize a string based on a space delimeter. This is a non-destructive function, which popu32ates a Token\_t descriptor array.

#### **Parameters**

szBuffer_	String to tokenize
pastTokens_	Pointer to the array of token descriptors
u8Max← Tokens_	Maximum number of tokens to parse (i.e. size of pastTokens_)

#### Returns

Count of tokens parsed

Definition at line 496 of file memutil.cpp.

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

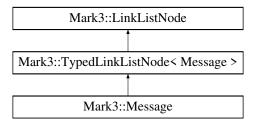
- /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h
- /home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp

# 19.20 Mark3::Message Class Reference

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

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

Inheritance diagram for Mark3::Message:



### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- void Init ()

Init Initialize the data and code in the message.

void SetData (void \*pvData )

SetData Set the data pointer for the message before transmission.

void \* GetData ()

GetData Get the data pointer stored in the message upon receipt.

• void SetCode (uint16 t u16Code )

SetCode Set the code in the message before transmission.

uint16\_t GetCode ()

GetCode Return the code set in the message upon receipt.

#### **Private Attributes**

void \* m\_pvData

Pointer to the message data.

uint16\_t m\_u16Code

Message code, providing context for the message.

#### **Additional Inherited Members**

## 19.20.1 Detailed Description

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

### **Examples:**

lab8\_messages/main.cpp.

Definition at line 97 of file message.h.

## 19.20.2 Member Function Documentation

```
19.20.2.1 GetCode()
```

```
uint16_t Mark3::Message::GetCode ( ) [inline]
```

GetCode Return the code set in the message upon receipt.

# Returns

user code set in the object

Definition at line 139 of file message.h.

### 19.20.2.2 GetData()

```
void* Mark3::Message::GetData ( ) [inline]
```

GetData 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.

### 19.20.2.3 Init()

Init Initialize the data and code in the message.

Definition at line 105 of file message.h.

# 19.20.2.4 operator new()

Definition at line 100 of file message.h.

# 19.20.2.5 SetCode()

SetCode Set the code in the message before transmission.

# **Parameters**

<i>u</i> 16⇔	Data code to set in the object
Code_	

# **Examples:**

lab8\_messages/main.cpp.

Definition at line 132 of file message.h.

#### 19.20.2.6 SetData()

SetData Set the data pointer for the message before transmission.

### **Parameters**

pv⇔	Pointer to the data object to send in the message
Data_	

Definition at line 118 of file message.h.

### 19.20.3 Member Data Documentation

```
19.20.3.1 m_pvData
```

```
void* Mark3::Message::m_pvData [private]
```

Pointer to the message data.

Definition at line 143 of file message.h.

```
19.20.3.2 m_u16Code
```

```
uint16_t Mark3::Message::m_u16Code [private]
```

Message code, providing context for the message.

Definition at line 146 of file message.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/message.h

# 19.21 Mark3::MessagePool Class Reference

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

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

### **Public Member Functions**

### **Private Attributes**

TypedDoubleLinkList< Message > m\_clList
 Linked list used to manage the Message objects.

### 19.21.1 Detailed Description

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

# **Examples:**

```
lab8_messages/main.cpp.
```

Definition at line 157 of file message.h.

#### 19.21.2 Constructor & Destructor Documentation

```
19.21.2.1 ~MessagePool()

Mark3::MessagePool::~MessagePool ( ) [inline]
```

Definition at line 161 of file message.h.

# 19.21.3 Member Function Documentation

```
19.21.3.1 GetHead()
```

```
Message * Mark3::MessagePool::GetHead ( )
```

GetHead.

Return a pointer to the first element in the message list

Returns

Definition at line 52 of file message.cpp.

# 19.21.3.2 Init()

Init.

Initialize the message queue prior to use

Definition at line 26 of file message.cpp.

### 19.21.3.3 operator new()

Definition at line 160 of file message.h.

### 19.21.3.4 Pop()

```
Message * Mark3::MessagePool::Pop ( )
```

Pop.

Pop a message from the queue, returning it to the user to be popu32ated before sending by a transmitter.

Returns

Pointer to a Message object

### **Examples:**

lab8\_messages/main.cpp.

Definition at line 41 of file message.cpp.

### 19.21.3.5 Push()

Push.

Return a previously-claimed message object back to the queue. used once the message has been processed by a receiver.

### **Parameters**

pcl←	Pointer to the Message object to return back to the queue
Message_	

### **Examples:**

lab8\_messages/main.cpp.

Definition at line 32 of file message.cpp.

# 19.21.4 Member Data Documentation

```
19.21.4.1 m_clList
```

```
TypedDoubleLinkList<Message> Mark3::MessagePool::m_clList [private]
```

Linked list used to manage the Message objects.

Definition at line 201 of file message.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/message.h
- /home/moslevin/projects/m3-repo/kernel/src/message.cpp

# 19.22 Mark3::MessageQueue Class Reference

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

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

### **Public Member Functions**

```
void * operator new (size_t sz, void *pv)
```

∼MessageQueue ()

• void Init ()

Init.

• Message \* Receive ()

Receive.

Message \* Receive (uint32\_t u32TimeWaitMS\_)

Receive.

• void Send (Message \*pclSrc\_)

Send.

• uint16\_t GetCount ()

GetCount.

### **Private Member Functions**

```
    Message * Receive_i (uint32_t u32TimeWaitMS_)
    Receive_i.
```

### **Private Attributes**

• Semaphore m\_clSemaphore

Counting semaphore used to manage thread blocking.

• TypedDoubleLinkList< Message > m\_clLinkList

List object used to store messages.

# 19.22.1 Detailed Description

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

### **Examples:**

```
lab8_messages/main.cpp.
```

Definition at line 210 of file message.h.

### 19.22.2 Constructor & Destructor Documentation

```
19.22.2.1 \simMessageQueue()
```

```
Mark3::MessageQueue::~MessageQueue ( ) [inline]
```

Definition at line 214 of file message.h.

### 19.22.3 Member Function Documentation

```
19.22.3.1 GetCount()
```

```
uint16_t Mark3::MessageQueue::GetCount ( )
```

### GetCount.

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

Returns

Count of pending messages in the queue.

Definition at line 108 of file message.cpp.

```
19.22.3.2 Init()
```

Init.

Initialize the message queue prior to use.

Definition at line 58 of file message.cpp.

#### 19.22.3.3 operator new()

Definition at line 213 of file message.h.

```
19.22.3.4 Receive() [1/2]

Message * Mark3::MessageQueue::Receive ( )
```

### Receive.

Receive a message from the message queue. If the message queue is empty, the thread will block until a message is available.

### Returns

Pointer to a message object at the head of the queue

### **Examples:**

lab8\_messages/main.cpp.

Definition at line 64 of file message.cpp.

### Receive.

Receive a message from the message queue. If the message queue is empty, the thread will block until a message is available for the duration specified. If no message arrives within that duration, the call will return with nullptr.

# **Parameters**

u32TimeWaitM⊷	The amount of time in ms to wait for a message before timing out and unblocking the
S_	waiting thread.

### Returns

Pointer to a message object at the head of the queue or nullptr on timeout.

Definition at line 70 of file message.cpp.

### 19.22.3.6 Receive\_i()

Receive\_i.

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

### **Parameters**

u32TimeWaitM⊷	Time (in ms) to block, 0 for un-timed call.
S_	

### Returns

Pointer to a message, or 0 on timeout.

Definition at line 76 of file message.cpp.

#### 19.22.3.7 Send()

### Send.

Send a message object into this message queue. Will un-block the first waiting thread blocked on this queue if that occurs.

### **Parameters**

pcl←	Pointer to the message object to add to the queue
Src_	

# Examples:

lab8\_messages/main.cpp.

Definition at line 92 of file message.cpp.

# 19.22.4 Member Data Documentation

19.22.4.1 m\_clLinkList

TypedDoubleLinkList<Message> Mark3::MessageQueue::m\_clLinkList [private]

List object used to store messages.

Definition at line 284 of file message.h.

19.22.4.2 m\_clSemaphore

```
Semaphore Mark3::MessageQueue::m_clSemaphore [private]
```

Counting semaphore used to manage thread blocking.

Definition at line 281 of file message.h.

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

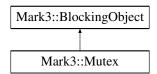
- /home/moslevin/projects/m3-repo/kernel/src/public/message.h
- /home/moslevin/projects/m3-repo/kernel/src/message.cpp

### 19.23 Mark3::Mutex Class Reference

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

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

Inheritance diagram for Mark3::Mutex:



### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ∼Mutex ()
- void Init (bool bRecursive\_=true)

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

• void Claim ()

Claim Claim the mutex. When the mutex is claimed, no other thread can claim a region protected by the object. If another Thread currently holds the Mutex when the Claim method is called, that Thread will block until the current owner of the mutex releases the Mutex.

bool Claim (uint32\_t u32WaitTimeMS\_)

Claim Claim a mutex, with timeout.

void WakeMe (Thread \*pclOwner\_)

WakeMe Wake a thread blocked on the mutex. This is an internal function used for implementing timed mutexes relying on timer callbacks. Since these do not have access to the private data of the mutex and its base classes, we have to wrap this as a public method - do not use this for any other purposes.

• void Release ()

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

### **Private Member Functions**

• uint8\_t WakeNext ()

WakeNext.

bool Claim\_i (uint32\_t u32WaitTimeMS\_)

Claim\_i Abstracts out timed/non-timed mutex claim operations.

#### **Private Attributes**

• uint8\_t m\_u8Recurse

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

bool m\_bReady

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

· bool m bRecursive

Whether or not the lock is recursive.

PORT\_PRIO\_TYPE m\_uMaxPri

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**

## 19.23.1 Detailed Description

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

### **Examples:**

lab5\_mutexes/main.cpp.

Definition at line 63 of file mutex.h.

### 19.23.2 Constructor & Destructor Documentation

```
19.23.2.1 \simMutex()
```

Mark3::Mutex::~Mutex ( )

Definition at line 55 of file mutex.cpp.

## 19.23.3 Member Function Documentation

Claim Claim the mutex. When the mutex is claimed, no other thread can claim a region protected by the object. If another Thread currently holds the Mutex when the Claim method is called, that Thread will block until the current owner of the mutex releases the Mutex.

If the calling Thread's priority is lower than that of a Thread that currently owns the Mutex object, then the priority of that Thread will be elevated to that of the highest-priority calling object until the Mutex is released. This property is known as "Priority Inheritence"

Note: A single thread can recursively claim a mutex up to a count of

1. Attempting to claim a mutex beyond that will cause a kernel panic.

### **Examples:**

lab5\_mutexes/main.cpp.

Definition at line 185 of file mutex.cpp.

Claim Claim a mutex, with timeout.

### **Parameters**

```
u32WaitTimeM←
S_
```

#### Returns

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

Definition at line 191 of file mutex.cpp.

Claim\_i Abstracts out timed/non-timed mutex claim operations.

#### **Parameters**

u32WaitTimeM←	Time in MS to wait, 0 for infinite
S_	

#### Returns

true on successful claim, false otherwise

Definition at line 108 of file mutex.cpp.

```
19.23.3.4 Init()
```

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

#### **Parameters**

b⇔	Whether or not the mutex can be recursively locked.
Recursive←	

Definition at line 93 of file mutex.cpp.

### 19.23.3.5 operator new()

Definition at line 66 of file mutex.h.

### 19.23.3.6 Release()

```
void Mark3::Mutex::Release ( )
```

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

If there are Threads waiting for the Mutex to become available, then the highest priority Thread will be unblocked at this time and will claim the Mutex lock immediately - this may result in an immediate context switch, depending on relative priorities.

If the calling Thread's priority was boosted as a result of priority inheritence, the Thread's previous priority will also be restored at this time.

Note that if a Mutex is held recursively, it must be Release'd the same number of times that it was Claim'd before it will be availabel for use by another Thread.

### **Examples:**

lab5\_mutexes/main.cpp.

Definition at line 197 of file mutex.cpp.

### 19.23.3.7 WakeMe()

WakeMe Wake a thread blocked on the mutex. This is an internal function used for implementing timed mutexes relying on timer callbacks. Since these do not have access to the private data of the mutex and its base classes, we have to wrap this as a public method - do not use this for any other purposes.

#### **Parameters**

pcl⇔	Thread to unblock from this object.
Owner_	

Definition at line 65 of file mutex.cpp.

### 19.23.3.8 WakeNext()

```
uint8_t Mark3::Mutex::WakeNext ( ) [private]
```

WakeNext.

Wake the next thread waiting on the Mutex.

Definition at line 73 of file mutex.cpp.

#### 19.23.4 Member Data Documentation

```
19.23.4.1 m_bReady
```

```
bool Mark3::Mutex::m_bReady [private]
```

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

Definition at line 159 of file mutex.h.

```
19.23.4.2 m_bRecursive
```

```
bool Mark3::Mutex::m_bRecursive [private]
```

Whether or not the lock is recursive.

Definition at line 160 of file mutex.h.

19.23.4.3 m\_pclOwner

```
Thread* Mark3::Mutex::m_pclOwner [private]
```

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

Definition at line 162 of file mutex.h.

19.23.4.4 m\_u8Recurse

```
uint8_t Mark3::Mutex::m_u8Recurse [private]
```

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

Definition at line 158 of file mutex.h.

19.23.4.5 m\_uMaxPri

```
PORT_PRIO_TYPE Mark3::Mutex::m_uMaxPri [private]
```

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

Definition at line 161 of file mutex.h.

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

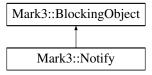
- /home/moslevin/projects/m3-repo/kernel/src/public/mutex.h
- /home/moslevin/projects/m3-repo/kernel/src/mutex.cpp

# 19.24 Mark3::Notify Class Reference

The Notify class. This class provides a blocking object type that allows one or more threads to wait for an event to occur before resuming operation.

```
#include <notify.h>
```

Inheritance diagram for Mark3::Notify:



#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ∼Notify ()
- void Init (void)

Init Initialze the Notification object prior to use.

void Signal (void)

Signal Signal the notification object. This will cause the highest priority thread currently blocking on the object to wake. If no threads are currently blocked on the object, the call has no effect.

void Wait (bool \*pbFlag )

Wait Block the current thread, waiting for a signal on the object.

bool Wait (uint32\_t u32WaitTimeMS\_, bool \*pbFlag\_)

Wait Block the current thread, waiting for a signal on the object.

void WakeMe (Thread \*pclChosenOne\_)

WakeMe Wake the specified thread from its current blocking queue. Note that this is only public in order to be accessible from a timer callack.

### **Private Attributes**

bool m\_bPending

### **Additional Inherited Members**

## 19.24.1 Detailed Description

The Notify class. This class provides a blocking object type that allows one or more threads to wait for an event to occur before resuming operation.

### **Examples:**

lab10\_notifications/main.cpp.

Definition at line 33 of file notify.h.

# 19.24.2 Constructor & Destructor Documentation

```
19.24.2.1 ~Notify()

Mark3::Notify::~Notify ( )

Definition at line 47 of file notify.cpp.
```

### 19.24.3 Member Function Documentation

Init Initialze the Notification object prior to use.

Definition at line 57 of file notify.cpp.

### 19.24.3.2 operator new()

Definition at line 36 of file notify.h.

### 19.24.3.3 Signal()

Signal Signal the notification object. This will cause the highest priority thread currently blocking on the object to wake. If no threads are currently blocked on the object, the call has no effect.

### **Examples:**

lab10\_notifications/main.cpp.

Definition at line 66 of file notify.cpp.

Wait Block the current thread, waiting for a signal on the object.

## **Parameters**

pb⊷	Flag set to false on block, and true upon wakeup.
Flag_	

## **Examples:**

lab10\_notifications/main.cpp.

Definition at line 95 of file notify.cpp.

Wait Block the current thread, waiting for a signal on the object.

## **Parameters**

u32WaitTimeM⊷	Time to wait for the notification event.
S_	
pbFlag_	Flag set to false on block, and true upon wakeup.

## Returns

true on notification, false on timeout

Definition at line 125 of file notify.cpp.

# 19.24.3.6 WakeMe()

WakeMe Wake the specified thread from its current blocking queue. Note that this is only public in order to be accessible from a timer callack.

## **Parameters**

pclChosen←	Thread to wake up
One_	

Definition at line 175 of file notify.cpp.

## 19.24.4 Member Data Documentation

#### 19.24.4.1 m\_bPending

```
bool Mark3::Notify::m_bPending [private]
```

Definition at line 89 of file notify.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/notify.h
- /home/moslevin/projects/m3-repo/kernel/src/notify.cpp

# 19.25 Mark3::PriorityMapL1 < T, C > Class Template Reference

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

```
#include <priomapl1.h>
```

## **Public Member Functions**

• PriorityMapL1 ()

PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.

void Set (T uXPrio )

Set Set the priority map bitmap data, at all levels, for the given priority.

• void Clear (T uXPrio\_)

Clear Clear the priority map bitmap data, at all levels, for the given priority.

• T HighestPriority (void)

HighestPriority Computes the numeric priority of the highest-priority thread represented in the priority map.

## **Static Private Member Functions**

- static T PrioBit (T prio)
- static T PrioMapWordIndex (T prio)
- static T PriorityFromBitmap (T uXPrio\_)

## **Private Attributes**

T m\_uXPriorityMap

## **Static Private Attributes**

- static constexpr size\_t m\_uXPrioMapShiftLUT [9] = {0, 3, 4, 0, 5, 0, 0, 0, 6}
- static constexpr auto m\_uXPrioMapWordShift = T { m\_uXPrioMapShiftLUT[sizeof(T)] }
- static constexpr auto m\_uXPrioMapBits = T { 8 \* sizeof(T) }
- static constexpr auto m\_uXPrioMapBitMask = T { (1 << m\_uXPrioMapWordShift) 1 }

## 19.25.1 Detailed Description

```
template<typename T, size_t C> class Mark3::PriorityMapL1< T, C >
```

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

The L1 version of the datastructure uses a single unsigned integer (specified by the templated typname "T"), to support "C" priorities. Thus, the maximum number of priorities supported in the map is the number of bits in the "T" parameter.

Definition at line 45 of file priomapl1.h.

## 19.25.2 Constructor & Destructor Documentation

## 19.25.2.1 PriorityMapL1()

```
template<typename T , size_t C>
Mark3::PriorityMapL1< T, C >::PriorityMapL1 ( ) [inline]
```

PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.

Definition at line 52 of file priomapl1.h.

# 19.25.3 Member Function Documentation

#### 19.25.3.1 Clear()

Clear Clear the priority map bitmap data, at all levels, for the given priority.

#### **Parameters**

uX⊷	Priority level to clear the bitmap data for.
Prio_	

Definition at line 73 of file priomapl1.h.

## 19.25.3.2 HighestPriority()

HighestPriority Computes the numeric priority of the highest-priority thread represented in the priority map.

#### Returns

Highest priority ready-thread's number.

Definition at line 86 of file priomapl1.h.

#### 19.25.3.3 PrioBit()

Definition at line 93 of file priomapl1.h.

#### 19.25.3.4 PrioMapWordIndex()

Definition at line 95 of file priomapl1.h.

## 19.25.3.5 PriorityFromBitmap()

Definition at line 97 of file priomapl1.h.

## 19.25.3.6 Set()

Set Set the priority map bitmap data, at all levels, for the given priority.

#### **Parameters**

uX⊷	Priority level to set the bitmap data for.
Prio_	

Definition at line 62 of file priomapl1.h.

# 19.25.4 Member Data Documentation

## 19.25.4.1 m\_uXPrioMapBitMask

```
template<typename T , size_t C> constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWord \leftarrow Shift) - 1 } [static], [private]
```

Definition at line 122 of file priomapl1.h.

## 19.25.4.2 m\_uXPrioMapBits

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapBits = T { 8 * sizeof(T) } [static],
[private]
```

Definition at line 121 of file priomapl1.h.

## 19.25.4.3 m\_uXPrioMapShiftLUT

```
template<typename T , size_t C>
constexpr size_t Mark3::PriorityMapL1< T, C >::m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0, 0, 6} [static], [private]
```

Definition at line 119 of file priomapl1.h.

# 19.25.4.4 m\_uXPrioMapWordShift

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapWordShift = T { m_uXPrioMapShiftL
UT[sizeof(T)] } [static], [private]
```

Definition at line 120 of file priomapl1.h.

## 19.25.4.5 m\_uXPriorityMap

```
template<typename T , size_t C>
T Mark3::PriorityMapL1< T, C >::m_uXPriorityMap [private]
```

Definition at line 124 of file priomapl1.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h

# 19.26 Mark3::PriorityMapL2< T, C > Class Template Reference

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

```
#include <priomapl2.h>
```

## **Public Member Functions**

• PriorityMapL2 ()

PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.

void Set (T uXPrio\_)

Set Set the priority map bitmap data, at all levels, for the given priority.

void Clear (T uXPrio\_)

Clear Clear the priority map bitmap data, at all levels, for the given priority.

T HighestPriority (void)

HighestPriority Computes the numeric priority of the highest-priority thread represented in the priority map.

#### **Static Private Member Functions**

- static T PrioBit (T prio)
- static T PrioMapWordIndex (T prio)
- static T PriorityFromBitmap (T uXPrio\_)

## **Private Attributes**

- T m\_auXPriorityMap [m\_uXPrioMapNumWords]
- T m\_uXPriorityMapL2

## **Static Private Attributes**

- static constexpr size\_t m\_uXPrioMapShiftLUT [9] = {0, 3, 4, 0, 5, 0, 0, 0, 6}
- static constexpr auto m\_uXPrioMapWordShift = T { m\_uXPrioMapShiftLUT[sizeof(T)] }
- static constexpr auto m\_uXPrioMapBits = T { 8 \* sizeof(T) }
- static constexpr auto m\_uXPrioMapBitMask = T { (1 << m\_uXPrioMapWordShift) 1 }
- static constexpr auto m\_uXPrioMapNumWords = T { (C + (m\_uXPrioMapBits 1)) / m\_uXPrioMapBits }

## 19.26.1 Detailed Description

```
template<typename T, size_t C> class Mark3::PriorityMapL2< T, C >
```

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

The L2 version of the datastructure uses a two-layer nested map structure, where a first layer bitmap contains a single unsigned integer of type "T", where each bit corresponds to an array entry in the second layer table. The second layer consists of an array of unsigned integers of type "T", where each bit in each element corresponds to a level of priority supported by the map structure. As a result, the maximum number of priorities ("C") supported by the object is n\*n, where n is the number of bits in the "T" parameter. i.e., if an 8-bit unsigned int is used, this object supports up to 64 priorities; and if a 32-bit unsigned int is used, the object supports up to 1024 priorities.

Definition at line 50 of file priomapl2.h.

## 19.26.2 Constructor & Destructor Documentation

## 19.26.2.1 PriorityMapL2()

```
template<typename T , size_t C>
Mark3::PriorityMapL2< T, C >::PriorityMapL2 ( ) [inline]
```

PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.

Definition at line 57 of file priomapl2.h.

## 19.26.3 Member Function Documentation

#### 19.26.3.1 Clear()

Clear Clear the priority map bitmap data, at all levels, for the given priority.

## **Parameters**

uX⊷	Priority level to clear the bitmap data for.
Prio_	

Definition at line 82 of file priomapl2.h.

## 19.26.3.2 HighestPriority()

HighestPriority Computes the numeric priority of the highest-priority thread represented in the priority map.

Returns

Highest priority ready-thread's number.

Definition at line 100 of file priomapl2.h.

## 19.26.3.3 PrioBit()

Definition at line 113 of file priomapl2.h.

# 19.26.3.4 PrioMapWordIndex()

Definition at line 115 of file priomapl2.h.

## 19.26.3.5 PriorityFromBitmap()

Definition at line 117 of file priomapl2.h.

#### 19.26.3.6 Set()

Set Set the priority map bitmap data, at all levels, for the given priority.

#### **Parameters**

uX⊷	Priority level to set the bitmap data for.
Prio_	

Definition at line 68 of file priomapl2.h.

## 19.26.4 Member Data Documentation

## 19.26.4.1 m\_auXPriorityMap

```
template<typename T , size_t C>
T Mark3::PriorityMapL2< T, C >::m_auXPriorityMap[m_uXPrioMapNumWords] [private]
```

Definition at line 148 of file priomapl2.h.

## 19.26.4.2 m\_uXPrioMapBitMask

```
template<typename T , size_t C> constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWord \leftarrow Shift) - 1 } [static], [private]
```

Definition at line 142 of file priomapl2.h.

# 19.26.4.3 m\_uXPrioMapBits

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapBits = T { 8 * sizeof(T) } [static],
[private]
```

Definition at line 141 of file priomapl2.h.

## 19.26.4.4 m\_uXPrioMapNumWords

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapNumWords = T { (C + (m_uXPrioMapBits -
1)) / m_uXPrioMapBits } [static], [private]
```

Definition at line 146 of file priomapl2.h.

## 19.26.4.5 m\_uXPrioMapShiftLUT

```
template<typename T , size_t C>
constexpr size_t Mark3::PriorityMapL2< T, C >::m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0, 0, 6} [static], [private]
```

Definition at line 139 of file priomapl2.h.

## 19.26.4.6 m\_uXPrioMapWordShift

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapWordShift = T { m_uXPrioMapShiftL\lefta}
UT[sizeof(T)] } [static], [private]
```

Definition at line 140 of file priomapl2.h.

## 19.26.4.7 m\_uXPriorityMapL2

```
template<typename T , size_t C>
T Mark3::PriorityMapL2< T, C >::m_uXPriorityMapL2 [private]
```

Definition at line 149 of file priomapl2.h.

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

/home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h

## 19.27 Mark3::ProfileTimer Class Reference

Profiling timer. This class is used to perform high-performance profiling of code to see how int32\_t certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

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

## **Public Member Functions**

• void Init ()

Init Initialize the profiling timer prior to use. Can also be used to reset a timer that's been used previously.

• void Start ()

Start Start a profiling session, if the timer is not already active. Has no effect if the timer is already active.

• void Stop ()

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

uint32\_t GetAverage ()

GetAverage Get the average time associated with this operation.

• uint32\_t GetCurrent ()

GetCurrent Return the current tick count held by the profiler. Valid for both active and stopped timers.

## **Private Attributes**

• uint32\_t m\_u32StartTicks

Cumulative tick-count for this timer.

• uint32\_t m\_u32CurrentIteration

Tick count for current iteration.

• uint32\_t m\_u32Cumulative

Cumulative ticks tracked.

• uint16\_t m\_u16Iterations

Number of iterations executed for this profiling timer.

· bool m bActive

Wheter or not the timer is active or stopped.

## 19.27.1 Detailed Description

Profiling timer. This class is used to perform high-performance profiling of code to see how int32\_t certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

Definition at line 67 of file profile.h.

## 19.27.2 Member Function Documentation

## 19.27.2.1 GetAverage()

```
uint32_t Mark3::ProfileTimer::GetAverage ( )
```

GetAverage Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 63 of file profile.cpp.

## 19.27.2.2 GetCurrent()

```
uint32_t Mark3::ProfileTimer::GetCurrent ( )
```

GetCurrent Return the current tick count held by the profiler. Valid for both active and stopped timers.

## Returns

The currently held tick count.

Definition at line 72 of file profile.cpp.

## 19.27.2.3 Init()

Init Initialize the profiling timer prior to use. Can also be used to reset a timer that's been used previously.

Definition at line 25 of file profile.cpp.

## 19.27.2.4 Start()

Start Start a profiling session, if the timer is not already active. Has no effect if the timer is already active.

Definition at line 34 of file profile.cpp.

## 19.27.2.5 Stop()

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

Definition at line 46 of file profile.cpp.

# 19.27.3 Member Data Documentation

## 19.27.3.1 m\_bActive

```
bool Mark3::ProfileTimer::m_bActive [private]
```

Wheter or not the timer is active or stopped.

Definition at line 113 of file profile.h.

## 19.27.3.2 m\_u16Iterations

```
uint16_t Mark3::ProfileTimer::m_u16Iterations [private]
```

Number of iterations executed for this profiling timer.

Definition at line 112 of file profile.h.

## 19.27.3.3 m\_u32Cumulative

```
uint32_t Mark3::ProfileTimer::m_u32Cumulative [private]
```

Cumulative ticks tracked.

Definition at line 111 of file profile.h.

#### 19.27.3.4 m\_u32CurrentIteration

```
uint32_t Mark3::ProfileTimer::m_u32CurrentIteration [private]
```

Tick count for current iteration.

Definition at line 110 of file profile.h.

## 19.27.3.5 m\_u32StartTicks

```
uint32_t Mark3::ProfileTimer::m_u32StartTicks [private]
```

Cumulative tick-count for this timer.

Definition at line 109 of file profile.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/profile.h
- /home/moslevin/projects/m3-repo/kernel/src/profile.cpp

## 19.28 Mark3::Quantum Class Reference

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

```
#include <quantum.h>
```

## **Static Public Member Functions**

- · static void Init ()
- static void SetInTimer ()

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.

static void ClearInTimer ()

ClearInTimer Clear the flag once the timer callback function has been completed.

static void Update (Thread \*pclTargetThread )

Update Update the current thread being tracked for round-robing scheduling. Note - this has no effect if called from the Timer thread, or if the Timer thread is active.

static void SetTimerThread (Thread \*pclTimerThread\_)

SetTimerThread Pass the timer thread's Thread pointer to the Quantum module to track against requests to update the round-robin timer.

· static void Cancel ()

## **Static Private Attributes**

- static Thread \* m\_pclActiveThread
- static Thread \* m pclTimerThread
- static uint16\_t m\_u16TicksRemain
- static bool m blnTimer

## 19.28.1 Detailed Description

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

Definition at line 42 of file quantum.h.

## 19.28.2 Member Function Documentation

```
19.28.2.1 Cancel()
```

```
static void Mark3::Quantum::Cancel ( ) [static]
```

Cancel the round-robin timer.

# 19.28.2.2 ClearInTimer()

```
static void Mark3::Quantum::ClearInTimer ( ) [static]
```

ClearInTimer Clear the flag once the timer callback function has been completed.

## 19.28.2.3 Init()

```
static void Mark3::Quantum::Init ( ) [static]
```

## 19.28.2.4 SetInTimer()

```
static void Mark3::Quantum::SetInTimer ( ) [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.

## 19.28.2.5 SetTimerThread()

SetTimerThread Pass the timer thread's Thread pointer to the Quantum module to track against requests to update the round-robin timer.

## **Parameters**

pclTimer←	Pointer to the Timer thread's Thread object.
Thread_	

Definition at line 79 of file quantum.h.

## 19.28.2.6 Update()

Update Update the current thread being tracked for round-robing scheduling. Note - this has no effect if called from the Timer thread, or if the Timer thread is active.

## **Parameters**

pclTarget←	New thread to track.
Thread	

# 19.28.3 Member Data Documentation

```
19.28.3.1 m_blnTimer
bool Mark3::Quantum::m_bInTimer [static], [private]
Definition at line 90 of file quantum.h.

19.28.3.2 m_pclActiveThread
Thread* Mark3::Quantum::m_pclActiveThread [static], [private]
Definition at line 87 of file quantum.h.

19.28.3.3 m_pclTimerThread
Thread* Mark3::Quantum::m_pclTimerThread [static], [private]
Definition at line 88 of file quantum.h.
```

19.28.3.4 m\_u16TicksRemain

```
uint16_t Mark3::Quantum::m_u16TicksRemain [static], [private]
```

Definition at line 89 of file quantum.h.

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

/home/moslevin/projects/m3-repo/kernel/src/public/quantum.h

## 19.29 Mark3::ReaderWriterLock Class Reference

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

```
#include <readerwriter.h>
```

# **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- void Init ()

Init Initialize the reader-writer lock before use. Must be called before attempting any other operations on the object.

void AcquireReader ()

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

bool AcquireReader (uint32 t u32TimeoutMs )

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

• void ReleaseReader ()

ReleaseReader Release a previously-held reader lock.

• void AcquireWriter ()

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

bool AcquireWriter (uint32 t u32TimeoutMs )

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

void ReleaseWriter ()

Release Writer Release the currently held writer, allowing other readers/writers to access the object.

## **Private Member Functions**

bool AcquireReader i (uint32 t u32TimeoutMs )

AcquireReader\_i Internal helper function for AcquireReaer.

bool AcquireWriter\_i (uint32\_t u32TimeoutMs\_)

AcquireWriter\_i Internal helper function for AcquireWriter.

## **Private Attributes**

Mutex m\_clGlobalMutex

Mutex used to lock the object against concurrent read + write.

· Mutex m clReaderMutex

Mutex used to lock object for readers.

· uint8 t m u8ReadCount

Number of concurrent readers.

## 19.29.1 Detailed Description

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

Definition at line 40 of file readerwriter.h.

# 19.29.2 Member Function Documentation

```
19.29.2.1 AcquireReader() [1/2]
void Mark3::ReaderWriterLock::AcquireReader ( )
```

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

Definition at line 32 of file readerwriter.cpp.

```
19.29.2.2 AcquireReader() [2/2]
bool Mark3::ReaderWriterLock::AcquireReader (
```

 $uint32\_t$   $u32TimeoutMs\_$  )

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

#### **Parameters**

u32Timeout←	Maximum time to wait (in ms) before the operation is aborted
Ms_	

## Returns

true on success, false on timeout

Definition at line 38 of file readerwriter.cpp.

## 19.29.2.3 AcquireReader\_i()

AcquireReader\_i Internal helper function for AcquireReaer.

#### **Parameters**

u32Timeout←	Maximum time to wait (in ms) before the operation is aborted
Ms_	

#### Returns

true on success, false on timeout

Definition at line 73 of file readerwriter.cpp.

```
19.29.2.4 AcquireWriter() [1/2]
```

void Mark3::ReaderWriterLock::AcquireWriter ( )

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

Definition at line 55 of file readerwriter.cpp.

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

## **Parameters**

u32Timeout←	Maximum time to wait (in ms) before the operation is aborted
Ms_	

## Returns

true on success, false on timeout

Definition at line 61 of file readerwriter.cpp.

## 19.29.2.6 AcquireWriter\_i()

AcquireWriter\_i Internal helper function for AcquireWriter.

#### **Parameters**

u32Timeout←	Maximum time to wait (in ms) before the operation is aborted
Ms_	

## Returns

true on success, false on timeout

Definition at line 90 of file readerwriter.cpp.

## 19.29.2.7 Init()

Init Initialize the reader-writer lock before use. Must be called before attempting any other operations on the object.

Definition at line 24 of file readerwriter.cpp.

## 19.29.2.8 operator new()

Definition at line 43 of file readerwriter.h.

## 19.29.2.9 ReleaseReader()

```
void Mark3::ReaderWriterLock::ReleaseReader ( )
```

ReleaseReader Release a previously-held reader lock.

Definition at line 44 of file readerwriter.cpp.

## 19.29.2.10 ReleaseWriter()

```
void Mark3::ReaderWriterLock::ReleaseWriter ( )
```

ReleaseWriter Release the currently held writer, allowing other readers/writers to access the object.

Definition at line 67 of file readerwriter.cpp.

## 19.29.3 Member Data Documentation

19.29.3.1 m\_clGlobalMutex

Mutex Mark3::ReaderWriterLock::m\_clGlobalMutex [private]

Mutex used to lock the object against concurrent read + write.

Definition at line 116 of file readerwriter.h.

19.29.3.2 m\_clReaderMutex

Mutex Mark3::ReaderWriterLock::m\_clReaderMutex [private]

Mutex used to lock object for readers.

Definition at line 117 of file readerwriter.h.

19.29.3.3 m\_u8ReadCount

uint8\_t Mark3::ReaderWriterLock::m\_u8ReadCount [private]

Number of concurrent readers.

Definition at line 118 of file readerwriter.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h
- /home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp

# 19.30 Mark3::Scheduler Class Reference

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

#include <scheduler.h>

## **Static Public Member Functions**

· static void Init ()

Init Intiailize the scheduler, must be called before use.

• static void Schedule ()

Schedule Run the scheduler, determines the next thread to run based on the current state of the threads. Note that the next-thread chosen from this function is only valid while in a critical section.

• static void Add (Thread \*pclThread )

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

static void Remove (Thread \*pclThread\_)

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

static bool SetScheduler (bool bEnable\_)

SetScheduler Set the active state of the scheduler. When the scheduler is disabled, the next thread is never set; the currently running thread will run forever until the scheduler is enabled again. Care must be taken to ensure that we don't end up trying to block while the scheduler is disabled, otherwise the system ends up in an unusable state.

static Thread \* GetCurrentThread ()

GetCurrentThread Return the pointer to the currently-running thread.

static volatile Thread \* GetNextThread ()

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

static ThreadList \* GetThreadList (PORT\_PRIO\_TYPE uXPriority\_)

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

static ThreadList \* GetStopList ()

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

static bool IsEnabled ()

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

static void QueueScheduler ()

QueueScheduler Tell the kernel to perform a scheduling operation as soon as the scheduler is re-enabled.

## **Static Private Attributes**

- static constexpr auto m\_uNumPriorities = size\_t { KERNEL\_NUM\_PRIORITIES }
- static bool m\_bEnabled

Scheduler's state - enabled or disabled.

• static bool m bQueuedSchedule

Variable representing whether or not there's a queued scheduler operation.

static ThreadList m\_clStopList

ThreadList for all stopped threads.

static ThreadList m\_aclPriorities [m\_uNumPriorities]

ThreadLists for all threads at all priorities.

static PriorityMap m\_clPrioMap

Priority bitmap lookup structure, 1-bit per thread priority.

## 19.30.1 Detailed Description

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

Definition at line 63 of file scheduler.h.

## 19.30.2 Member Function Documentation

## 19.30.2.1 Add()

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

#### **Parameters**

pcl⇔	Pointer to the thread to add to the scheduler
Thread_	

Definition at line 59 of file scheduler.cpp.

## 19.30.2.2 GetCurrentThread()

```
static Thread* Mark3::Scheduler::GetCurrentThread ( ) [inline], [static]
```

GetCurrentThread Return the pointer to the currently-running thread.

## Returns

Pointer to the currently-running thread

# Examples:

lab9\_dynamic\_threads/main.cpp.

Definition at line 116 of file scheduler.h.

## 19.30.2.3 GetNextThread()

```
static volatile Thread* Mark3::Scheduler::GetNextThread ( ) [inline], [static]
```

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

## Returns

Pointer to the next-running thread

Definition at line 124 of file scheduler.h.

## 19.30.2.4 GetStopList()

```
static ThreadList* Mark3::Scheduler::GetStopList ( ) [inline], [static]
```

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

## Returns

Pointer to the ThreadList containing the stopped threads

Definition at line 142 of file scheduler.h.

## 19.30.2.5 GetThreadList()

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

## **Parameters**

uX⊷	Priority level of the threadlist
Priority_	

## Returns

Pointer to the ThreadList for the given priority level

Definition at line 134 of file scheduler.h.

## 19.30.2.6 Init()

Init Intiailize the scheduler, must be called before use.

Definition at line 36 of file scheduler.cpp.

## 19.30.2.7 IsEnabled()

```
static bool Mark3::Scheduler::IsEnabled ( ) [inline], [static]
```

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

#### Returns

true - scheduler enabled, false - disabled

Definition at line 150 of file scheduler.h.

## 19.30.2.8 QueueScheduler()

```
static void Mark3::Scheduler::QueueScheduler ( ) [inline], [static]
```

QueueScheduler Tell the kernel to perform a scheduling operation as soon as the scheduler is re-enabled.

Definition at line 156 of file scheduler.h.

## 19.30.2.9 Remove()

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

## **Parameters**

pcl←	Pointer to the thread to be removed from the scheduler	]
Thread_		

Definition at line 67 of file scheduler.cpp.

## 19.30.2.10 Schedule()

```
void Mark3::Scheduler::Schedule ( ) [static]
```

Schedule Run the scheduler, determines the next thread to run based on the current state of the threads. Note that the next-thread chosen from this function is only valid while in a critical section.

Definition at line 45 of file scheduler.cpp.

## 19.30.2.11 SetScheduler()

SetScheduler Set the active state of the scheduler. When the scheduler is disabled, the *next thread* is never set; the currently running thread will run forever until the scheduler is enabled again. Care must be taken to ensure that we don't end up trying to block while the scheduler is disabled, otherwise the system ends up in an unusable state.

#### **Parameters**

b⇔	true to enable, false to disable the scheduler
Enable⊷	

Definition at line 75 of file scheduler.cpp.

#### 19.30.3 Member Data Documentation

```
19.30.3.1 m_aclPriorities
```

```
ThreadList Mark3::Scheduler::m_aclPriorities [static], [private]
```

ThreadLists for all threads at all priorities.

Definition at line 171 of file scheduler.h.

```
19.30.3.2 m_bEnabled
```

```
bool Mark3::Scheduler::m_bEnabled [static], [private]
```

Scheduler's state - enabled or disabled.

Definition at line 162 of file scheduler.h.

#### 19.30.3.3 m\_bQueuedSchedule

```
bool Mark3::Scheduler::m_bQueuedSchedule [static], [private]
```

Variable representing whether or not there's a queued scheduler operation.

Definition at line 165 of file scheduler.h.

```
19.30.3.4 m_clPrioMap
```

```
PriorityMap Mark3::Scheduler::m_clPrioMap [static], [private]
```

Priority bitmap lookup structure, 1-bit per thread priority.

Definition at line 174 of file scheduler.h.

## 19.30.3.5 m\_clStopList

```
ThreadList Mark3::Scheduler::m_clStopList [static], [private]
```

ThreadList for all stopped threads.

Definition at line 168 of file scheduler.h.

## 19.30.3.6 m\_uNumPriorities

```
constexpr auto Mark3::Scheduler::m_uNumPriorities = size_t { KERNEL_NUM_PRIORITIES } [static],
[private]
```

Definition at line 159 of file scheduler.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h
- /home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp

# 19.31 Mark3::SchedulerGuard Class Reference

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

```
#include <schedulerguard.h>
```

## **Public Member Functions**

- · SchedulerGuard ()
- ∼SchedulerGuard ()

#### **Private Attributes**

bool m\_bSchedState

# 19.31.1 Detailed Description

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

Definition at line 37 of file schedulerguard.h.

## 19.31.2 Constructor & Destructor Documentation

```
19.31.2.1 SchedulerGuard()
```

```
Mark3::SchedulerGuard::SchedulerGuard ( ) [inline]
```

Definition at line 39 of file schedulerguard.h.

19.31.2.2 ∼SchedulerGuard()

```
{\tt Mark3::SchedulerGuard::}{\sim} {\tt SchedulerGuard ( ) [inline]}
```

Definition at line 44 of file schedulerguard.h.

## 19.31.3 Member Data Documentation

## 19.31.3.1 m\_bSchedState

```
bool Mark3::SchedulerGuard::m_bSchedState [private]
```

Definition at line 50 of file schedulerguard.h.

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

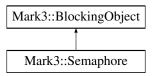
/home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h

# 19.32 Mark3::Semaphore Class Reference

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

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

Inheritance diagram for Mark3::Semaphore:



## **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ∼Semaphore ()
- void Init (uint16\_t u16InitVal\_, uint16\_t u16MaxVal\_)

Initialize a semaphore before use. Must be called before attempting post/pend operations on the object.

· bool Post ()

Increment the semaphore count. If the semaphore count is zero at the time this is called, and there are threads blocked on the object, this will immediately unblock the highest-priority blocked Thread.

· void Pend ()

Decrement the semaphore count. If the count is zero, the calling Thread will block until the semaphore is posted, and the Thread's priority is higher than that of any other Thread blocked on the object.

• uint16 t GetCount ()

Return the current semaphore counter. This can be usedd by a thread to bypass blocking on a semaphore - allowing it to do other things until a non-zero count is returned, instead of blocking until the semaphore is posted.

bool Pend (uint32\_t u32WaitTimeMS\_)

Decrement the semaphore count. If the count is zero, the thread will block until the semaphore is pended. If the specified interval expires before the thread is unblocked, then the status is returned back to the user.

void 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 used this for any other purposes.

# **Private Member Functions**

• uint8\_t WakeNext ()

Wake the next thread waiting on the semaphore. Used internally.

bool Pend i (uint32 t u32WaitTimeMS )

Pend i.

## **Private Attributes**

• uint16\_t m\_u16Value

Current count held by the semaphore.

uint16\_t m\_u16MaxValue

Maximum count that can be held by this semaphore.

## **Additional Inherited Members**

## 19.32.1 Detailed Description

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

#### **Examples:**

lab4 semaphores/main.cpp, lab6 timers/main.cpp, and lab9 dynamic threads/main.cpp.

Definition at line 36 of file ksemaphore.h.

## 19.32.2 Constructor & Destructor Documentation

```
19.32.2.1 ~Semaphore()

Mark3::Semaphore::~Semaphore ( )
```

Definition at line 58 of file ksemaphore.cpp.

## 19.32.3 Member Function Documentation

```
19.32.3.1 GetCount()
uint16_t Mark3::Semaphore::GetCount ( )
```

Return the current semaphore counter. This can be usedd by a thread to bypass blocking on a semaphore - allowing it to do other things until a non-zero count is returned, instead of blocking until the semaphore is posted.

## Returns

The current semaphore counter value.

Definition at line 206 of file ksemaphore.cpp.

```
19.32.3.2 Init()
```

Initialize a semaphore before use. Must be called before attempting post/pend operations on the object.

This initialization is required to configure the behavior of the semaphore with regards to the initial and maximum values held by the semaphore. By providing access to the raw initial and maximum count elements of the semaphore, these objects are able to be used as either counting or binary semaphores.

To configure a semaphore object for use as a binary semaphore, set values of 0 and 1 respectively for the initial/maximum value parameters.

Any other combination of values can be used to implement a counting semaphore.

#### **Parameters**

u16InitVal←	Initial value held by the semaphore
_	
u16Max⊷	Maximum value for the semaphore. Must be nonzero.
Val_	

## **Examples:**

lab6\_timers/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 94 of file ksemaphore.cpp.

## 19.32.3.3 operator new()

Definition at line 39 of file ksemaphore.h.

```
19.32.3.4 Pend() [1/2]
void Mark3::Semaphore::Pend ( )
```

Decrement the semaphore count. If the count is zero, the calling Thread will block until the semaphore is posted, and the Thread's priority is higher than that of any other Thread blocked on the object.

## **Examples:**

lab4\_semaphores/main.cpp.

Definition at line 194 of file ksemaphore.cpp.

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 200 of file ksemaphore.cpp.

# 19.32.3.6 Pend\_i()

# Pend\_i.

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

#### **Parameters**

u32WaitTimeM⊷	Time in MS to wait
<i>S_</i>	

#### Returns

true on success, false on failure.

Definition at line 152 of file ksemaphore.cpp.

## 19.32.3.7 Post()

```
bool Mark3::Semaphore::Post ( )
```

Increment the semaphore count. If the semaphore count is zero at the time this is called, and there are threads blocked on the object, this will immediately unblock the highest-priority blocked Thread.

Note that if the priority of that Thread is higher than the current thread's priority, a context switch will occur and control will be relinquished to that Thread.

## Returns

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

## Examples:

lab4\_semaphores/main.cpp, and lab6\_timers/main.cpp.

Definition at line 108 of file ksemaphore.cpp.

## 19.32.3.8 WakeMe()

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 used this for any other purposes.

Definition at line 68 of file ksemaphore.cpp.

## 19.32.3.9 WakeNext()

```
uint8_t Mark3::Semaphore::WakeNext ( ) [private]
```

Wake the next thread waiting on the semaphore. Used internally.

Definition at line 78 of file ksemaphore.cpp.

## 19.32.4 Member Data Documentation

```
19.32.4.1 m_u16MaxValue
```

```
uint16_t Mark3::Semaphore::m_u16MaxValue [private]
```

Maximum count that can be held by this semaphore.

Definition at line 140 of file ksemaphore.h.

## 19.32.4.2 m\_u16Value

```
uint16_t Mark3::Semaphore::m_u16Value [private]
```

Current count held by the semaphore.

Definition at line 139 of file ksemaphore.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h
- /home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp

## 19.33 Mark3::Streamer Class Reference

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

```
#include <streamer.h>
```

## **Public Member Functions**

void Init (uint8 t \*pau8Buffer , uint16 t u16Size )

Init. Initialize the Streamer object prior to its use, providing a blob of memory for the object to manage.

bool Read (uint8\_t \*pu8Data\_)

Read. Read a byte of data from the stream, if available.

uint16\_t Read (uint8\_t \*pu8Data\_, uint16\_t u16Len\_)

Read. Read from the stream until a specified number of bytes have been read, or the stream is exhausted.

bool Write (uint8\_t u8Data\_)

Write. Write a byte of data into the stream.

• uint16 t Write (uint8 t \*pu8Data , uint16 t u16Len )

Write. Write a maximum number of bytes to the stream.

bool Claim (uint8 t \*\*pu8Addr )

Claim. Claim a byte of data for writing, without actually writing into it. When the writer is ready to write into the data byte as a result of another operation, it can then populate the byte.

void Lock (uint8\_t \*pu8LockAddr\_)

Lock. When the lock is set, a client can neither read from, or write to the buffer at the index specified. This is used to in conjunction with Claim to safely reserve data from the buffer, while preventing race conditions occurring as a result of a consumer acting on the data before it is ready.

void Unlock (void)

Unlock. Reset the lock pointer in the object, allowing a consumer to read any previously unavailable data that might still be in the stream.

• uint16 t GetAvailable (void)

GetAvailable.

· bool CanRead (void)

CanRead.

• bool CanWrite (void)

CanWrite.

· bool IsEmpty (void)

IsEmpty.

#### **Private Attributes**

uint8 t \* m pau8Buffer

Pointer to the buffer managed in this object.

uint8 t \* m pu8LockAddr

Address of the lock point in the stream.

• uint16\_t m\_u16Size

Size of the stream's circular buffer (in bytes)

• uint16\_t m\_u16Avail

Number of bytes free in the stream.

• uint16\_t m\_u16Head

Current head index (write to) of the stream.

uint16\_t m\_u16Tail

Current tail index (read from) of the stream.

## 19.33.1 Detailed Description

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

Definition at line 35 of file streamer.h.

# 19.33.2 Member Function Documentation

## 19.33.2.1 CanRead()

CanRead.

Returns

true if the stream has any unread data

Definition at line 200 of file streamer.cpp.

## 19.33.2.2 CanWrite()

CanWrite.

Returns

true if the stream has any free space

Definition at line 211 of file streamer.cpp.

## 19.33.2.3 Claim()

Claim. Claim a byte of data for writing, without actually writing into it. When the writer is ready to write into the data byte as a result of another operation, it can then populate the byte.

This method is useful when encoding data from a raw format into a packet-based format, where one byte of input may result in multiple bytes of output being generated. Especially in cases where a user wants to write to a stream while a peripheral reads from it to transmisit its contents asynchronously (i.e. an interrupt-driven UART consuming a data packet, where the packet is being framed and consumed simultaneously).

This should be used in conjunction with the Lock method and judicious use of critical sections to prevent race conditions.

pu8⇔	[out] Pointer to a byte pointer that will contain the address of the "claimed" byte on success.
Addr_	

### Returns

true if successful, false if buffer full or locked.

Definition at line 224 of file streamer.cpp.

### 19.33.2.4 GetAvailable()

GetAvailable.

### Returns

The current number of bytes available for write in the streams

Definition at line 134 of file streamer.h.

### 19.33.2.5 Init()

Init. Initialize the Streamer object prior to its use, providing a blob of memory for the object to manage.

# **Parameters**

pau8← Buffer_	Blob of memory to use as a circular buffer
u16Size_	Size of the supplied buffer in bytes

Definition at line 27 of file streamer.cpp.

# 19.33.2.6 IsEmpty()

IsEmpty.

#### Returns

true if the stream is empty

Definition at line 264 of file streamer.cpp.

```
19.33.2.7 Lock()
```

Lock. When the lock is set, a client can neither read from, or write to the buffer at the index specified. This is used to in conjunction with Claim to safely reserve data from the buffer, while preventing race conditions occurring as a result of a consumer acting on the data before it is ready.

### **Parameters**

pu8Lock⊷	Address (within the stream) to set as the lockpoint.
Addr_	

Definition at line 250 of file streamer.cpp.

Read. Read a byte of data from the stream, if available.

### **Parameters**

pu8⇔	Pointer to read data into from the stream
Data	

## Returns

true if data was read, false if data unavailable or buffer locked

Definition at line 38 of file streamer.cpp.

Read. Read from the stream until a specified number of bytes have been read, or the stream is exhausted.

### **Parameters**

pu8← Data	pointer to an array of data read into
u16Len⊷	maximum number of bytes to read

uint16\_t *u16Len\_* )

## Returns

number of bytes read

Definition at line 64 of file streamer.cpp.

### 19.33.2.10 Unlock()

Unlock. Reset the lock pointer in the object, allowing a consumer to read any previously unavailable data that might still be in the stream.

Definition at line 257 of file streamer.cpp.

Write. Write a byte of data into the stream.

### **Parameters**

<i>u8</i> ⇔	Data byte to be written into the stream
Data	

## Returns

true if byte was written, false on buffer full or buffer locked at index.

Definition at line 120 of file streamer.cpp.

Write. Write a maximum number of bytes to the stream.

# **Parameters**

ри8⊷	pointer to an array of bytes to write out to the stream
Data_	
u16Len⊷	Length of data held in the array

#### Returns

number of bytes written to the stream

Definition at line 144 of file streamer.cpp.

# 19.33.3 Member Data Documentation

```
19.33.3.1 m_pau8Buffer
uint8_t* Mark3::Streamer::m_pau8Buffer [private]
```

Pointer to the buffer managed in this object.

Definition at line 154 of file streamer.h.

```
19.33.3.2 m_pu8LockAddr
uint8_t* Mark3::Streamer::m_pu8LockAddr [private]
```

Address of the lock point in the stream.

Definition at line 155 of file streamer.h.

```
19.33.3.3 m_u16Avail
```

```
uint16_t Mark3::Streamer::m_u16Avail [private]
```

Number of bytes free in the stream.

Definition at line 158 of file streamer.h.

## 19.33.3.4 m\_u16Head

```
uint16_t Mark3::Streamer::m_u16Head [private]
```

Current head index (write to) of the stream.

Definition at line 159 of file streamer.h.

## 19.33.3.5 m\_u16Size

```
uint16_t Mark3::Streamer::m_u16Size [private]
```

Size of the stream's circular buffer (in bytes)

Definition at line 157 of file streamer.h.

# 19.33.3.6 m\_u16Tail

```
uint16_t Mark3::Streamer::m_u16Tail [private]
```

Current tail index (read from) of the stream.

Definition at line 160 of file streamer.h.

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

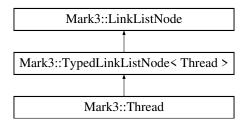
- /home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h
- /home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp

### 19.34 Mark3::Thread Class Reference

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

#include <thread.h>

Inheritance diagram for Mark3::Thread:



#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ∼Thread ()
- Thread ()
- bool IsInitialized ()

IsInitialized Used to check whether or not a thread has been initialized prior to use.

 void Init (K\_WORD \*pwStack\_, uint16\_t u16StackSize\_, PORT\_PRIO\_TYPE uXPriority\_, ThreadEntryFunc pfEntryPoint , void \*pvArg )

Init Initialize a thread prior to its use. Initialized threads are placed in the stopped state, and are not scheduled until the thread's start method has been invoked first.

• void Start ()

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 Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the Start() API.

void SetName (const char \*szName\_)

SetName Set the name of the thread - this is purely optional, but can be useful when identifying issues that come along when multiple threads are at play in a system.

const char \* GetName ()

GetName.

ThreadList \* GetOwner (void)

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

ThreadList \* GetCurrent (void)

GetCurrent Return the ThreadList where the thread is currently located.

PORT\_PRIO\_TYPE GetPriority (void)

GetPriority Return the priority of the current thread.

• PORT PRIO TYPE GetCurPriority (void)

GetCurPriority Return the priority of the current thread.

void SetQuantum (uint16\_t u16Quantum\_)

SetQuantum Set the thread's round-robin execution quantum.

uint16\_t GetQuantum (void)

GetQuantum Get the thread's round-robin execution quantum.

void SetCurrent (ThreadList \*pclNewList\_)

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

void SetOwner (ThreadList \*pclNewList\_)

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

void SetPriority (PORT\_PRIO\_TYPE uXPriority\_)

SetPriority. Set the priority of the Thread (running or otherwise) to a different level. This activity involves re-scheduling, and must be done so with due caution, as it may effect the determinism of the system.

void InheritPriority (PORT\_PRIO\_TYPE uXPriority\_)

InheritPriority Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions. This should only be called from within the implementation of blocking-objects.

void Exit ()

Exit. Remove the thread from being scheduled again. The thread is effectively destroyed when this occurs. This is extremely useful for cases where a thread encounters an unrecoverable error and needs to be restarted, or in the context of systems where threads need to be created and destroyed dynamically.

void SetID (uint8\_t u8ID\_)

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

• uint8 t GetID ()

GetID Return the thread's integer ID. Note that this ID is not guaranteed to be unique when dynamic threading is used in the system, or there are more than 255 threads. Also not guaranteed to be unique if the SetID function is called by the user.

uint16 t GetStackSlack ()

GetStackSlack Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack. If you're having problems with blowing your stack, you can run this function at points in your code during development to see what operations cause problems. Also useful during development as a tool to optimally size thread stacks.

• uint16 t GetEventFlagMask ()

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

void SetEventFlagMask (uint16\_t u16Mask\_)

SetEventFlagMask Sets the active event flag bitfield mask.

void SetEventFlagMode (EventFlagOperation eMode\_)

SetEventFlagMode Sets the active event flag operation mode.

EventFlagOperation GetEventFlagMode ()

GetEventFlagMode Returns the thread's event flag's operating mode.

- Timer \* GetTimer ()
- void SetExpired (bool bExpired\_)

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

bool GetExpired ()

GetExpired Return the status of the most-recent blocking call on the thread.

void \* GetExtendedContext ()

GetExtendedContext Return the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model. Pointer exists only for the lifespan of the Thread.

void SetExtendedContext (void \*pvData\_)

SetExtendedContext Assign the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model.

ThreadState GetState ()

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.

void SetState (ThreadState eState\_)

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.

K WORD \* GetStack ()

GetStack.

uint16\_t GetStackSize ()

GetStackSize.

#### Static Public Member Functions

 static Thread \* Init (uint16\_t u16StackSize\_, PORT\_PRIO\_TYPE uXPriority\_, ThreadEntryFunc pfEntry← Point\_, void \*pvArg\_)

Init Create and initialize a new thread, using memory from the auto-allocated heap region to supply both the thread object and its stack. The thread returned can then be started using the Start() method directly. Note that the memory used to create this thread cannot be reclaimed, and so this API is only suitable for threads that exist for the duration of runtime.

static void Sleep (uint32 t u32TimeMs )

Sleep Put the thread to sleep for the specified time (in milliseconds). Actual time slept may be longer (but not less than) the interval specified.

• static void Yield (void)

Yield Yield the thread - this forces the system to call the scheduler and determine what thread should run next. This is typically used when threads are moved in and out of the scheduler.

static void CoopYield (void)

CoopYield Cooperative yield - This forces the system to not only call the scheduler, but also move the currently executing thread to the back of the current thread list, allowing other same-priority threads the opportunity to run. This is used primarily for cooperative scheduling between threads in the same priority level.

### **Private Member Functions**

void SetPriorityBase (PORT\_PRIO\_TYPE uXPriority\_)
 SetPriorityBase.

### **Static Private Member Functions**

static void ContextSwitchSWI (void)

ContextSwitchSWI This code is used to trigger the context switch interrupt. Called whenever the kernel decides that it is necessary to swap out the current thread for the "next" thread.

## **Private Attributes**

K\_WORD \* m\_pwStackTop

Pointer to the top of the thread's stack.

K WORD \* m pwStack

Pointer to the thread's stack.

uint8\_t m\_u8ThreadID

Thread ID.

PORT\_PRIO\_TYPE m\_uXPriority

Default priority of the thread.

• PORT PRIO TYPE m uXCurPriority

Current priority of the thread (priority inheritence)

ThreadState m\_eState

Enum indicating the thread's current state.

void \* m pvExtendedContext

Pointer provided to a Thread to implement thread-local storage.

const char \* m\_szName

Thread name.

· uint16 t m u16StackSize

Size of the stack (in bytes)

ThreadList \* m\_pclCurrent

Pointer to the thread-list where the thread currently resides.

ThreadList \* m\_pclOwner

Pointer to the thread-list where the thread resides when active.

ThreadEntryFunc m pfEntryPoint

The entry-point function called when the thread starts.

void \* m\_pvArg

Pointer to the argument passed into the thread's entrypoint.

uint16\_t m\_u16Quantum

Thread quantum (in milliseconds)

uint16\_t m\_u16FlagMask

Event-flag mask.

• EventFlagOperation m\_eFlagMode

Event-flag mode.

• Timer m\_clTimer

Timer used for blocking-object timeouts.

bool m bExpired

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

#### **Friends**

class ThreadPort

### **Additional Inherited Members**

## 19.34.1 Detailed Description

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

# **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 64 of file thread.h.

# 19.34.2 Constructor & Destructor Documentation

```
19.34.2.1 \simThread()
```

 $Mark3::Thread::\sim Thread ( )$ 

Definition at line 30 of file thread.cpp.

## 19.34.2.2 Thread()

```
Mark3::Thread::Thread ( ) [inline]
```

Definition at line 70 of file thread.h.

## 19.34.3 Member Function Documentation

### 19.34.3.1 ContextSwitchSWI()

ContextSwitchSWI This code is used to trigger the context switch interrupt. Called whenever the kernel decides that it is necessary to swap out the current thread for the "next" thread.

Definition at line 392 of file thread.cpp.

## 19.34.3.2 CoopYield()

CoopYield Cooperative yield - This forces the system to not only call the scheduler, but also move the currently executing thread to the back of the current thread list, allowing other same-priority threads the opportunity to run. This is used primarily for cooperative scheduling between threads in the same priority level.

Definition at line 325 of file thread.cpp.

### 19.34.3.3 Exit()

```
void Mark3::Thread::Exit ( )
```

Exit. Remove the thread from being scheduled again. The thread is effectively destroyed when this occurs. This is extremely useful for cases where a thread encounters an unrecoverable error and needs to be restarted, or in the context of systems where threads need to be created and destroyed dynamically.

This must not be called on the idle thread.

## **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 187 of file thread.cpp.

### 19.34.3.4 GetCurPriority()

GetCurPriority Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 181 of file thread.h.

## 19.34.3.5 GetCurrent()

GetCurrent Return the ThreadList where the thread is currently located.

Returns

Pointer to the thread's current list

Definition at line 166 of file thread.h.

# 19.34.3.6 GetEventFlagMask()

```
uint16_t Mark3::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 317 of file thread.h.

## 19.34.3.7 GetEventFlagMode()

```
EventFlagOperation Mark3::Thread::GetEventFlagMode ( ) [inline]
```

GetEventFlagMode Returns the thread's event flag's operating mode.

Returns

The thread's event flag mode.

Definition at line 336 of file thread.h.

### 19.34.3.8 GetExpired()

```
bool Mark3::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 425 of file thread.cpp.

## 19.34.3.9 GetExtendedContext()

```
void* Mark3::Thread::GetExtendedContext ( ) [inline]
```

GetExtendedContext Return the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model. Pointer exists only for the lifespan of the Thread.

Returns

Thread's extended context data pointer.

Definition at line 368 of file thread.h.

## 19.34.3.10 GetID()

```
uint8_t Mark3::Thread::GetID ( ) [inline]
```

GetID Return the thread's integer ID. Note that this ID is not guaranteed to be unique when dynamic threading is used in the system, or there are more than 255 threads. Also not guaranteed to be unique if the SetID function is called by the user.

Returns

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

Definition at line 292 of file thread.h.

### 19.34.3.11 GetName()

```
const char* Mark3::Thread::GetName ( ) [inline]
```

GetName.

Returns

Pointer to the name of the thread. If this is not set, will be nullptr.

Definition at line 149 of file thread.h.

## 19.34.3.12 GetOwner()

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

Returns

Pointer to the Thread's owner list

Definition at line 159 of file thread.h.

## 19.34.3.13 GetPriority()

GetPriority Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 174 of file thread.h.

## 19.34.3.14 GetQuantum()

GetQuantum Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 197 of file thread.h.

# 19.34.3.15 GetStack()

```
K_WORD* Mark3::Thread::GetStack ( ) [inline]
```

GetStack.

Returns

Pointer to the blob of memory used as the thread's stack

Definition at line 403 of file thread.h.

# 19.34.3.16 GetStackSize()

```
uint16_t Mark3::Thread::GetStackSize ( ) [inline]
```

GetStackSize.

Returns

Size of the thread's stack in bytes

Definition at line 409 of file thread.h.

### 19.34.3.17 GetStackSlack()

```
uint16_t Mark3::Thread::GetStackSlack ( )
```

GetStackSlack Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack. If you're having problems with blowing your stack, you can run this function at points in your code during development to see what operations cause problems. Also useful during development as a tool to optimally size thread stacks.

#### Returns

The amount of slack (unused bytes) on the stack

### **Examples:**

lab9\_dynamic\_threads/main.cpp.

#### 19.34.3.18 GetState()

```
ThreadState Mark3::Thread::GetState ( ) [inline]
```

GetState Returns the current state of the thread to the caller. Can be used to determine whether or not a thread is ready (or running), stopped, or terminated/exit'd.

#### Returns

ThreadState\_t representing the thread's current state

## **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 389 of file thread.h.

## 19.34.3.19 GetTimer()

```
Timer * Mark3::Thread::GetTimer ( )
```

Return a pointer to the thread's timer object

Definition at line 412 of file thread.cpp.

# 19.34.3.20 InheritPriority()

InheritPriority Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions. This should only be called from within the implementation of blocking-objects.

#### **Parameters**

uX⇔	New Priority to boost to.
Priority_	

Definition at line 383 of file thread.cpp.

Init Initialize a thread prior to its use. Initialized threads are placed in the stopped state, and are not scheduled until the thread's start method has been invoked first.

#### **Parameters**

pwStack_	Pointer to the stack to use for the thread
u16Stack⊷	Size of the stack (in bytes)
Size_	
uXPriority_	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.

# **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 49 of file thread.cpp.

Init Create and initialize a new thread, using memory from the auto-allocated heap region to supply both the thread object and its stack. The thread returned can then be started using the Start() method directly. Note that the memory used to create this thread cannot be reclaimed, and so this API is only suitable for threads that exist for the duration of runtime.

u16Stack⊷	Size of the stack (in bytes)
Size_	
uXPriority_	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.

## Returns

Pointer to a newly-created thread.

Definition at line 101 of file thread.cpp.

## 19.34.3.23 IsInitialized()

IsInitialized Used to check whether or not a thread has been initialized prior to use.

## Returns

return true if the thread object has been initialized, false otherwise.

Definition at line 77 of file thread.h.

## 19.34.3.24 operator new()

Definition at line 67 of file thread.h.

## 19.34.3.25 SetCurrent()

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

#### **Parameters**

pclNew←	Pointer to the threadlist to apply thread ownership
List_	

Definition at line 206 of file thread.h.

## 19.34.3.26 SetEventFlagMask()

```
void Mark3::Thread::SetEventFlagMask ( \label{eq:condition} \mbox{uint16\_t} \ u16\mbox{Mask} \ \ ) \ \ [\mbox{inline}]
```

SetEventFlagMask Sets the active event flag bitfield mask.

## **Parameters**

```
u16⊷
Mask_
```

Definition at line 323 of file thread.h.

# 19.34.3.27 SetEventFlagMode()

SetEventFlagMode Sets the active event flag operation mode.

### **Parameters**

e⊷	Event flag operation mode, defines the logical operator to apply to the event flag.
Mode⊷	
_	

Definition at line 330 of file thread.h.

### 19.34.3.28 SetExpired()

```
void Mark3::Thread::SetExpired (
          bool bExpired_)
```

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

b⇔	true - call expired, false - call did not expire
Expired←	

Definition at line 418 of file thread.cpp.

### 19.34.3.29 SetExtendedContext()

SetExtendedContext Assign the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model.

Object assigned to the context pointer should persist for the duration of the Thread.

#### **Parameters**

pv⊷	Object to assign to the extended data pointer.+
Data_	

Definition at line 380 of file thread.h.

## 19.34.3.30 SetID()

```
void Mark3::Thread::SetID (
          uint8_t u8ID_ ) [inline]
```

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

## **Parameters**

```
u8l← 8-bit Thread ID, set by the user D_
```

Definition at line 282 of file thread.h.

## 19.34.3.31 SetName()

SetName Set the name of the thread - this is purely optional, but can be useful when identifying issues that come along when multiple threads are at play in a system.

SZ⊷	Char string containing the thread name
Name_	

Definition at line 143 of file thread.h.

## 19.34.3.32 SetOwner()

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

### **Parameters**

pclNew⊷	Pointer to the threadlist to apply thread ownership
List_	

Definition at line 213 of file thread.h.

# 19.34.3.33 SetPriority()

SetPriority. Set the priority of the Thread (running or otherwise) to a different level. This activity involves rescheduling, 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**

uX⊷	New priority of the thread
Priority_	

Definition at line 342 of file thread.cpp.

### 19.34.3.34 SetPriorityBase()

SetPriorityBase.

### **Parameters**

```
uX←
Priority_
```

Definition at line 332 of file thread.cpp.

## 19.34.3.35 SetQuantum()

SetQuantum Set the thread's round-robin execution quantum.

### **Parameters**

<i>u</i> 16⇔	Thread's execution quantum (in milliseconds)
Quantum_	

Definition at line 190 of file thread.h.

# 19.34.3.36 SetState()

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**

e⊷	New thread state to set.
State↩	
_	

Definition at line 397 of file thread.h.

## 19.34.3.37 Sleep()

Sleep Put the thread to sleep for the specified time (in milliseconds). Actual time slept may be longer (but not less than) the interval specified.

u32Time⊷	Time to sleep (in ms)
Ms_	

### **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 249 of file thread.cpp.

## 19.34.3.38 Start()

```
void Mark3::Thread::Start (
     void )
```

Start Start the thread - remove it from the stopped list, add it to the scheduler's list of threads (at the thread's set priority), and continue along.

### **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 110 of file thread.cpp.

## 19.34.3.39 Stop()

Stop Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the Start() API.

Definition at line 141 of file thread.cpp.

# 19.34.3.40 Yield()

Yield Yield the thread - this forces the system to call the scheduler and determine what thread should run next. This is typically used when threads are moved in and out of the scheduler.

Definition at line 305 of file thread.cpp.

# 19.34.4 Friends And Related Function Documentation

```
19.34.4.1 ThreadPort
```

```
friend class ThreadPort [friend]
```

Definition at line 411 of file thread.h.

# 19.34.5 Member Data Documentation

```
19.34.5.1 m_bExpired
```

```
bool Mark3::Thread::m_bExpired [private]
```

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

Definition at line 489 of file thread.h.

```
19.34.5.2 m_clTimer
```

```
Timer Mark3::Thread::m_clTimer [private]
```

Timer used for blocking-object timeouts.

Definition at line 486 of file thread.h.

19.34.5.3 m\_eFlagMode

EventFlagOperation Mark3::Thread::m\_eFlagMode [private]

Event-flag mode.

Definition at line 482 of file thread.h.

```
19.34.5.4 m_eState
```

```
ThreadState Mark3::Thread::m_eState [private]
```

Enum indicating the thread's current state.

Definition at line 445 of file thread.h.

19.34.5.5 m\_pclCurrent

```
ThreadList* Mark3::Thread::m_pclCurrent [private]
```

Pointer to the thread-list where the thread currently resides.

Definition at line 461 of file thread.h.

19.34.5.6 m\_pclOwner

```
ThreadList* Mark3::Thread::m_pclOwner [private]
```

Pointer to the thread-list where the thread resides when active.

Definition at line 464 of file thread.h.

19.34.5.7 m\_pfEntryPoint

```
ThreadEntryFunc Mark3::Thread::m_pfEntryPoint [private]
```

The entry-point function called when the thread starts.

Definition at line 467 of file thread.h.

19.34.5.8 m\_pvArg

```
void* Mark3::Thread::m_pvArg [private]
```

Pointer to the argument passed into the thread's entrypoint.

Definition at line 470 of file thread.h.

```
19.34.5.9 m_pvExtendedContext
void* Mark3::Thread::m_pvExtendedContext [private]
Pointer provided to a Thread to implement thread-local storage.
Definition at line 449 of file thread.h.
19.34.5.10 m_pwStack
K_WORD* Mark3::Thread::m_pwStack [private]
Pointer to the thread's stack.
Definition at line 433 of file thread.h.
19.34.5.11 m_pwStackTop
K_WORD* Mark3::Thread::m_pwStackTop [private]
Pointer to the top of the thread's stack.
Definition at line 430 of file thread.h.
19.34.5.12 m_szName
const char* Mark3::Thread::m_szName [private]
Thread name.
Definition at line 454 of file thread.h.
19.34.5.13 m_u16FlagMask
uint16_t Mark3::Thread::m_u16FlagMask [private]
```

Definition at line 479 of file thread.h.

Event-flag mask.

```
19.34.5.14 m_u16Quantum
uint16_t Mark3::Thread::m_u16Quantum [private]
Thread quantum (in milliseconds)
Definition at line 474 of file thread.h.
19.34.5.15 m_u16StackSize
uint16_t Mark3::Thread::m_u16StackSize [private]
Size of the stack (in bytes)
Definition at line 458 of file thread.h.
19.34.5.16 m_u8ThreadID
uint8_t Mark3::Thread::m_u8ThreadID [private]
Thread ID.
Definition at line 436 of file thread.h.
19.34.5.17 m_uXCurPriority
PORT_PRIO_TYPE Mark3::Thread::m_uXCurPriority [private]
Current priority of the thread (priority inheritence)
Definition at line 442 of file thread.h.
19.34.5.18 m_uXPriority
PORT_PRIO_TYPE Mark3::Thread::m_uXPriority [private]
```

Definition at line 439 of file thread.h.

Default priority of the thread.

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

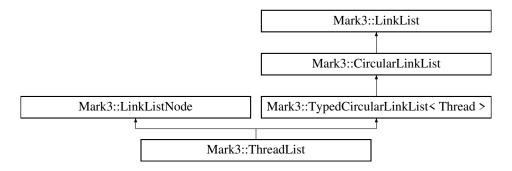
- /home/moslevin/projects/m3-repo/kernel/src/public/thread.h
- /home/moslevin/projects/m3-repo/kernel/src/thread.cpp

## 19.35 Mark3::ThreadList Class Reference

The ThreadList Class. This class is used for building thread-management facilities, such as schedulers, and blocking objects.

#include <threadlist.h>

Inheritance diagram for Mark3::ThreadList:



### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ThreadList ()

ThreadList Default constructor - zero-initializes the data.

void SetPriority (PORT\_PRIO\_TYPE uXPriority\_)

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

void SetMapPointer (PriorityMap \*pclMap\_)

SetMapPointer Set the pointer to a bitmap to use for this threadlist. Once again, only needed when the threadlist is being used for scheduling purposes.

void Add (Thread \*node\_)

Add Add a thread to the threadlist.

void Add (Thread \*node\_, PriorityMap \*pclMap\_, PORT\_PRIO\_TYPE uXPriority\_)

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

void AddPriority (Thread \*node\_)

AddPriority Add a thread to the list such that threads are ordered from highest to lowest priority from the head of the list

void Remove (Thread \*node\_)

Remove Remove the specified thread from the threadlist.

• Thread \* HighestWaiter ()

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

## **Private Attributes**

PORT\_PRIO\_TYPE m\_uXPriority

Priority of the threadlist.

PriorityMap \* m\_pclMap

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

**Additional Inherited Members** 

# 19.35.1 Detailed Description

The ThreadList Class. This class is used for building thread-management facilities, such as schedulers, and blocking objects.

Definition at line 38 of file threadlist.h.

## 19.35.2 Constructor & Destructor Documentation

```
19.35.2.1 ThreadList()
```

```
Mark3::ThreadList::ThreadList ( )
```

ThreadList Default constructor - zero-initializes the data.

Definition at line 27 of file threadlist.cpp.

## 19.35.3 Member Function Documentation

```
19.35.3.1 Add() [1/2]
```

Add Add a thread to the threadlist.

### **Parameters**

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

Definition at line 47 of file threadlist.cpp.

```
19.35.3.2 Add() [2/2]
```

```
PriorityMap * pclMap_,
PORT_PRIO_TYPE uXPriority_ )
```

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

node_	Pointer to the thread to add (link list node)
pclMap_	Pointer to the bitmap flag to set (if used in a scheduler context), or nullptr for non-scheduler.
uX⇔	Priority of the threadlist
Priority_	

Definition at line 101 of file threadlist.cpp.

## 19.35.3.3 AddPriority()

AddPriority Add a thread to the list such that threads are ordered from highest to lowest priority from the head of the list.

#### **Parameters**

node⊷	Pointer to a thread to add to the list.

Definition at line 66 of file threadlist.cpp.

## 19.35.3.4 HighestWaiter()

```
Thread * Mark3::ThreadList::HighestWaiter ( )
```

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

## Returns

Pointer to the highest-priority thread

Definition at line 128 of file threadlist.cpp.

### 19.35.3.5 operator new()

Definition at line 41 of file threadlist.h.

# 19.35.3.6 Remove()

Remove Remove the specified thread from the threadlist.

noue←	Pointer to the thread to remove

Definition at line 111 of file threadlist.cpp.

## 19.35.3.7 SetMapPointer()

SetMapPointer 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**

pcl←	Pointer to the priority map object used to track this thread.
Мар_	

Definition at line 40 of file threadlist.cpp.

## 19.35.3.8 SetPriority()

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

#### **Parameters**

uX⊷	Priority level of the thread list
Priority_	

Definition at line 34 of file threadlist.cpp.

# 19.35.4 Member Data Documentation

## 19.35.4.1 m\_pclMap

```
PriorityMap* Mark3::ThreadList::m_pclMap [private]
```

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

Definition at line 116 of file threadlist.h.

```
19.35.4.2 m_uXPriority
```

```
PORT_PRIO_TYPE Mark3::ThreadList::m_uXPriority [private]
```

Priority of the threadlist.

Definition at line 113 of file threadlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h
- /home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp

## 19.36 Mark3::ThreadListList Class Reference

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

```
#include <threadlistlist.h>
```

### Static Public Member Functions

- static void Add (ThreadList \*pclThreadList\_)
  - Add Add a ThreadList to the list for tracking.
- static void Remove (ThreadList \*pclThreadList\_)

Remove Remove a threadlist from tracking.

static ThreadList \* GetHead ()
 GetHead.

## **Static Private Attributes**

• static TypedDoubleLinkList< ThreadList > m\_clThreadListList

# 19.36.1 Detailed Description

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

Definition at line 36 of file threadlistlist.h.

#### 19.36.2 Member Function Documentation

```
19.36.2.1 Add()
```

Add Add a ThreadList to the list for tracking.

pclThread↩	threadlist to add for tracking
List_	

Definition at line 43 of file threadlistlist.h.

### 19.36.2.2 GetHead()

```
static ThreadList* Mark3::ThreadListList::GetHead ( ) [inline], [static]
```

GetHead.

#### Returns

The threadlist at the beginning of the list

Definition at line 62 of file threadlistlist.h.

## 19.36.2.3 Remove()

Remove Remove a threadlist from tracking.

## **Parameters**

pclThread←	threadlist to remove from tracking
List_	

Definition at line 53 of file threadlistlist.h.

## 19.36.3 Member Data Documentation

# 19.36.3.1 m\_clThreadListList

```
\label{thm:continuity} \textbf{TypedDoubleLinkList} < \textbf{ThreadList} > \texttt{Mark3::ThreadListList::m\_clThreadListList} \quad [\texttt{static}] \text{, [private]}
```

Definition at line 68 of file threadlistlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h
- /home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp

## 19.37 Mark3::ThreadPort Class Reference

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

```
#include <ithreadport.h>
```

#### Static Public Member Functions

• static void Init ()

Init Function to perform early init of the target environment prior to using OS primatives.

• static void StartThreads ()

StartThreads Function to start the scheduler, initial threads, etc.

### **Static Private Member Functions**

static void InitStack (Thread \*pstThread\_)
 InitStack Initialize the thread's stack.

### **Friends**

· class Thread

# 19.37.1 Detailed Description

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

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 35 of file ithreadport.h.

## 19.37.2 Member Function Documentation

```
19.37.2.1 Init()
```

Init Function to perform early init of the target environment prior to using OS primatives.

Definition at line 43 of file ithreadport.h.

## 19.37.2.2 InitStack()

InitStack Initialize the thread's stack.

#### **Parameters**

pst⇔	Pointer to the thread to initialize
Thread_	

Definition at line 43 of file threadport.cpp.

#### 19.37.2.3 StartThreads()

```
void Mark3::ThreadPort::StartThreads ( ) [static]
```

StartThreads Function to start the scheduler, initial threads, etc.

Definition at line 100 of file threadport.cpp.

#### 19.37.3 Friends And Related Function Documentation

#### 19.37.3.1 Thread

```
friend class Thread [friend]
```

Definition at line 50 of file ithreadport.h.

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

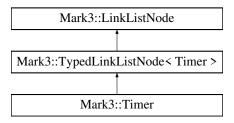
- /home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp

# 19.38 Mark3::Timer Class Reference

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

```
#include <timer.h>
```

Inheritance diagram for Mark3::Timer:



#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- ~Timer ()
- Timer ()

Timer Default Constructor - Do nothing. Allow the init call to perform the necessary object initialization prior to use.

• void Init ()

Init Re-initialize the Timer to default values.

void Start (bool bRepeat\_, uint32\_t u32IntervalMs\_, TimerCallback pfCallback\_, void \*pvData\_)

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

• void Start ()

Start Start or restart a timer using parameters previously configured via calls to Start(<with args>), or via the a-lacarte parameter setter methods. This is especially useful for retriggering one-shot timers that have previously expired, using the timer's previous configuration.

• void Stop ()

Stop Stop a timer already in progress. Has no effect on timers that have already been stopped.

#### **Private Member Functions**

· void SetInitialized ()

SetInitialized.

• bool IsInitialized (void)

IsInitialized.

#### **Static Private Member Functions**

- static uint32 t SecondsToTicks (uint32 t x)
- static uint32 t MSecondsToTicks (uint32 t x)
- static uint32\_t USecondsToTicks (uint32\_t x)

#### **Private Attributes**

uint8\_t m\_u8Initialized

Cookie used to determine whether or not the timer is initialized.

uint8\_t m\_u8Flags

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

• TimerCallback m\_pfCallback

Pointer to the callback function.

• uint32 t m u32Interval

Interval of the timer in timer ticks.

• uint32\_t m\_u32TimeLeft

Time remaining on the timer.

Thread \* m\_pclOwner

Pointer to the owner thread.

void \* m\_pvData

Pointer to the callback data.

# **Static Private Attributes**

- static constexpr auto m\_uTimerInvalidCookie = uint8\_t { 0x3C }
- static constexpr auto m\_uTimerInitCookie = uint8\_t { 0xC3 }

#### **Friends**

· class TimerList

#### **Additional Inherited Members**

# 19.38.1 Detailed Description

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

#### **Examples:**

lab6\_timers/main.cpp.

Definition at line 68 of file timer.h.

#### 19.38.2 Constructor & Destructor Documentation

```
19.38.2.1 \simTimer()
```

Mark3::Timer::~Timer ( ) [inline]

Definition at line 72 of file timer.h.

```
19.38.2.2 Timer()
```

```
Mark3::Timer::Timer ( )
```

Timer Default Constructor - Do nothing. Allow the init call to perform the necessary object initialization prior to use.

Definition at line 29 of file timer.cpp.

# 19.38.3 Member Function Documentation

Init Re-initialize the Timer to default values.

Definition at line 36 of file timer.cpp.

# 19.38.3.2 IsInitialized()

IsInitialized.

Returns

true if initialized, false if not initialized.

Definition at line 127 of file timer.h.

# 19.38.3.3 MSecondsToTicks()

Definition at line 130 of file timer.h.

# 19.38.3.4 operator new()

Definition at line 71 of file timer.h.

#### 19.38.3.5 SecondsToTicks()

Definition at line 129 of file timer.h.

# 19.38.3.6 SetInitialized()

SetInitialized.

Definition at line 121 of file timer.h.

```
19.38.3.7 Start() [1/2]
```

```
void Mark3::Timer::Start (
          bool bRepeat_,
          uint32_t u32IntervalMs_,
          TimerCallback pfCallback_,
          void * pvData_ )
```

Start 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.
u32Interval← Ms_	- Interval of the timer in miliseconds
pfCallback_	- Function to call on timer expiry
pvData_	- Data to pass into the callback function

# **Examples:**

lab6\_timers/main.cpp.

Definition at line 51 of file timer.cpp.

```
19.38.3.8 Start() [2/2]
```

Start Start or restart a timer using parameters previously configured via calls to Start(<with args>), or via the ala-carte parameter setter methods. This is especially useful for retriggering one-shot timers that have previously expired, using the timer's previous configuration.

Definition at line 73 of file timer.cpp.

```
19.38.3.9 Stop()
```

Stop Stop a timer already in progress. Has no effect on timers that have already been stopped.

Definition at line 86 of file timer.cpp.

# 19.38.3.10 USecondsToTicks()

Definition at line 131 of file timer.h.

# 19.38.4 Friends And Related Function Documentation

```
19.38.4.1 TimerList
```

```
friend class TimerList [friend]
```

Definition at line 116 of file timer.h.

## 19.38.5 Member Data Documentation

```
19.38.5.1 m_pclOwner
```

```
Thread* Mark3::Timer::m_pclOwner [private]
```

Pointer to the owner thread.

Definition at line 152 of file timer.h.

```
19.38.5.2 m_pfCallback
TimerCallback Mark3::Timer::m_pfCallback [private]
Pointer to the callback function.
Definition at line 143 of file timer.h.
19.38.5.3 m_pvData
void* Mark3::Timer::m_pvData [private]
Pointer to the callback data.
Definition at line 155 of file timer.h.
19.38.5.4 m_u32Interval
uint32_t Mark3::Timer::m_u32Interval [private]
Interval of the timer in timer ticks.
Definition at line 146 of file timer.h.
19.38.5.5 m_u32TimeLeft
uint32_t Mark3::Timer::m_u32TimeLeft [private]
Time remaining on the timer.
Definition at line 149 of file timer.h.
19.38.5.6 m_u8Flags
```

Generated by Doxygen

uint8\_t Mark3::Timer::m\_u8Flags [private]

Definition at line 140 of file timer.h.

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

#### 19.38.5.7 m\_u8Initialized

```
uint8_t Mark3::Timer::m_u8Initialized [private]
```

Cookie used to determine whether or not the timer is initialized.

Definition at line 137 of file timer.h.

#### 19.38.5.8 m\_uTimerInitCookie

```
constexpr auto Mark3::Timer::m_uTimerInitCookie = uint8_t { 0xC3 } [static], [private]
```

Definition at line 134 of file timer.h.

# 19.38.5.9 m\_uTimerInvalidCookie

```
constexpr auto Mark3::Timer::m_uTimerInvalidCookie = uint8_t { 0x3C } [static], [private]
```

Definition at line 133 of file timer.h.

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

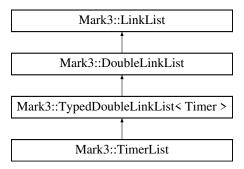
- /home/moslevin/projects/m3-repo/kernel/src/public/timer.h
- /home/moslevin/projects/m3-repo/kernel/src/timer.cpp

# 19.39 Mark3::TimerList Class Reference

the TimerList class. This class implements a doubly-linked-list of timer objects.

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

Inheritance diagram for Mark3::TimerList:



# **Public Member Functions**

• void Init ()

Init Initialize the TimerList object. Must be called before using the object.

void Add (Timer \*pclListNode\_)

Add Add a timer to the TimerList.

void Remove (Timer \*pclLinkListNode\_)

Remove Remove a timer from the TimerList, cancelling its expiry.

· void Process ()

Process Process all timers in the timerlist as a result of the timer expiring. This will select a new timer epoch based on the next timer to expire.

#### **Private Attributes**

• uint32\_t m\_u32NextWakeup

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

• bool m\_bTimerActive

Whether or not the timer is active.

• Mutex m\_clMutex

Guards against concurrent access to the timer list - Only needed when running threaded.

# **Additional Inherited Members**

# 19.39.1 Detailed Description

the TimerList class. This class implements a doubly-linked-list of timer objects.

Definition at line 40 of file timerlist.h.

# 19.39.2 Member Function Documentation

```
19.39.2.1 Add()
```

Add Add a timer to the TimerList.

#### **Parameters**

pclList⊷	Pointer to the Timer to Add
Node	

Definition at line 35 of file timerlist.cpp.

#### 19.39.2.2 Init()

Init Initialize the TimerList object. Must be called before using the object.

Definition at line 27 of file timerlist.cpp.

#### 19.39.2.3 Process()

Process Process all timers in the timerlist as a result of the timer expiring. This will select a new timer epoch based on the next timer to expire.

Definition at line 61 of file timerlist.cpp.

#### 19.39.2.4 Remove()

Remove Remove a timer from the TimerList, cancelling its expiry.

#### **Parameters**

pclLinkList←	Pointer to the Timer to remove
Node_	

Definition at line 51 of file timerlist.cpp.

# 19.39.3 Member Data Documentation

# 19.39.3.1 m\_bTimerActive

```
bool Mark3::TimerList::m_bTimerActive [private]
```

Whether or not the timer is active.

Definition at line 78 of file timerlist.h.

19.39.3.2 m\_clMutex

```
Mutex Mark3::TimerList::m_clMutex [private]
```

Guards against concurrent access to the timer list - Only needed when running threaded.

Definition at line 81 of file timerlist.h.

19.39.3.3 m\_u32NextWakeup

```
uint32_t Mark3::TimerList::m_u32NextWakeup [private]
```

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

Definition at line 75 of file timerlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h
- /home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp

# 19.40 Mark3::TimerScheduler Class Reference

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

```
#include <timerscheduler.h>
```

#### **Static Public Member Functions**

• static void Init ()

Init Initialize the timer scheduler. Must be called before any timer, or timer-derived functions are used.

static void Add (Timer \*pclListNode\_)

Add Add a timer to the timer scheduler. Adding a timer implicitly starts the timer as well.

static void Remove (Timer \*pclListNode\_)

Remove Remove a timer from the timer scheduler. May implicitly stop the timer if this is the only active timer scheduled.

• static void Process ()

Process This function must be called on timer expiry (from the timer's ISR context). This will result in all timers being updated based on the epoch that just elapsed. The next timer epoch is set based on the next Timer object to expire.

#### **Static Private Attributes**

• static TimerList m\_clTimerList

TimerList object manipu32ated by the Timer Scheduler.

# 19.40.1 Detailed Description

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

Definition at line 38 of file timerscheduler.h.

#### 19.40.2 Member Function Documentation

```
19.40.2.1 Add()
```

Add Add a timer to the timer scheduler. Adding a timer implicitly starts the timer as well.

#### **Parameters**

pclList←	Pointer to the timer list node to add
Node_	

Definition at line 54 of file timerscheduler.h.

```
19.40.2.2 Init()
```

Init Initialize the timer scheduler. Must be called before any timer, or timer-derived functions are used.

Definition at line 46 of file timerscheduler.h.

## 19.40.2.3 Process()

Process This function must be called on timer expiry (from the timer's ISR context). This will result in all timers being updated based on the epoch that just elapsed. The next timer epoch is set based on the next Timer object to expire.

Definition at line 70 of file timerscheduler.h.

#### 19.40.2.4 Remove()

Remove Remove a timer from the timer scheduler. May implicitly stop the timer if this is the only active timer scheduled.

#### **Parameters**

pclList⇔	Pointer to the timer list node to remove
Node_	

Definition at line 62 of file timerscheduler.h.

# 19.40.3 Member Data Documentation

## 19.40.3.1 m\_clTimerList

```
TimerList Mark3::TimerScheduler::m_clTimerList [static], [private]
```

TimerList object manipu32ated by the Timer Scheduler.

Definition at line 74 of file timerscheduler.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/timerscheduler.h
- /home/moslevin/projects/m3-repo/kernel/src/timer.cpp

# 19.41 Mark3::Token\_t Struct Reference

Token descriptor struct format.

```
#include <memutil.h>
```

# **Public Attributes**

const char \* pcToken

Pointer to the beginning of the token string.

• uint8\_t u8Len

Length of the token (in bytes)

# 19.41.1 Detailed Description

Token descriptor struct format.

Definition at line 32 of file memutil.h.

# 19.41.2 Member Data Documentation

```
19.41.2.1 pcToken
```

```
const char* Mark3::Token_t::pcToken
```

Pointer to the beginning of the token string.

Definition at line 33 of file memutil.h.

#### 19.41.2.2 u8Len

```
uint8_t Mark3::Token_t::u8Len
```

Length of the token (in bytes)

Definition at line 34 of file memutil.h.

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

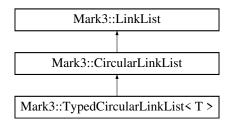
• /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h

# 19.42 Mark3::TypedCircularLinkList< T > Class Template Reference

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

```
#include <11.h>
```

Inheritance diagram for Mark3::TypedCircularLinkList< T >:



#### **Public Member Functions**

- void \* operator new (size\_t sz, void \*pv)
- TypedCircularLinkList ()
- T \* GetHead ()

GetHead Get the head node in the linked list.

void SetHead (T \*pclNode\_)

SetHead Set the head node of a linked list.

T \* GetTail ()

GetTail Get the tail node of the linked list.

void SetTail (T \*pclNode\_)

SetTail Set the tail node of the linked list.

void Add (T \*pNode\_)

Add the linked list node to this linked list.

void Remove (T \*pNode\_)

Remove Add the linked list node to this linked list.

void InsertNodeBefore (T \*pNode\_, T \*pInsert\_)

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

#### **Additional Inherited Members**

# 19.42.1 Detailed Description

```
template<typename T> class Mark3::TypedCircularLinkList< T>
```

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

Definition at line 347 of file II.h.

# 19.42.2 Constructor & Destructor Documentation

# 19.42.2.1 TypedCircularLinkList()

```
template<typename T>
Mark3::TypedCircularLinkList< T >::TypedCircularLinkList ( ) [inline]
```

Definition at line 352 of file II.h.

#### 19.42.3 Member Function Documentation

```
19.42.3.1 Add()
```

Add the linked list node to this linked list.

#### **Parameters**

node↩	Pointer to the node to add
_	

Definition at line 394 of file II.h.

# 19.42.3.2 GetHead()

```
template<typename T>
T* Mark3::TypedCircularLinkList< T >::GetHead ( ) [inline]
```

GetHead Get the head node in the linked list.

#### Returns

Pointer to the head node in the list

Definition at line 362 of file II.h.

#### 19.42.3.3 GetTail()

```
template<typename T>
T* Mark3::TypedCircularLinkList< T >::GetTail ( ) [inline]
```

GetTail Get the tail node of the linked list.

#### Returns

Pointer to the tail node in the list

Definition at line 378 of file II.h.

# 19.42.3.4 InsertNodeBefore()

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

#### **Parameters**

node⊷	Node to insert into the list
_	
insert⊷	Insert point.
_	

Definition at line 418 of file II.h.

#### 19.42.3.5 operator new()

Definition at line 350 of file II.h.

#### 19.42.3.6 Remove()

Remove Add the linked list node to this linked list.

# **Parameters**

node⊷	Pointer to the node to remove

Definition at line 405 of file II.h.

# 19.42.3.7 SetHead()

SetHead Set the head node of a linked list.

#### **Parameters**

pcl←	Pointer to node to set as the head of the linked list
Node_	

Definition at line 370 of file II.h.

#### 19.42.3.8 SetTail()

SetTail Set the tail node of the linked list.

#### **Parameters**

pcl←	Pointer to the node to set as the tail of the linked list
Node_	

Definition at line 386 of file II.h.

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

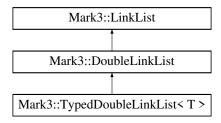
• /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

# 19.43 Mark3::TypedDoubleLinkList < T > Class Template Reference

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

```
#include <11.h>
```

Inheritance diagram for Mark3::TypedDoubleLinkList< T >:



#### **Public Member Functions**

```
    void * operator new (size t sz, void *pv)
```

- TypedDoubleLinkList ()
- T \* GetHead ()

GetHead Get the head node in the linked list.

void SetHead (T \*pclNode )

SetHead Set the head node of a linked list.

T \* GetTail ()

GetTail Get the tail node of the linked list.

void SetTail (T \*pclNode\_)

SetTail Set the tail node of the linked list.

void Add (T \*pNode\_)

Add Add the linked list node to this linked list.

void Remove (T \*pNode\_)

Remove Add the linked list node to this linked list.

#### **Additional Inherited Members**

# 19.43.1 Detailed Description

```
\label{template} \begin{tabular}{ll} template < typename T > \\ class Mark3::TypedDoubleLinkList < T > \\ \end{tabular}
```

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

Definition at line 276 of file II.h.

#### 19.43.2 Constructor & Destructor Documentation

## 19.43.2.1 TypedDoubleLinkList()

```
template<typename T>
Mark3::TypedDoubleLinkList< T >::TypedDoubleLinkList ( ) [inline]
```

Definition at line 281 of file II.h.

#### 19.43.3 Member Function Documentation

#### 19.43.3.1 Add()

Add Add the linked list node to this linked list.

#### **Parameters**

node↩	Pointer to the node to add

Definition at line 323 of file II.h.

#### 19.43.3.2 GetHead()

```
template<typename T>
T* Mark3::TypedDoubleLinkList< T >::GetHead ( ) [inline]
```

GetHead Get the head node in the linked list.

Returns

Pointer to the head node in the list

Definition at line 291 of file II.h.

# 19.43.3.3 GetTail()

```
template<typename T>
T* Mark3::TypedDoubleLinkList< T >::GetTail ( ) [inline]
```

GetTail Get the tail node of the linked list.

Returns

Pointer to the tail node in the list

Definition at line 307 of file II.h.

#### 19.43.3.4 operator new()

Definition at line 279 of file II.h.

# 19.43.3.5 Remove()

Remove Add the linked list node to this linked list.

#### **Parameters**

noue←	Pointer to the node to remove

Definition at line 334 of file II.h.

#### 19.43.3.6 SetHead()

SetHead Set the head node of a linked list.

#### **Parameters**

pcl⇔	Pointer to node to set as the head of the linked list	
Node_		

Definition at line 299 of file II.h.

# 19.43.3.7 SetTail()

SetTail Set the tail node of the linked list.

# **Parameters**

pcl←	Pointer to the node to set as the tail of the linked list	]
Node_		

Definition at line 315 of file II.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

# 19.44 Mark3::TypedLinkListNode < T > Class Template Reference

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

```
#include <11.h>
```

Inheritance diagram for Mark3::TypedLinkListNode< T >:

```
Mark3::LinkListNode

Mark3::TypedLinkListNode < T >
```

# **Public Member Functions**

- T \* GetNext ()
- T \* GetPrev ()

#### **Additional Inherited Members**

# 19.44.1 Detailed Description

```
\label{template} \mbox{template} < \mbox{typename T} > \\ \mbox{class Mark3::TypedLinkListNode} < \mbox{T} > \\ \mbox{}
```

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

Definition at line 108 of file II.h.

#### 19.44.2 Member Function Documentation

#### 19.44.2.1 GetNext()

Definition at line 111 of file II.h.

#### 19.44.2.2 GetPrev()

Definition at line 112 of file II.h.

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

• /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

# **Chapter 20**

# **File Documentation**

# 20.1 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference

Implementation of C-language wrappers for the Mark3 kernel.

```
#include "mark3c.h"
#include "mark3.h"
```

#### **Functions**

```
void * Alloc_Memory (uint16_t u16Size_)
```

void Free\_Memory (void \*pvObject\_)

Free\_Memory.

• Semaphore\_t Alloc\_Semaphore (void)

Alloc\_Semaphore.

- void Free\_Semaphore (Semaphore\_t handle)
- Mutex\_t Alloc\_Mutex (void)

Alloc\_Mutex.

- void Free\_Mutex (Mutex\_t handle)
- EventFlag\_t Alloc\_EventFlag (void)
- void Free\_EventFlag (EventFlag\_t handle)
- Message\_t Alloc\_Message (void)

Alloc\_Message.

- void Free Message (Message t handle)
- MessageQueue\_t Alloc\_MessageQueue (void)

Alloc\_MessageQueue.

- void Free\_MessageQueue (MessageQueue\_t handle)
- MessagePool\_t Alloc\_MessagePool (void)
- void Free\_MessagePool (MessagePool\_t handle)
- Notify\_t Alloc\_Notify (void)

Alloc\_Notify.

- void Free\_Notify (Notify\_t handle)
- Mailbox t Alloc Mailbox (void)

Alloc\_Mailbox.

void Free\_Mailbox (Mailbox\_t handle)

ConditionVariable\_t Alloc\_ConditionVariable (void)

```
    void Free_ConditionVariable (ConditionVariable_t handle)

    ReaderWriterLock_t Alloc_ReaderWriterLock (void)

    void Free ReaderWriterLock (ReaderWriterLock t handle)

    Thread_t Alloc_Thread (void)

     Alloc_Thread.

    void Free_Thread (Thread_t handle)

    Timer_t Alloc_Timer (void)

     Alloc Timer.

    void Free_Timer (Timer_t handle)

    void Kernel_Init (void)

     Kernel_Init.

    void Kernel_Start (void)

     Kernel_Start.

    bool Kernel_IsStarted (void)

     Kernel_IsStarted.
· void Kernel SetPanic (PanicFunc pfPanic )
     Kernel_SetPanic.

    bool Kernel_IsPanic (void)

     Kernel IsPanic.

    void Kernel Panic (uint16 t u16Cause )

     Kernel Panic.

    uint32_t Kernel_GetTicks (void)

     Kernel GetTicks.

    void Kernel_SetStackGuardThreshold (uint16_t u16Threshold_)

    uint16 t Kernel GetStackGuardThreshold (void)

    void Kernel_SetDebugPrintFunction (kernel_debug_print_t pfPrintFunction_)

      Kernel_SetDebugPrintFunction.

    void Kernel DebugPrint (const char *szString )

     KernelDebug_DebugPrint.

    void Scheduler_Enable (bool bEnable_)

     Scheduler_Enable.

    bool Scheduler IsEnabled (void)

     Scheduler_IsEnabled.

    Thread_t Scheduler_GetCurrentThread (void)

     Scheduler GetCurrentThread.

    void Thread Init (Thread t handle, K WORD *pwStack , uint16 t u16StackSize , PORT PRIO TYPE u←

  XPriority_, ThreadEntryFunc pfEntryPoint_, void *pvArg_)
      Thread_Init.
· void Thread_Start (Thread_t handle)
      Thread Start.

    void Thread Stop (Thread t handle)

      Thread_Stop.

    void Thread_SetName (Thread_t handle, const char *szName_)

    const char * Thread GetName (Thread t handle)

    PORT PRIO TYPE Thread GetPriority (Thread t handle)

      Thread_GetPriority.

    PORT_PRIO_TYPE Thread_GetCurPriority (Thread_t handle)

      Thread GetCurPriority.

    void Thread SetQuantum (Thread t handle, uint16 t u16Quantum )

    uint16 t Thread GetQuantum (Thread t handle)

    void Thread_SetPriority (Thread_t handle, PORT_PRIO_TYPE uXPriority_)
```

```
Thread_SetPriority.

    void Thread_Exit (Thread_t handle)

     Thread Exit.

    void Thread_Sleep (uint32_t u32TimeMs_)

     Thread_Sleep.

    void Thread_Yield (void)

     Thread Yield.

    void Thread CoopYield (void)

     Thread_CoopYield.

    void Thread_SetID (Thread_t handle, uint8_t u8ID_)

     Thread SetID.
• uint8_t Thread_GetID (Thread_t handle)
     Thread_GetID.

    uint16_t Thread_GetStackSlack (Thread_t handle)

• thread state t Thread GetState (Thread t handle)
     Thread_GetState.

    void Timer_Init (Timer_t handle)

     Timer_Init.
· void Timer Start (Timer t handle, bool bRepeat, uint32 t u32IntervalMs, timer callback t pfCallback,
  void *pvData )
     Timer_Start.

    void Timer_Stop (Timer_t handle)

     Timer Stop.

    void Timer Restart (Timer t handle)

     Timer Restart.

    void Semaphore_Init (Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_)

     Semaphore Init.

    void Semaphore_Post (Semaphore_t handle)

     Semaphore_Post.
• void Semaphore_Pend (Semaphore_t handle)
     Semaphore Pend.

    bool Semaphore_TimedPend (Semaphore_t handle, uint32_t u32WaitTimeMS_)

     Semaphore_TimedPend.

    void Mutex_Init (Mutex_t handle)

     Mutex_Init.

    void Mutex_Claim (Mutex_t handle)

     Mutex_Claim.
· void Mutex Release (Mutex t handle)
     Mutex_Release.

    bool Mutex_TimedClaim (Mutex_t handle, uint32_t u32WaitTimeMS_)

     Mutex_TimedClaim.

    void Notify_Init (Notify_t handle)

     Notify_Init.

    void Notify_Signal (Notify_t handle)

     Notify_Signal.

    void Notify_Wait (Notify_t handle, bool *pbFlag_)

• bool Notify_TimedWait (Notify_t handle, uint32_t u32WaitTimeMS_, bool *pbFlag_)
     Notify_TimedWait.
• uint8_t Atomic_Set8 (uint8_t *pu8Source_, uint8_t u8Val_)
     Atomic_Set8.
```

```
uint16_t Atomic_Set16 (uint16_t *pu16Source_, uint16_t u16Val_)
     Atomic_Set16.

    uint32 t Atomic Set32 (uint32 t *pu32Source , uint32 t u32Val )

     Atomic_Set32.

    uint8_t Atomic_Add8 (uint8_t *pu8Source_, uint8_t u8Val_)

     Atomic_Add8.
• uint16_t Atomic_Add16 (uint16_t *pu16Source_, uint16_t u16Val_)
     Atomic_Add16.
uint32_t Atomic_Add32 (uint32_t *pu32Source_, uint32_t u32Val_)
     Atomic Add32.
• uint8 t Atomic Sub8 (uint8 t *pu8Source , uint8 t u8Val )
     Atomic_Sub8.
uint16_t Atomic_Sub16 (uint16_t *pu16Source_, uint16_t u16Val_)
     Atomic_Sub16.

    uint32 t Atomic Sub32 (uint32 t *pu32Source , uint32 t u32Val )

     Atomic_Sub32.

    bool Atomic_TestAndSet (bool *pbLock)

     Atomic TestAndSet.

    void Message_Init (Message_t handle)

     Message_Init.

    void Message_SetData (Message_t handle, void *pvData_)

     Message SetData.

    void * Message_GetData (Message_t handle)

     Message_GetData.

    void Message_SetCode (Message_t handle, uint16_t u16Code_)

     Message SetCode.

    uint16_t Message_GetCode (Message_t handle)

     Message_GetCode.

    void MessageQueue_Init (MessageQueue_t handle)

     MessageQueue Init.

    Message_t MessageQueue_Receive (MessageQueue_t handle)

     MessageQueue_Receive.

    void MessagePool_Init (MessagePool_t handle)

     MessagePool Init.

    void MessagePool_Push (MessagePool_t handle, Message_t msg)

     MessagePool_Push.

    Message t MessagePool Pop (MessagePool t handle)

     MessagePool Pop.

    Message_t MessageQueue_TimedReceive (MessageQueue_t handle, uint32_t u32TimeWaitMS_)

     MessageQueue_TimedReceive.

    void MessageQueue Send (MessageQueue t handle, Message t hMessage)

     MessageQueue Send.

    uint16_t MessageQueue_GetCount (MessageQueue_t handle)

     MessageQueue_GetCount.
• void Mailbox_Init (Mailbox_t handle, void *pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize )
     Mailbox Init.

    bool Mailbox_Send (Mailbox_t handle, void *pvData_)

     Mailbox_Send.

    bool Mailbox SendTail (Mailbox t handle, void *pvData )

     Mailbox SendTail.

    bool Mailbox_TimedSend (Mailbox_t handle, void *pvData_, uint32_t u32TimeoutMS_)
```

Mailbox\_TimedSend.

• bool Mailbox\_TimedSendTail (Mailbox\_t handle, void \*pvData\_, uint32\_t u32TimeoutMS\_)

Mailbox TimedSendTail.

void Mailbox\_Receive (Mailbox\_t handle, void \*pvData\_)

Mailbox\_Receive.

void Mailbox\_ReceiveTail (Mailbox\_t handle, void \*pvData\_)

Mailbox\_ReceiveTail.

• bool Mailbox\_TimedReceive (Mailbox\_t handle, void \*pvData\_, uint32\_t u32TimeoutMS\_)

Mailbox TimedReceive.

bool Mailbox TimedReceiveTail (Mailbox t handle, void \*pvData , uint32 t u32TimeoutMS )

Mailbox\_TimedReceiveTail.

• uint16\_t Mailbox\_GetFreeSlots (Mailbox\_t handle)

Mailbox GetFreeSlots.

• bool Mailbox\_lsFull (Mailbox\_t handle)

Mailbox\_IsFull.

bool Mailbox IsEmpty (Mailbox t handle)

Mailbox\_IsEmpty.

void ConditionVariable\_Init (ConditionVariable\_t handle)

ConditionVariable Init.

void ConditionVariable\_Wait (ConditionVariable\_t handle, Mutex\_t hMutex\_)

ConditionVariable\_Wait.

void ConditionVariable\_Signal (ConditionVariable\_t handle)

ConditionVariable\_Signal.

void ConditionVariable Broadcast (ConditionVariable t handle)

ConditionVariable\_Broadcast.

bool ConditionVariable\_TimedWait (ConditionVariable\_t handle, Mutex\_t hMutex\_, uint32\_t u32WaitTime

 MS )

ConditionVariable\_TimedWait.

void ReaderWriterLock\_Init (ReaderWriterLock\_t handle)

ReaderWriterLock Init.

void ReaderWriterLock\_AcquireReader (ReaderWriterLock\_t handle)

ReaderWriterLock\_AcquireReader.

void ReaderWriterLock ReleaseReader (ReaderWriterLock t handle)

ReaderWriterLock\_ReleaseReader.

void ReaderWriterLock\_AcquireWriter (ReaderWriterLock\_t handle)

ReaderWriterLock AcquireWriter.

• void ReaderWriterLock\_ReleaseWriter (ReaderWriterLock\_t handle)

ReaderWriterLock\_ReleaseWriter.

bool ReaderWriterLock\_TimedAcquireWriter (ReaderWriterLock\_t handle, uint32\_t u32TimeoutMs\_)

ReaderWriterLock\_TimedAcquireWriter.

bool ReaderWriterLock\_TimedAcquireReader (ReaderWriterLock\_t handle, uint32\_t u32TimeoutMs\_)

ReaderWriterLock\_TimedAcquireReader.

# 20.1.1 Detailed Description

Implementation of C-language wrappers for the Mark3 kernel.

Definition in file mark3c.cpp.

# 20.1.2 Function Documentation

```
20.1.2.1 Alloc_ConditionVariable()
```

Definition at line 134 of file mark3c.cpp.

# 20.1.2.2 Alloc\_EventFlag()

Definition at line 64 of file mark3c.cpp.

# 20.1.2.3 Alloc\_Mailbox()

Alloc\_Mailbox.

See also

Mailbox\* AutoAlloc::NewMailbox()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 123 of file mark3c.cpp.

# 20.1.2.4 Alloc\_Memory()

Definition at line 29 of file mark3c.cpp.

#### 20.1.2.5 Alloc\_Message()

Alloc\_Message.

See also

AutoAlloc::NewMessage()

#### Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 76 of file mark3c.cpp.

#### 20.1.2.6 Alloc\_MessagePool()

Definition at line 100 of file mark3c.cpp.

# 20.1.2.7 Alloc\_MessageQueue()

Alloc\_MessageQueue.

See also

MesageQueue\* AutoAlloc::NewMessageQueue()

#### Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 88 of file mark3c.cpp.

```
20.1.2.8 Alloc_Mutex()
```

Alloc\_Mutex.

See also

Mutex\* AutoAlloc::NewMutex()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 52 of file mark3c.cpp.

# 20.1.2.9 Alloc\_Notify()

```
Notify_t Alloc_Notify ( void )
```

Alloc\_Notify.

See also

Notify\* AutoAlloc::NewNotify()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 112 of file mark3c.cpp.

# 20.1.2.10 Alloc\_ReaderWriterLock()

Definition at line 145 of file mark3c.cpp.

```
20.1.2.11 Alloc_Semaphore()
Semaphore_t Alloc_Semaphore (
              void )
Alloc_Semaphore.
See also
     Semaphore* AutoAlloc::NewSemaphore()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 40 of file mark3c.cpp.
20.1.2.12 Alloc_Thread()
Thread_t Alloc_Thread (
             void )
Alloc_Thread.
See also
     Thread* AutoAlloc::NewThread()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 156 of file mark3c.cpp.
20.1.2.13 Alloc_Timer()
Timer_t Alloc_Timer (
             void )
Alloc_Timer.
See also
     Timer* AutoAlloc::NewTimer()
```

# Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 167 of file mark3c.cpp.

# 20.1.2.14 Atomic\_Add16()

Atomic\_Add16.

# See also

```
uint16_t Atomic::Add(uint16_t *pu16Source_, uint16_t u16Val_)
```

#### **Parameters**

pu16⇔	Pointer to a variable
Source_	
u16Val_	Value to add to the variable

#### Returns

Previously-held value in pu16Source\_

Definition at line 638 of file mark3c.cpp.

# 20.1.2.15 Atomic\_Add32()

Atomic\_Add32.

# See also

```
uint32_t Atomic::Add(uint32_t *pu32Source_, uint32_t u32Val_)
```

# **Parameters**

<i>pu32</i> ⇔	Pointer to a variable
Source_	
u32Val_	Value to add to the variable

# Returns

Previously-held value in pu32Source\_

Definition at line 644 of file mark3c.cpp.

#### 20.1.2.16 Atomic\_Add8()

Atomic\_Add8.

See also

```
uint8_t Atomic::Add(uint8_t *pu8Source_, uint8_t u8Val_)
```

#### **Parameters**

pu8←	Pointer to a variable
Source_	
u8Val_	Value to add to the variable

#### Returns

Previously-held value in pu8Source\_

Definition at line 632 of file mark3c.cpp.

# 20.1.2.17 Atomic\_Set16()

Atomic\_Set16.

See also

```
uint16_t Atomic::Set(uint16_t *pu16Source_, uint16_t u16Val_)
```

# **Parameters**

pu16← Source_	Pointer to a variable to set the value of
u16Val_	New value to set in the variable

# Returns

Previously-set value

Definition at line 620 of file mark3c.cpp.

# 20.1.2.18 Atomic\_Set32()

Atomic\_Set32.

See also

```
uint32_t Atomic::Set(uint32_t *pu32Source_, uint32_t u32Val_)
```

#### **Parameters**

<i>pu32</i> ⇔	Pointer to a variable to set the value of
Source_	
u32Val_	New value to set in the variable

#### Returns

Previously-set value

Definition at line 626 of file mark3c.cpp.

# 20.1.2.19 Atomic\_Set8()

Atomic\_Set8.

See also

```
uint8_t Atomic::Set(uint8_t *pu8Source_, uint8_t u8Val_)
```

# **Parameters**

pu8⊷	Pointer to a variable to set the value of
Source_	
u8Val_	New value to set in the variable

# Returns

Previously-set value

Definition at line 614 of file mark3c.cpp.

#### 20.1.2.20 Atomic\_Sub16()

Atomic\_Sub16.

# See also

```
uint16_t Atomic::Sub(uint16_t *pu16Source_, uint16_t u16Val_)
```

#### **Parameters**

pu16↔ Source_	Pointer to a variable
u16Val_	Value to subtract from the variable

#### Returns

Previously-held value in pu16Source\_

Definition at line 656 of file mark3c.cpp.

#### 20.1.2.21 Atomic\_Sub32()

Atomic\_Sub32.

#### See also

```
uint32_t Atomic::Sub(uint32_t *pu32Source_, uint32_t u32Val_)
```

# **Parameters**

pu32⇔ Source_	Pointer to a variable
u32Val_	Value to subtract from the variable

# Returns

Previously-held value in pu32Source\_

Definition at line 662 of file mark3c.cpp.

#### 20.1.2.22 Atomic\_Sub8()

Atomic\_Sub8.

See also

```
uint8_t Atomic::Sub(uint8_t *pu8Source_, uint8_t u8Val_)
```

#### **Parameters**

pu8← Source_	Pointer to a variable
u8Val_	Value to subtract from the variable

#### Returns

Previously-held value in pu8Source\_

Definition at line 650 of file mark3c.cpp.

# 20.1.2.23 Atomic\_TestAndSet()

Atomic\_TestAndSet.

See also

bool Atomic::TestAndSet(bool \*pbLock)

#### **Parameters**

pbLock	Pointer to a value to test against.	This will always be set to "tru	e" at the end o	of a call to TestAndSet.
--------	-------------------------------------	---------------------------------	-----------------	--------------------------

# Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 668 of file mark3c.cpp.

### 20.1.2.24 ConditionVariable\_Broadcast()

 $Condition Variable\_Broadcast.$ 

See also

void ConditionVariable::Broadcast()

#### **Parameters**

handle	Handle of the condition variable object
--------	---

Definition at line 874 of file mark3c.cpp.

### 20.1.2.25 ConditionVariable\_Init()

ConditionVariable\_Init.

See also

void ConditionVariable::Init()

#### **Parameters**

handle	Handle of the condition variable object
--------	---

Definition at line 855 of file mark3c.cpp.

### 20.1.2.26 ConditionVariable\_Signal()

ConditionVariable\_Signal.

See also

void ConditionVariable::Signal()

#### **Parameters**

handle	Handle of the condition variable object
--------	---

Definition at line 868 of file mark3c.cpp.

### 20.1.2.27 ConditionVariable\_TimedWait()

ConditionVariable\_TimedWait.

#### See also

bool ConditionVariable::Wait(Mutex\* pclMutex\_, uint32\_t u32WaitTimeMS\_)

#### **Parameters**

handle	Handle of the condition variable object
hMutex_	Handle of the mutex to lock on acquisition of the condition variable
u32WaitTimeM↔ S_	Maximum time to wait for object

#### Returns

true on success, false on timeout

Definition at line 880 of file mark3c.cpp.

### 20.1.2.28 ConditionVariable\_Wait()

ConditionVariable\_Wait.

#### See also

void ConditionVariable::Wait(Mutex\* pclMutex\_)

handle	Handle of the condition variable object
h <i>⊷</i> Mutex <i>←</i>	Handle of the mutex to lock on acquisition of the condition variable
_	

Definition at line 861 of file mark3c.cpp.

# 20.1.2.29 Free\_ConditionVariable()

Definition at line 139 of file mark3c.cpp.

#### 20.1.2.30 Free\_EventFlag()

Definition at line 70 of file mark3c.cpp.

## 20.1.2.31 Free\_Mailbox()

Definition at line 128 of file mark3c.cpp.

# 20.1.2.32 Free\_Memory()

Free\_Memory.

### **Parameters**

pv⊷	Pointer to previously allocated block of memory
Object_	

Definition at line 34 of file mark3c.cpp.

```
20.1.2.33 Free_Message()
```

Definition at line 82 of file mark3c.cpp.

### 20.1.2.34 Free\_MessagePool()

Definition at line 106 of file mark3c.cpp.

## 20.1.2.35 Free\_MessageQueue()

Definition at line 94 of file mark3c.cpp.

## 20.1.2.36 Free\_Mutex()

Definition at line 58 of file mark3c.cpp.

## 20.1.2.37 Free\_Notify()

```
void Free_Notify (
          Notify_t handle )
```

Definition at line 118 of file mark3c.cpp.

```
20.1.2.38 Free_ReaderWriterLock()
void Free_ReaderWriterLock (
             ReaderWriterLock_t handle )
Definition at line 150 of file mark3c.cpp.
20.1.2.39 Free_Semaphore()
void Free_Semaphore (
              Semaphore_t handle )
Definition at line 46 of file mark3c.cpp.
20.1.2.40 Free_Thread()
void Free_Thread (
            Thread_t handle )
Definition at line 161 of file mark3c.cpp.
20.1.2.41 Free_Timer()
void Free_Timer (
             Timer_t handle )
Definition at line 172 of file mark3c.cpp.
20.1.2.42 Kernel_DebugPrint()
```

#### **Parameters**

SZ⊷	String to print to debug interface
String_	

Definition at line 282 of file mark3c.cpp.

## 20.1.2.43 Kernel\_GetStackGuardThreshold()

Definition at line 270 of file mark3c.cpp.

### 20.1.2.44 Kernel\_GetTicks()

 $Kernel\_GetTicks.$ 

See also

Kernel::GetTicks()

## Returns

Number of kernel ticks that have elapsed since boot

Definition at line 216 of file mark3c.cpp.

## 20.1.2.45 Kernel\_Init()

```
void Kernel_Init (
     void )
```

Kernel\_Init.

See also

void Kernel::Init()

Definition at line 180 of file mark3c.cpp.

```
20.1.2.46 Kernel_IsPanic()
bool Kernel_IsPanic (
              void )
Kernel_IsPanic.
See also
     bool Kernel::IsPanic()
Returns
     Whether or not the kernel is in a panic state
Definition at line 204 of file mark3c.cpp.
20.1.2.47 Kernel_IsStarted()
bool Kernel_IsStarted (
               void )
Kernel_IsStarted.
See also
     bool Kernel::IsStarted()
Returns
     Whether or not the kernel has started - true = running, false = not started
Definition at line 192 of file mark3c.cpp.
20.1.2.48 Kernel_Panic()
void Kernel_Panic (
             uint16_t u16Cause_ )
Kernel Panic.
See also
```

void Kernel::Panic(uint16\_t u16Cause\_)

### **Parameters**

u16⇔	Reason for the kernel panic
Cause_	

Definition at line 210 of file mark3c.cpp.

### 20.1.2.49 Kernel\_SetDebugPrintFunction()

 $Kernel\_SetDebugPrintFunction.$ 

See also

void Kernel::SetDebugPrintFunction()

#### **Parameters**

pfPrint←	Function to use to print debug information from the kernel
Function_	

Definition at line 276 of file mark3c.cpp.

### 20.1.2.50 Kernel\_SetPanic()

Kernel\_SetPanic.

See also

void Kernel::SetPanic(PanicFunc\_t pfPanic\_)

# **Parameters**

pf⇔	Panic function pointer
Panic⊷	

Definition at line 198 of file mark3c.cpp.

#### 20.1.2.51 Kernel\_SetStackGuardThreshold()

Definition at line 264 of file mark3c.cpp.

## 20.1.2.52 Kernel\_Start()

```
void Kernel_Start (
    void )
```

Kernel\_Start.

See also

void Kernel::Start()

Definition at line 186 of file mark3c.cpp.

## 20.1.2.53 Mailbox\_GetFreeSlots()

Mailbox\_GetFreeSlots.

See also

uint16\_t Mailbox::GetFreeSlots()

#### **Parameters**

handle	Handle of the mailbox object
--------	------------------------------

#### Returns

Number of free slots in the mailbox

Definition at line 832 of file mark3c.cpp.

### 20.1.2.54 Mailbox\_Init()

Mailbox\_Init.

#### See also

void Mailbox::Init(void \*pvBuffer\_, uint16\_t u16BufferSize\_, uint16\_t u16ElementSize\_)

#### **Parameters**

handle	Handle of the mailbox object
pvBuffer_	Pointer to the static buffer to use for the mailbox
u16BufferSize_	Size of the mailbox buffer, in bytes
u16Element← Size_	Size of each envelope, in bytes

Definition at line 769 of file mark3c.cpp.

## 20.1.2.55 Mailbox\_lsEmpty()

Mailbox\_IsEmpty.

See also

bool Mailbox::IsEmpty()

#### **Parameters**

handle	Handle of the mailbox object
--------	------------------------------

### Returns

true if the mailbox is empty, false otherwise

Definition at line 846 of file mark3c.cpp.

### 20.1.2.56 Mailbox\_lsFull()

Mailbox\_IsFull.

See also

bool Mailbox::IsFull()

# **Parameters**

#### Returns

true if the mailbox is full, false otherwise

Definition at line 839 of file mark3c.cpp.

### 20.1.2.57 Mailbox\_Receive()

Mailbox\_Receive.

See also

void Mailbox::Receive(void \*pvData\_)

### **Parameters**

handle	Handle of the mailbox object
pv⇔ Data	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

Definition at line 804 of file mark3c.cpp.

#### 20.1.2.58 Mailbox\_ReceiveTail()

Mailbox\_ReceiveTail.

#### See also

```
void Mailbox::ReceiveTail(void *pvData_)
```

#### **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
Data_	

Definition at line 811 of file mark3c.cpp.

#### 20.1.2.59 Mailbox\_Send()

Mailbox\_Send.

### See also

bool Mailbox::Send(void \*pvData\_)

## **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to the data object to send to the mailbox.
Data_	

### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 776 of file mark3c.cpp.

# 20.1.2.60 Mailbox\_SendTail()

Mailbox\_SendTail.

See also

bool Mailbox::SendTail(void \*pvData\_)

handle	Handle of the mailbox object
pv⊷	Pointer to the data object to send to the mailbox.
Data_	

#### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 783 of file mark3c.cpp.

### 20.1.2.61 Mailbox\_TimedReceive()

Mailbox\_TimedReceive.

#### See also

bool Mailbox::Receive(void \*pvData\_, uint32\_t u32TimeoutMS\_)

### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM⊷ S_	Maximum time to wait for delivery.

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 818 of file mark3c.cpp.

## 20.1.2.62 Mailbox\_TimedReceiveTail()

Mailbox\_TimedReceiveTail.

#### See also

bool Mailbox::ReceiveTail(void \*pvData\_, uint32\_t u32TimeoutMS\_)

### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM⊷ S_	Maximum time to wait for delivery.

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 825 of file mark3c.cpp.

## 20.1.2.63 Mailbox\_TimedSend()

Mailbox\_TimedSend.

### See also

bool Mailbox::Send(void \*pvData\_, uint32\_t u32TimeoutMS\_)

### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM⊷ S_	Maximum time to wait for a free transmit slot

#### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 790 of file mark3c.cpp.

### 20.1.2.64 Mailbox\_TimedSendTail()

Mailbox\_TimedSendTail.

See also

bool Mailbox::Send(void \*pvData\_, uint32\_t u32TimeoutMS\_)

#### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM← S_	Maximum time to wait for a free transmit slot

#### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 797 of file mark3c.cpp.

## 20.1.2.65 Message\_GetCode()

Message\_GetCode.

See also

uint16\_t Message::GetCode()

## **Parameters**

handle	Handle of the message object

#### Returns

user code set in the object

Definition at line 704 of file mark3c.cpp.

### 20.1.2.66 Message\_GetData()

 $Message\_GetData.$ 

See also

void\* Message::GetData()

#### **Parameters**

### Returns

Pointer to the data set in the message object

Definition at line 690 of file mark3c.cpp.

## 20.1.2.67 Message\_Init()

Message Init.

See also

void Message::Init()

### **Parameters**

handle Handle of the message ob	ject
---------------------------------	------

Definition at line 676 of file mark3c.cpp.

## 20.1.2.68 Message\_SetCode()

Message\_SetCode.

### See also

void Message::SetCode(uint16\_t u16Code\_)

### **Parameters**

handle	Handle of the message object
<i>u</i> 16⇔	Data code to set in the object
Code_	

Definition at line 697 of file mark3c.cpp.

## 20.1.2.69 Message\_SetData()

Message\_SetData.

### See also

void Message::SetData(void \*pvData\_)

### **Parameters**

handle	Handle of the message object
pv⊷	Pointer to the data object to send in the message
Data_	

Definition at line 683 of file mark3c.cpp.

### 20.1.2.70 MessagePool\_Init()

MessagePool\_Init.

### See also

void MessagePool::Init()

### **Parameters**

handle	Handle of the message pool object
--------	-----------------------------------

Definition at line 725 of file mark3c.cpp.

## 20.1.2.71 MessagePool\_Pop()

MessagePool\_Pop.

See also

Message\* MessagePool::Pop()

#### **Parameters**

handle	Handle of the message pool object
--------	-----------------------------------

## Returns

Handle to a Message object, or nullptr on allocation error

Definition at line 739 of file mark3c.cpp.

### 20.1.2.72 MessagePool\_Push()

MessagePool\_Push.

#### See also

void MessagePool::Push(Message\* pclMessage\_)

### **Parameters**

handle	Handle of the message pool object
msg	Message object to return back to the pool

Definition at line 732 of file mark3c.cpp.

```
20.1.2.73 MessageQueue_GetCount()
```

MessageQueue\_GetCount.

See also

uint16\_t MessageQueue::GetCount()

#### Returns

Count of pending messages in the queue.

Definition at line 760 of file mark3c.cpp.

### 20.1.2.74 MessageQueue\_Init()

MessageQueue\_Init.

See also

void MessageQueue::Init()

### **Parameters**

```
handle Handle to the message queue to initialize
```

Definition at line 711 of file mark3c.cpp.

## 20.1.2.75 MessageQueue\_Receive()

MessageQueue\_Receive.

#### See also

Message\_t MessageQueue::Receive()

#### **Parameters**

handle	Handle of the message queue object
--------	------------------------------------

#### Returns

Pointer to a message object at the head of the queue

Definition at line 718 of file mark3c.cpp.

### 20.1.2.76 MessageQueue\_Send()

MessageQueue\_Send.

### See also

void MessageQueue::Send(Message \*pclMessage\_)

### **Parameters**

handle	Handle of the message queue object
h⊷	Handle to the message to send to the given queue
Message←	
_	

Definition at line 753 of file mark3c.cpp.

# 20.1.2.77 MessageQueue\_TimedReceive()

MessageQueue\_TimedReceive.

### See also

Message\_t MessageQueue::TimedReceive(uint32\_t u32TimeWaitMS\_)

handle	Handle of the message queue object
u32TimeWaitM↔ S_	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 nullptr on timeout.

Definition at line 746 of file mark3c.cpp.

## 20.1.2.78 Mutex\_Claim()

Mutex\_Claim.

See also

void Mutex::Claim()

### **Parameters**

handle	Handle of the mutex

Definition at line 515 of file mark3c.cpp.

# 20.1.2.79 Mutex\_Init()

Mutex\_Init.

See also

void Mutex::Init()

#### **Parameters**

handle Handle of the mutex
----------------------------

Definition at line 508 of file mark3c.cpp.

### 20.1.2.80 Mutex\_Release()

 $Mutex\_Release.$ 

See also

void Mutex::Release()

#### **Parameters**

handle	Handle of the mutex
--------	---------------------

Definition at line 522 of file mark3c.cpp.

## 20.1.2.81 Mutex\_TimedClaim()

Mutex\_TimedClaim.

See also

bool Mutex::Claim(uint32\_t u32WaitTimeMS\_)

#### **Parameters**

handle	Handle of the mutex
u32WaitTimeM⊷	Time to wait before aborting
S_	

### Returns

true if mutex was claimed, false on timeout

Definition at line 529 of file mark3c.cpp.

```
20.1.2.82 Notify_Init()
```

Notify\_Init.

See also

void Notify::Init()

#### **Parameters**

handle	Handle of the notification object
--------	-----------------------------------

Definition at line 584 of file mark3c.cpp.

## 20.1.2.83 Notify\_Signal()

Notify\_Signal.

See also

void Notify::Signal()

## **Parameters**

handle Handle of the notification obj	ect

Definition at line 591 of file mark3c.cpp.

### 20.1.2.84 Notify\_TimedWait()

Notify\_TimedWait.

See also

bool Notify::Wait(uint32\_t u32WaitTimeMS\_, bool \*pbFlag\_)

### **Parameters**

handle	Handle of the notification object
u32WaitTimeM⊷ S_	Maximum time to wait for notification in ms
pbFlag_	Flag to set to true on notification

#### Returns

true on unblock, false on timeout

Definition at line 605 of file mark3c.cpp.

## 20.1.2.85 Notify\_Wait()

Notify\_Wait.

## See also

void Notify::Wait(bool \*pbFlag\_)

### **Parameters**

handle	Handle of the notification object
pb⊷	Flag to set to true on notification
Flag_	

Definition at line 598 of file mark3c.cpp.

# 20.1.2.86 ReaderWriterLock\_AcquireReader()

ReaderWriterLock\_AcquireReader.

## See also

void ReaderWriterLock::AcquireReader()

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 897 of file mark3c.cpp.

### 20.1.2.87 ReaderWriterLock\_AcquireWriter()

ReaderWriterLock\_AcquireWriter.

See also

void ReaderWriterLock::AcquireWriter()

#### **Parameters**

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 911 of file mark3c.cpp.

#### 20.1.2.88 ReaderWriterLock\_Init()

ReaderWriterLock\_Init.

See also

void ReaderWriterLock::Init()

## **Parameters**

Handle of the reader-writer object
------------------------------------

Definition at line 890 of file mark3c.cpp.

#### 20.1.2.89 ReaderWriterLock\_ReleaseReader()

 $Reader Writer Lock\_Release Reader.$ 

See also

void ReaderWriterLock::ReleaseReader()

#### **Parameters**

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 904 of file mark3c.cpp.

### 20.1.2.90 ReaderWriterLock\_ReleaseWriter()

ReaderWriterLock\_ReleaseWriter.

See also

void ReaderWriterLock::ReleaseWriter()

#### **Parameters**

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 918 of file mark3c.cpp.

### 20.1.2.91 ReaderWriterLock\_TimedAcquireReader()

 $Reader Writer Lock\_Timed Acquire Reader.$ 

See also

bool ReaderWriterLock::AcquireReader(uint32\_t u32TimeoutMs\_)

handle	Handle of the reader-writer object
u32Timeout⊷	Maximum time to wait for the reader lock before bailing
Ms_	

#### Returns

true on success, false on timeout

Definition at line 932 of file mark3c.cpp.

### 20.1.2.92 ReaderWriterLock\_TimedAcquireWriter()

ReaderWriterLock\_TimedAcquireWriter.

#### See also

bool ReaderWriterLock::AcquireWriter(uint32\_t u32TimeoutMs\_)

#### **Parameters**

handle	Handle of the reader-writer object
u32Timeout←	Maximum time to wait for the writer lock before bailing
Ms_	

#### Returns

true on success, false on timeout

Definition at line 925 of file mark3c.cpp.

# 20.1.2.93 Scheduler\_Enable()

```
void Scheduler_Enable (
          bool bEnable_ )
```

Scheduler\_Enable.

#### See also

void Scheduler::SetScheduler(bool bEnable\_)

#### **Parameters**

b⇔	true to enable, false to disable the scheduler
Enable⊷	

Definition at line 289 of file mark3c.cpp.

```
20.1.2.94 Scheduler_GetCurrentThread()
```

Scheduler\_GetCurrentThread.

See also

Thread\* Scheduler::GetCurrentThread()

Returns

Handle of the currently-running thread

Definition at line 301 of file mark3c.cpp.

```
20.1.2.95 Scheduler_IsEnabled()
```

Scheduler\_IsEnabled.

See also

bool Scheduler::IsEnabled()

Returns

true - scheduler enabled, false - disabled

Definition at line 295 of file mark3c.cpp.

# 20.1.2.96 Semaphore\_Init()

Semaphore\_Init.

See also

void Semaphore::Init(uint16\_t u16InitVal\_, uint16\_t u16MaxVal\_)

handle	Handle of the semaphore
u16InitVal⇔	Initial value of the semaphore
_	
u16Max⇔	Maximum value that can be held for a semaphore
Val_	

Definition at line 478 of file mark3c.cpp.

## 20.1.2.97 Semaphore\_Pend()

Semaphore\_Pend.

See also

void Semaphore::Pend()

### **Parameters**

handl	e Hand	dle of the semaphore
-------	--------	----------------------

Definition at line 492 of file mark3c.cpp.

## 20.1.2.98 Semaphore\_Post()

Semaphore\_Post.

See also

void Semaphore::Post()

## **Parameters**

handle Handle of the semaphore
--------------------------------

Definition at line 485 of file mark3c.cpp.

### 20.1.2.99 Semaphore\_TimedPend()

Semaphore\_TimedPend.

See also

bool Semaphore::Pend(uint32\_t u32WaitTimeMS\_)

#### **Parameters**

handle	Handle of the semaphore
u32WaitTimeM⊷	Time in ms to wait
S_	

### Returns

true if semaphore was acquired, false on timeout

Definition at line 499 of file mark3c.cpp.

### 20.1.2.100 Thread\_CoopYield()

 $Thread\_CoopYield.$ 

See also

void Thread::CoopYield()

Definition at line 415 of file mark3c.cpp.

```
20.1.2.101 Thread_Exit()
```

Thread\_Exit.

See also

void Thread::Exit()

handle Handle of the thread
-----------------------------

Definition at line 381 of file mark3c.cpp.

## 20.1.2.102 Thread\_GetCurPriority()

Thread\_GetCurPriority.

See also

PORT\_PRIO\_TYPE Thread::GetCurPriority()

#### **Parameters**

handle H	landle of the thread
----------	----------------------

#### Returns

Current priority of the thread considering priority inheritence

Definition at line 354 of file mark3c.cpp.

### 20.1.2.103 Thread\_GetID()

Thread\_GetID.

See also

uint8\_t Thread::GetID()

### **Parameters**

handle Handle of the thread

Returns

Return ID assigned to the thread

Definition at line 426 of file mark3c.cpp.

```
20.1.2.104 Thread_GetName()
```

Definition at line 341 of file mark3c.cpp.

### 20.1.2.105 Thread\_GetPriority()

Thread\_GetPriority.

See also

PORT\_PRIO\_TYPE Thread::GetPriority()

#### **Parameters**

```
handle Handle of the thread
```

Returns

Current priority of the thread not considering priority inheritence

Definition at line 348 of file mark3c.cpp.

### 20.1.2.106 Thread\_GetQuantum()

Definition at line 367 of file mark3c.cpp.

### 20.1.2.107 Thread\_GetStackSlack()

Definition at line 432 of file mark3c.cpp.

## 20.1.2.108 Thread\_GetState()

Thread GetState.

See also

ThreadState Thread::GetState()

#### **Parameters**

```
handle Handle of the thread
```

### Returns

The thread's current execution state

Definition at line 438 of file mark3c.cpp.

# 20.1.2.109 Thread\_Init()

```
void Thread_Init (
          Thread_t handle,
          K_WORD * pwStack_,
          uint16_t u16StackSize_,
          PORT_PRIO_TYPE uXPriority_,
          thread_entry_func_t pfEntryPoint_,
          void * pvArg_ )
```

Thread\_Init.

### See also

```
void Thread::Init(K_WORD *pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, Thread← Entry_t pfEntryPoint_, void *pvArg_)
```

### **Parameters**

handle	Handle of the thread to initialize	
pwStack_	Pointer to the stack to use for the thread	
u16Stack⊸	Size of the stack (in bytes)	
Size_		
uXPriority_	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 310 of file mark3c.cpp.

# 20.1.2.110 Thread\_SetID()

Thread\_SetID.

#### See also

void Thread::SetID(uint8\_t u8ID\_)

## **Parameters**

handle	Handle of the thread
u8l⊷	ID To assign to the thread
<i>D_</i>	

Definition at line 420 of file mark3c.cpp.

# 20.1.2.111 Thread\_SetName()

Definition at line 335 of file mark3c.cpp.

## 20.1.2.112 Thread\_SetPriority()

Thread\_SetPriority.

See also

void Thread::SetPriority(PORT\_PRIO\_TYPE uXPriority\_)

#### **Parameters**

handle	Handle of the thread
uX⇔	New priority level
Priority_	

Definition at line 374 of file mark3c.cpp.

### 20.1.2.113 Thread\_SetQuantum()

Definition at line 361 of file mark3c.cpp.

## 20.1.2.114 Thread\_Sleep()

Thread\_Sleep.

See also

void Thread::Sleep(uint32\_t u32TimeMs\_)

### **Parameters**

u32Time⊷	Time in ms to block the thread for
Ms	

Definition at line 388 of file mark3c.cpp.

```
20.1.2.115 Thread_Start()
```

Thread\_Start.

See also

void Thread::Start()

#### **Parameters**

Definition at line 322 of file mark3c.cpp.

## 20.1.2.116 Thread\_Stop()

Thread\_Stop.

See also

void Thread::Stop()

### **Parameters**

handle	Handle of the thread to stop
--------	------------------------------

Definition at line 329 of file mark3c.cpp.

## 20.1.2.117 Thread\_Yield()

```
void Thread_Yield (
     void )
```

Thread\_Yield.

```
See also
```

```
void Thread::Yield()
```

Definition at line 410 of file mark3c.cpp.

```
20.1.2.118 Timer_Init()
```

Timer\_Init.

See also

void Timer::Init()

#### **Parameters**

handle	Handle of the timer
--------	---------------------

Definition at line 448 of file mark3c.cpp.

## 

Timer\_Restart.

See also

void Timer::Start()

## **Parameters**

handle	Handle of the timer to restart.
--------	---------------------------------

Definition at line 469 of file mark3c.cpp.

# 20.1.2.120 Timer\_Start()

```
bool bRepeat_,
uint32_t u32IntervalMs_,
timer_callback_t pfCallback_,
void * pvData_ )
```

Timer\_Start.

## See also

 $void\ Timer::Start(bool\ bRepeat\_,\ uint32\_t\ u32IntervalMs\_,\ uint32\_t\ u32ToleranceMs\_,\ TimerCallbackC\_t\ pf \leftarrow Callback\_,\ void\ *pvData\_)$ 

#### **Parameters**

handle	Handle of the timer
bRepeat_	Restart the timer continuously on expiry
u32Interval← Ms_	Time in ms to expiry
pfCallback_	Callback to run on timer expiry
pvData_	Data to pass to the callback on expiry

Definition at line 455 of file mark3c.cpp.

## 20.1.2.121 Timer\_Stop()

Timer\_Stop.

See also

void Timer::Stop()

#### **Parameters**

handle Handle of the timer
----------------------------

Definition at line 462 of file mark3c.cpp.



```
00007
                            _1
                       1_
                                   1___
                                         ___
                                                1__
               ___
00008
00009 -- [Mark3 Realtime Platform] -----
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00021 #include "mark3c.h"
00022 #include "mark3.h"
00023
00024 using namespace Mark3;
00025
00026 /
00027 // Kernel Memory managment APIs
00028 //--
00029 void* Alloc_Memory(uint16_t u16Size_)
00030 {
00031
          return AutoAlloc::NewRawData(u16Size);
00032 }
00033 //--
00034 void Free_Memory(void* pv0bject_)
00035 {
00036
         AutoAlloc::DestroyRawData(pvObject_);
00037 }
00038
00039 //--
00040 Semaphore_t Alloc_Semaphore(void)
00041 {
00042
          return (Semaphore_t)AutoAlloc::NewObject<Semaphore, AutoAllocType::Semaphore>();
00043 }
00044
00045 //-
00046 void Free_Semaphore(Semaphore_t handle)
00047 {
00048
          AutoAlloc::DestroyObject<Semaphore, AutoAllocType::Semaphore>((Semaphore*)handle);
00049 }
00050
00051 //--
00052 Mutex_t Alloc_Mutex(void)
00053 {
00054
          return (Mutex_t)AutoAlloc::NewObject<Mutex, AutoAllocType::Mutex>();
00055 }
00056
00057 //-
00058 void Free_Mutex(Mutex_t handle)
00059 {
00060
         AutoAlloc::DestroyObject<Mutex, AutoAllocType::Mutex>((Mutex*)handle);
00061 }
00062
00063 //
00064 EventFlag_t Alloc_EventFlag(void)
00065 {
00066
          return (EventFlag_t)AutoAlloc::NewObject<EventFlag, AutoAllocType::EventFlag>();
00067 }
00068
00069 //
00070 void Free_EventFlag(EventFlag_t handle)
00071 {
00072
          \verb|AutoAlloc::DestroyObject<EventFlag, AutoAllocType::EventFlag>((EventFlag*) handle);\\
00073 }
00074
00075 //-
00076 Message_t Alloc_Message(void)
00077 {
00078
          return (Message_t)AutoAlloc::NewObject<Message, AutoAllocType::Message>();
00079 }
08000
00081 //-
00082 void Free_Message(Message_t handle)
00083 {
00084
          AutoAlloc::DestroyObject<Message, AutoAllocType::Message>((Message*)handle);
00085 }
00086
00087 //--
00088 MessageQueue_t Alloc_MessageQueue(void)
00089 {
00090
          return (MessageQueue_t)AutoAlloc::NewObject<MessageQueue, AutoAllocType::MessageQueue>();
00091 }
00092
00093 //---
00094 void Free MessageQueue (MessageQueue t handle)
00095 {
          AutoAlloc::DestroyObject<MessageQueue, AutoAllocType::MessageQueue>((
00096
     MessageQueue*)handle);
00097 }
00098
00099 //
```

```
00100 MessagePool_t Alloc_MessagePool(void)
00101 {
00102
          return (MessagePool_t)AutoAlloc::NewObject<MessagePool, AutoAllocType::MessagePool>();
00103 }
00104
00105 //-
00106 void Free_MessagePool(MessagePool_t handle)
00107 {
00108
         AutoAlloc::DestroyObject<MessagePool, AutoAllocType::MessagePool>((
     MessagePool*)handle);
00109 }
00110
00111 //-
00112 Notify_t Alloc_Notify(void)
00113 {
00114
          return (Notify_t)AutoAlloc::NewObject<Notify, AutoAllocType::Notify>();
00115 }
00116
00117 //-
00118 void Free_Notify(Notify_t handle)
00119 {
00120
          AutoAlloc::DestroyObject<Notify, AutoAllocType::Notify>((Notify*)handle);
00121 }
00122 //-
00123 Mailbox_t Alloc_Mailbox(void)
00124 {
00125
          return (Mailbox_t)AutoAlloc::NewObject<Mailbox, AutoAllocType::MailBox>();
00126 }
00127 //---
00128 void Free Mailbox (Mailbox t handle)
00129 {
00130
          AutoAlloc::DestroyObject<Mailbox, AutoAllocType::MailBox>((Mailbox*)handle);
00131 }
00132
00133 //---
00134 ConditionVariable_t Alloc_ConditionVariable(void)
00135 {
00136
          return (ConditionVariable_t) AutoAlloc::NewObject<ConditionVariable,</pre>
       AutoAllocType::ConditionVariable>();
00137 }
00138 //----
00139 void Free ConditionVariable (ConditionVariable t handle)
00140 {
         AutoAlloc::DestroyObject<ConditionVariable, AutoAllocType::ConditionVariable>((
00141
      ConditionVariable*) handle);
00142 }
00143
00144 //---
00145 ReaderWriterLock t Alloc ReaderWriterLock(void)
00146 {
00147
          return (ReaderWriterLock_t) AutoAlloc::NewObject<ReaderWriterLock,</pre>
       AutoAllocType::ReaderWriterLock>();
00148 }
00149 //---
00150 void Free_ReaderWriterLock(ReaderWriterLock_t handle)
00151 {
00152
          AutoAlloc::DestroyObject<ReaderWriterLock, AutoAllocType::ReaderWriterLock>((
      ReaderWriterLock*)handle);
00153 }
00154
00155 //----
00156 Thread_t Alloc_Thread(void)
00157 {
00158
          return (Thread_t)AutoAlloc::NewObject<Thread, AutoAllocType::Thread>();
00159
00160 //---
00161 void Free_Thread(Thread_t handle)
00162 {
00163
          AutoAlloc::DestroyObject<Thread, AutoAllocType::Thread>((Thread*)handle);
00164 }
00165
00166 //---
00167 Timer_t Alloc_Timer(void)
00168 {
00169
          return (Thread t)AutoAlloc::NewObject<Timer, AutoAllocType::Timer>();
00170 }
00171 //--
00172 void Free_Timer(Timer_t handle)
00173 {
00174
         AutoAlloc::DestrovObject<Timer, AutoAllocType::Timer>((Timer*)handle):
00175 }
00176
00177 /
00178 // Kernel APIs
00179 //--
00180 void Kernel_Init(void)
00181 {
```

```
00182
         Kernel::Init();
00183 }
00184
00185 //----
00186 void Kernel_Start (void)
00187 {
00188
         Kernel::Start();
00189 }
00190
00191 //----
00192 bool Kernel_IsStarted(void)
00193 {
00194
         return Kernel::IsStarted();
00195 }
00196
00197 //---
00198 void Kernel_SetPanic(PanicFunc pfPanic_)
00199 {
00200
          Kernel::SetPanic(pfPanic_);
00201 }
00202
00203 //--
00204 bool Kernel_IsPanic(void)
00205 {
00206
         return Kernel::IsPanic();
00207 }
00208
00209 //--
00210 void Kernel_Panic(uint16_t u16Cause_)
00211 {
00212
          Kernel::Panic(u16Cause);
00213 }
00214
00215 //--
00216 uint32_t Kernel_GetTicks(void)
00217 {
00218
         return Kernel::GetTicks();
00219 }
00220
00221 #if KERNEL_THREAD_CREATE_CALLOUT
00222 //--
00223 void Kernel_SetThreadCreateCallout(thread_create_callout_t pfCreate_)
00224 {
         Kernel::SetThreadCreateCallout((
00225
      ThreadCreateCallout)pfCreate_);
00226 }
00227 #endif // #if KERNEL_THREAD_CREATE_CALLOUT
00228 //--
00229 #if KERNEL THREAD EXIT CALLOUT
00230 void Kernel_SetThreadExitCallout(thread_exit_callout_t pfExit_)
00231 {
00232
          Kernel::SetThreadExitCallout((ThreadExitCallout)pfExit_);
00233 }
00234 #endif //#if KERNEL_THREAD_EXIT_CALLOUT
00235 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00236 //-
00237 void Kernel_SetThreadContextSwitchCallout(thread_context_callout_t pfContext_)
00238 {
00239
         Kernel::SetThreadContextSwitchCallout((
     ThreadContextCallout)pfContext_);
00240 }
00241 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00242 #if KERNEL_THREAD_CREATE_CALLOUT
00243 //-
00244 thread_create_callout_t Kernel_GetThreadCreateCallout(void)
00245 {
00246
          return (thread_create_callout_t)Kernel::GetThreadCreateCallout();
00247 }
00248 #endif // #if KERNEL_THREAD_CREATE_CALLOUT
00249 #if KERNEL_THREAD_EXIT_CALLOUT
00250 //-
00251 thread_exit_callout_t Kernel_GetThreadExitCallout(void)
00252 {
00253
          return (thread_exit_callout_t) Kernel::GetThreadExitCallout();
00254 }
00255 #endif // #if KERNEL_THREAD_EXIT_CALLOUT
00256 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00257 //-
\tt 00258\ thread\_context\_callout\_t\ Kernel\_GetThreadContextSwitchCallout(void)
00259 {
00260
          return (thread_context_callout_t)Kernel::GetThreadContextSwitchCallout
      ();
00261 }
00262 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00263 //--
00264 void Kernel_SetStackGuardThreshold(uint16_t u16Threshold_)
00265 {
```

```
00266
         Kernel::SetStackGuardThreshold(u16Threshold_);
00267 }
00268
00269 //----
00270 uint16_t Kernel_GetStackGuardThreshold(void)
00271 {
          return Kernel::GetStackGuardThreshold();
00273 }
00274
00275 //----
00276 void Kernel SetDebugPrintFunction(
     kernel_debug_print_t pfPrintFunction_)
00277 {
00278
         Kernel::SetDebugPrintFunction(pfPrintFunction_);
00279 }
00280
00281 //--
00282 void Kernel_DebugPrint(const char* szString_)
00283 {
00284
         Kernel::DebugPrint(szString_);
00285 }
00286
00287 //--
00288 // Scheduler APIs
00289 void Scheduler_Enable(bool bEnable_)
00290 {
00291
          Scheduler::SetScheduler(bEnable_);
00292 }
00293
00294 //----
00295 bool Scheduler IsEnabled(void)
00296 {
00297
         return Scheduler::IsEnabled();
00298 }
00299
00300 //---
00301 Thread t Scheduler GetCurrentThread(void)
00302 {
00303
         return (Thread_t)Scheduler::GetCurrentThread();
00304 }
00305
00306 //----
00307 // Thread APIs
00308 //---
00310 void Thread_Init(Thread_t
                                      handle,
                                   pwStack_
00311
                     K_WORD*
                                      u16StackSize_,
00312
                       uint16 t
                       PORT_PRIO_TYPE uXPriority_,
00313
00314
                      ThreadEntryFunc pfEntryPoint_,
00315
                      void*
                                      pvArg_)
00316 {
00317
         Thread* pclThread = new ((void*)handle) Thread();
00318
         pclThread->Init(pwStack_, u16StackSize_, uXPriority_, pfEntryPoint_, pvArg_);
00319 }
00320
00321 //-
00322 void Thread_Start(Thread_t handle)
00323 {
00324
         Thread* pclThread = (Thread*)handle;
00325
         pclThread->Start();
00326 }
00327
00328 //--
00329 void Thread_Stop(Thread_t handle)
00330 {
         Thread* pclThread = (Thread*)handle;
00331
00332
         pclThread->Stop();
00333 }
00334 //--
00335 void Thread_SetName(Thread_t handle, const char* szName_)
00336 {
00337
         Thread* pclThread = (Thread*)handle;
00338
         pclThread->SetName(szName_);
00339 }
00340 //--
00341 const char* Thread_GetName(Thread_t handle)
00342 {
00343
         Thread* pclThread = (Thread*)handle;
         return pclThread->GetName();
00344
00345 }
00346
00347 //---
00348 PORT_PRIO_TYPE Thread_GetPriority(Thread_t handle)
00349 {
          Thread* pclThread = (Thread*)handle;
00350
         return pclThread->GetPriority();
00351
```

```
00352 }
00353 //--
00354 PORT_PRIO_TYPE Thread_GetCurPriority(Thread_t handle)
00355 {
00356
          Thread* pclThread = (Thread*)handle;
         return pclThread->GetCurPriority();
00357
00358 }
00359
00360 //--
00361 void Thread_SetQuantum(Thread_t handle, uint16_t u16Quantum_)
00362 {
00363
          Thread* pclThread = (Thread*)handle;
00364
         pclThread->SetQuantum(u16Quantum);
00365 }
00366 //--
00367 uint16_t Thread_GetQuantum(Thread_t handle)
00368 {
00369
          Thread* pclThread = (Thread*)handle;
         return pclThread->GetQuantum();
00370
00371 }
00372
00373 //---
00374 void Thread_SetPriority(Thread_t handle,
     PORT_PRIO_TYPE uXPriority_)
00375 {
00376
          Thread* pclThread = (Thread*)handle;
00377
         pclThread->SetPriority(uXPriority_);
00378 }
00379
00380 //-----
00381 void Thread Exit(Thread t handle)
00382 {
00383
         Thread* pclThread = (Thread*)handle;
00384
         pclThread->Exit();
00385 }
00386
00387 //-
00388 void Thread_Sleep(uint32_t u32TimeMs_)
00389 {
00390
         Thread::Sleep(u32TimeMs_);
00391 }
00392
00393 #if KERNEL EXTENDED CONTEXT
00394 //-
00395 void* Thread_GetExtendedContext(Thread_t handle)
00396 {
00397
         Thread* pclThread = (Thread*) handle;
00398
         return pclThread->GetExtendedContext();
00399 }
00400
00401 //-
00402 void Thread_SetExtendedContext(Thread_t handle, void* pvData_)
00403 {
00404
         Thread* pclThread = (Thread*)handle;
00405
         pclThread->SetExtendedContext(pvData_);
00406 }
00407 #endif // #if KERNEL_EXTENDED_CONTEXT
00408
00409 //--
00410 void Thread_Yield(void)
00411 {
00412
         Thread::Yield();
00413 }
00414 //---
00415 void Thread_CoopYield(void)
00416 {
00417
         Thread::CoopYield();
00418 }
00419 //--
00420 void Thread_SetID(Thread_t handle, uint8_t u8ID_)
00421 {
00422
         Thread* pclThread = (Thread*) handle;
00423
         pclThread->SetID(u8ID_);
00424 }
00425 //----
00426 uint8_t Thread_GetID(Thread_t handle)
00427 {
00428
         Thread* pclThread = (Thread*)handle;
00429
         return pclThread->GetID();
00430 }
00431 //-
00432 uint16_t Thread_GetStackSlack(Thread_t handle)
00433 {
00434
         Thread* pclThread = (Thread*)handle;
00435
         return pclThread->GetStackSlack();
00436 }
00437 //-
```

```
00438 thread_state_t Thread_GetState(Thread_t handle)
00439 {
00440
          Thread* pclThread = (Thread*) handle;
00441
         return (thread_state_t)pclThread->GetState();
00442 }
00443 //--
00444 // Timer APIs
00445 //---
00446
00447 //----
00448 void Timer_Init(Timer_t handle)
00449 {
00450
          Timer* pclTimer = new ((void*)handle) Timer();
00451
         pclTimer->Init();
00452 }
00453
00454 //--
00455 void Timer_Start(Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_,
      timer_callback_t pfCallback_, void* pvData_)
00456 {
00457
         Timer* pclTimer = (Timer*)handle;
00458
         pclTimer->Start(bRepeat_, u32IntervalMs_, (TimerCallback)pfCallback_, pvData_);
00459 }
00460
00461 //-
00462 void Timer_Stop(Timer_t handle)
00463 {
00464
         Timer* pclTimer = (Timer*)handle;
00465
         pclTimer->Stop();
00466 }
00467
00468 //-
00469 void Timer_Restart(Timer_t handle)
00470 {
00471
         Timer* pclTimer = (Timer*)handle;
00472
         pclTimer->Start();
00473 }
00474
00475 //--
00476 // Semaphore APIs
00477 //--
00478 void Semaphore_Init(Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_)
00479 {
00480
         Semaphore* pclSemaphore = new ((void*)handle) Semaphore();
00481
         pclSemaphore->Init(u16InitVal_, u16MaxVal_);
00482 }
00483
00484 //---
00485 void Semaphore_Post(Semaphore_t handle)
00486 {
00487
         Semaphore* pclSemaphore = (Semaphore*)handle;
00488
         pclSemaphore->Post();
00489 }
00490
00491 //----
00492 void Semaphore_Pend(Semaphore_t handle)
00493 {
00494
          Semaphore* pclSemaphore = (Semaphore*)handle;
00495
         pclSemaphore->Pend();
00496 }
00497
00498 //--
00499 bool Semaphore_TimedPend(Semaphore_t handle, uint32_t u32WaitTimeMS_)
00500 {
00501
          Semaphore* pclSemaphore = (Semaphore*)handle;
00502
         return pclSemaphore->Pend(u32WaitTimeMS_);
00503 }
00504
00505 //-
00506 // Mutex APIs
00507 //-
00508 void Mutex_Init(Mutex_t handle)
00509 {
00510
         Mutex* pclMutex = new ((void*)handle) Mutex();
00511
         pclMutex->Init();
00512 }
00513
00514 //---
00515 void Mutex_Claim(Mutex_t handle)
00516 {
00517
         Mutex* pclMutex = (Mutex*)handle;
         pclMutex->Claim();
00519 }
00520
00521 //---
00522 void Mutex_Release(Mutex_t handle)
00523 {
```

```
Mutex* pclMutex = (Mutex*)handle;
00525
         pclMutex->Release();
00526 }
00527
00528 //---
00529 bool Mutex TimedClaim(Mutex t handle, uint32 t u32WaitTimeMS)
00530 {
00531
          Mutex* pclMutex = (Mutex*)handle;
         return pclMutex->Claim(u32WaitTimeMS_);
00532
00533 }
00534
00535 #if KERNEL EVENT FLAGS
00536 //
00537 // EventFlag APIs
00538 //-
00539 void EventFlag_Init(EventFlag_t handle)
00540 {
00541
          EventFlag* pclFlag = new ((void*)handle) EventFlag();
         pclFlag->Init();
00542
00543 }
00544
00545 //---
00546 uint16_t EventFlag_Wait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_)
00547 {
00548
          EventFlag* pclFlag = (EventFlag*)handle;
00549
          return pclFlag->Wait(u16Mask_, (EventFlagOperation)eMode_);
00550 }
00551
00552 //---
00553 uint16_t EventFlag_TimedWait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_, uint32_t
      u32TimeMS )
00554 {
00555
          EventFlag* pclFlag = (EventFlag*)handle;
00556
          return pclFlag->Wait(u16Mask_, (EventFlagOperation)eMode_, u32TimeMS_);
00557 }
00558
00559 //-
00560 void EventFlag_Set(EventFlag_t handle, uint16_t u16Mask_)
00561 {
00562
          EventFlag* pclFlag = (EventFlag*)handle;
00563
          pclFlag->Set(u16Mask_);
00564 }
00565
00566 //--
00567 void EventFlag_Clear(EventFlag_t handle, uint16_t u16Mask_)
00568 {
00569
          EventFlag* pclFlag = (EventFlag*)handle;
00570
         pclFlag->Clear(u16Mask_);
00571 }
00572
00574 uint16_t EventFlag_GetMask(EventFlag_t handle)
00575 {
00576
         EventFlag* pclFlag = (EventFlag*)handle;
00577
          return pclFlag->GetMask();
00578 }
00579 #endif // #if #if KERNEL_EVENT_FLAGS
00580
00581 //----
00582 // Notification APIs
00583 //--
00584 void Notify_Init(Notify_t handle)
00585 {
00586
          Notify* pclNotify = new ((void*)handle) Notify();
00587
         pclNotify->Init();
00588 }
00589
00590 //--
00591 void Notify_Signal(Notify_t handle)
00592 {
00593
          Notify* pclNotify = (Notify*)handle;
00594
         pclNotify->Signal();
00595 }
00596
00597 //-
00598 void Notify_Wait(Notify_t handle, bool* pbFlag_)
00599 {
00600
          Notify* pclNotify = (Notify*)handle;
00601
         pclNotify->Wait (pbFlag_);
00602 }
00603
00604 //-
00605 bool Notify_TimedWait(Notify_t handle, uint32_t u32WaitTimeMS_, bool* pbFlag_)
00606 {
00607
          Notify* pclNotify = (Notify*)handle;
00608
          return pclNotify->Wait(u32WaitTimeMS_, pbFlag_);
00609 }
```

```
00610
00611 //---
00612 // Atomic Functions
00613 //----
00614 uint8_t Atomic_Set8(uint8_t* pu8Source_, uint8_t u8Val_)
00615 {
00616
          return Atomic::Set(pu8Source_, u8Val_);
00617 }
00618
00619 //---
00620 uint16_t Atomic_Set16(uint16_t* pu16Source_, uint16_t u16Val_)
00621 {
00622
          return Atomic::Set(pul6Source, ul6Val);
00623 }
00624
00625 //---
00626 uint32_t Atomic_Set32(uint32_t* pu32Source_, uint32_t u32Val_)
00627 {
00628
          return Atomic::Set(pu32Source_, u32Val_);
00629 }
00630
00631 //--
00632 uint8_t Atomic_Add8(uint8_t* pu8Source_, uint8_t u8Val_)
00633 {
00634
          return Atomic::Add(pu8Source_, u8Val_);
00635 }
00636
00637 //--
00638 uint16_t Atomic_Add16(uint16_t* pu16Source_, uint16_t u16Val_)
00639 {
00640
          return Atomic::Add(pu16Source, u16Val);
00641 }
00642
00643 //--
00644 uint32_t Atomic_Add32(uint32_t* pu32Source_, uint32_t u32Val_)
00645 {
00646
          return Atomic::Add(pu32Source_, u32Val_);
00647 }
00648
00649 //--
00650 uint8_t Atomic_Sub8(uint8_t* pu8Source_, uint8_t u8Val_)
00651 {
00652
          return Atomic::Sub(pu8Source, u8Val);
00653 }
00654
00655 //--
00656 uint16_t Atomic_Sub16(uint16_t* pu16Source_, uint16_t u16Val_)
00657 {
00658
          return Atomic::Sub(pu16Source_, u16Val_);
00659 }
00660
00661 //---
00662 uint32_t Atomic_Sub32(uint32_t* pu32Source_, uint32_t u32Val_)
00663 {
00664
          return Atomic::Sub(pu32Source_, u32Val_);
00665 }
00666
00667 //---
00668 bool Atomic_TestAndSet(bool* pbLock)
00669 {
00670
         return Atomic::TestAndSet(pbLock);
00671 }
00672
00673 //--
00674 // Message/Message Queue APIs
00675 //--
00676 void Message_Init(Message_t handle)
00677 {
00678
         Message* pclMessage = new ((void*)handle) Message();
00679
         return pclMessage->Init();
00680 }
00681
00682 //--
00683 void Message_SetData(Message_t handle, void* pvData_)
00684 {
00685
         Message* pclMessage = (Message*)handle;
00686
         pclMessage->SetData(pvData_);
00687 }
00688
00689 //---
00690 void* Message_GetData(Message_t handle)
00691 {
00692
          Message* pclMessage = (Message*)handle;
00693
          return pclMessage->GetData();
00694 }
00695
00696 //---
```

```
00697 void Message_SetCode(Message_t handle, uint16_t u16Code_)
00698 {
00699
          Message* pclMessage = (Message*)handle;
00700
          pclMessage->SetCode (u16Code_);
00701 }
00702
00704 uint16_t Message_GetCode(Message_t handle)
00705 {
00706
          Message* pclMessage = (Message*)handle;
          return pclMessage->GetCode();
00707
00708 }
00709
00710 //---
00711 void MessageQueue_Init(MessageQueue_t handle)
00712 {
00713
         MessageQueue* pclMsqQ = new ((void*)handle) MessageQueue();
00714
         pclMsgQ->Init();
00715 }
00716
00717 //---
00718 Message_t MessageQueue_Receive(MessageQueue_t handle)
00719 {
          MessageQueue* pclMsgQ = (MessageQueue*) handle;
00720
00721
          return pclMsgQ->Receive();
00722 }
00723
00724 //--
00725 void MessagePool_Init(MessagePool_t handle)
00726 {
00727
          MessagePool* pclMsgPool = new ((void*)handle) MessagePool();
00728
         pclMsgPool->Init();
00729 }
00730
00731 //--
00732 void MessagePool_Push (MessagePool_t handle,
     Message_t msg)
00733 {
00734
          MessagePool* pclMsgPool = (MessagePool*)handle;
00735
         pclMsgPool->Push((Message*)msg);
00736 }
00737
00738 //--
00739 Message_t MessagePool_Pop(MessagePool_t handle)
00740 {
00741
          MessagePool* pclMsgPool = (MessagePool*)handle;
00742
         return (Message_t)pclMsgPool->Pop();
00743 }
00744
00745 //--
00746 Message_t MessageQueue_TimedReceive(
      MessageQueue_t handle, uint32_t u32TimeWaitMS_)
00747 {
00748
          MessageQueue* pclMsgQ = (MessageQueue*) handle;
00749
          return (Message_t)pclMsgQ->Receive(u32TimeWaitMS_);
00750 }
00751
00752 //---
00753 void MessageQueue_Send(MessageQueue_t handle,
     Message_t hMessage_)
00754 {
00755
          MessageQueue* pclMsq0 = (MessageQueue*) handle;
00756
         pclMsgQ->Send((Message*)hMessage_);
00757 }
00758
00759 //---
00760 uint16_t MessageQueue_GetCount(MessageQueue_t handle)
00761 {
00762
          MessageQueue* pclMsqQ = (MessageQueue*) handle;
00763
         return pclMsgQ->GetCount();
00764 }
00765
00766 //---
00767 // Mailbox APIs
00768 //-
00769 void Mailbox_Init(Mailbox_t handle, void* pvBuffer_, uint16_t u16BufferSize_, uint16_t
      u16ElementSize_)
00770 {
00771
          Mailbox* pclMBox = new ((void*)handle) Mailbox();
00772
          pclMBox->Init(pvBuffer_, u16BufferSize_, u16ElementSize_);
00773 }
00774
00775 //--
00776 bool Mailbox_Send(Mailbox_t handle, void* pvData_)
00777 {
         Mailbox* pclMBox = (Mailbox*)handle;
00778
00779
         return pclMBox->Send(pvData);
```

```
00780 }
00781
00782 //-
00783 bool Mailbox_SendTail(Mailbox_t handle, void* pvData_)
00784 {
00785
         Mailbox* pclMBox = (Mailbox*)handle;
         return pclMBox->SendTail(pvData_);
00786
00787 }
00788
00789 //---
00790 bool Mailbox_TimedSend(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_)
00791 {
00792
         Mailbox* pclMBox = (Mailbox*)handle;
00793
         return pclMBox->Send(pvData_, u32TimeoutMS_);
00794 }
00795
00796 //--
00797 bool Mailbox_TimedSendTail(Mailbox_t handle, void* pvData_, uint32_t
     u32TimeoutMS_)
00798 {
00799
          Mailbox* pclMBox = (Mailbox*)handle;
00800
          return pclMBox->SendTail(pvData_, u32TimeoutMS_);
00801 }
00802
00803 //-
00804 void Mailbox_Receive(Mailbox_t handle, void* pvData_)
00805 {
00806
         Mailbox* pclMBox = (Mailbox*)handle;
00807
         pclMBox->Receive(pvData_);
00808 }
00809
00810 //-
00811 void Mailbox_ReceiveTail(Mailbox_t handle, void* pvData_)
00812 {
00813
         Mailbox* pclMBox = (Mailbox*)handle;
         pclMBox->ReceiveTail(pvData_);
00814
00815 }
00817 //--
00818 bool Mailbox_TimedReceive(Mailbox_t handle, void* pvData_, uint32_t
     u32TimeoutMS_)
00819 {
00820
          Mailbox* pclMBox = (Mailbox*)handle:
00821
         return pclMBox->Receive(pvData_, u32TimeoutMS_);
00822 }
00823
00824 //----
00825 bool Mailbox_TimedReceiveTail(Mailbox_t handle, void* pvData_, uint32_t
     u32TimeoutMS )
00826 {
00827
         Mailbox* pclMBox = (Mailbox*)handle;
00828
         return pclMBox->ReceiveTail(pvData_, u32TimeoutMS_);
00829 }
00830
00831 //----
00832 uint16_t Mailbox_GetFreeSlots(Mailbox_t handle)
00833 {
00834
          Mailbox* pclMBox = (Mailbox*)handle;
00835
         return pclMBox->GetFreeSlots();
00836 }
00837
00838 //---
00839 bool Mailbox_IsFull(Mailbox_t handle)
00840 {
00841
         Mailbox* pclMBox = (Mailbox*)handle;
00842
         return pclMBox->IsFull();
00843 }
00844
00845 //-
00846 bool Mailbox_IsEmpty(Mailbox_t handle)
00847 {
00848
         Mailbox* pclMBox = (Mailbox*)handle;
00849
         return pclMBox->IsEmpty();
00850 }
00851
00852 //--
00853 // Condition Variables
00854 //--
00855 void ConditionVariable_Init(ConditionVariable_t handle)
00856 {
          ConditionVariable* pclCondvar = new ((void*)handle)
00857
     ConditionVariable();
         pclCondvar->Init();
00858
00859 }
00860 //---
00861 void ConditionVariable_Wait(ConditionVariable_t handle,
      Mutex t hMutex )
```

```
00862 {
00863
          ConditionVariable* pclCondvar = (ConditionVariable*) handle;
00864
          Mutex*
                             pclMutex = (Mutex*)hMutex_;
          pclCondvar->Wait(pclMutex);
00865
00866 }
00867 //--
00868 void ConditionVariable_Signal(ConditionVariable_t handle)
00869 {
00870
         ConditionVariable* pclCondvar = (ConditionVariable*) handle;
00871
         pclCondvar->Signal();
00872 }
00873 //-
00874 void ConditionVariable_Broadcast(ConditionVariable_t handle)
00875 {
00876
          ConditionVariable* pclCondvar = (ConditionVariable*) handle;
00877
         pclCondvar->Broadcast();
00878 3
00879 //--
00880 bool ConditionVariable_TimedWait(ConditionVariable_t handle,
      Mutex_t hMutex_, uint32_t u32WaitTimeMS_)
00881 {
          ConditionVariable* pclCondvar = (ConditionVariable*)handle;
Mutex* pclMutex = (Mutex*)hMutex_;
00882
00883
         Mut.ex*
00884
         return pclCondvar->Wait(pclMutex, u32WaitTimeMS_);
00885 }
00886
00887 //-
00888 // Reader-writer locks
00889 //--
00890 void ReaderWriterLock_Init(ReaderWriterLock_t handle)
00891 {
00892
          ReaderWriterLock* pclReaderWriter = new ((void*)handle)
      ReaderWriterLock();
00893
        pclReaderWriter->Init();
00894 }
00895
00896 //--
00897 void ReaderWriterLock_AcquireReader(
      ReaderWriterLock_t handle)
00898 {
00899
         ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00900
         pclReaderWriter->AcquireReader();
00901 }
00902
00904 void ReaderWriterLock_ReleaseReader(
     ReaderWriterLock_t handle)
00905 {
00906
         ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00907
         pclReaderWriter->ReleaseReader();
00908 }
00909
00910 //---
00911 void ReaderWriterLock AcquireWriter(
      ReaderWriterLock_t handle)
00912 {
00913
          ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00914
         pclReaderWriter->AcquireWriter();
00915 }
00916
00917 //----
00918 void ReaderWriterLock_ReleaseWriter(
      ReaderWriterLock_t handle)
00919 {
00920
          ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00921
         pclReaderWriter->ReleaseWriter();
00922 }
00923
00924 //-
00925 bool ReaderWriterLock_TimedAcquireWriter(
      ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)
00926 {
00927
          ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00928
          return pclReaderWriter->AcquireWriter(u32TimeoutMs_);
00929 }
00930
00931 //--
00932 bool ReaderWriterLock_TimedAcquireReader(
      ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)
00933 {
00934
          ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
00935
          return pclReaderWriter->AcquireReader(u32TimeoutMs_);
00936 }
```

# 20.3 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/fake\_types.h File Reference

C-struct definitions that mirror.

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "mark3cfg.h"
```

## 20.3.1 Detailed Description

C-struct definitions that mirror.

This header contains a set of "fake" structures that have the same memory layout as the kernel objects in C++ (taking into account inheritence, etc.). These are used for sizing the opaque data blobs that are declared in C, which then become instantiated as C++ kernel objects via the bindings provided.

Definition in file fake\_types.h.

# 20.4 fake\_types.h

```
00002
00003
00004 |
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00026 #include <stdint.h>
00027 #include <stddef.h>
00028 #include <stdbool.h>
00029 #include "mark3cfg.h"
00030
00031 #pragma once
00032 #if defined(__cplusplus)
00034 extern "C" {
00035 #endif
00036
00037 //---
00038 typedef struct {
       void* prev;
void* next;
00039
00040
00041 } Fake_LinkedListNode;
00042
00043 //--
00044 typedef struct {
       void* head;
void* tail;
00045
00046
00047 } Fake_LinkedList;
00048
00049 //--
00050 typedef struct {
00051
          Fake_LinkedListNode fake_node;
00052
          Fake_LinkedList fake_list;
00053
          PORT_PRIO_TYPE m_uXPriority;
00054
          void*
                           m_pclMap;
00055 } Fake_ThreadList;
00056
00057 //---
```

20.4 fake\_types.h 359

```
00058 typedef struct {
        Fake_LinkedListNode m_ll_node;
                      m_u8Initialized;
m_u8Flags;
          uint8_t
00060
00061
          uint8 t
                              m_pfCallback;
m_u32Interval;
m_u32TimeLeft;
m_pclOwner;
00062
          void*
00063
          uint32_t
00064
          uint32_t
00065
        void*
void*
00066
                               m_pvData;
00067 } Fake_Timer;
00068
00069 //----
00070 typedef struct {
00071
         Fake_LinkedListNode m_ll_node;
                    m_pwStackTop;
00072
          K_WORD*
00073
          K WORD*
                               m_pwStack;
00074
         uint8 t
                               m_u8ThreadID;
        PORT_PRIO_TYPE
PORT_PRIO_TYPE
                              m_uXPriority;
m_uXCurPriority;
00075
00077
          uint8_t
                                m_eState;
00078 #if KERNEL_EXTENDED_CONTEXT
00079 void* m_pvExtendedContext;
00080 #endif // #if KERNEL_EXTENDED_CONTEXT
00081 #if KERNEL_NAMED_THREADS
00082 const char* m_szName;
00083 #endif // #if KERNEL_NAMED_THREADS
00084 uint16_t m_u16StackSize;
         void* m_pclCurrent;
void* m_pclOwner;
00085
00086
        void*
                  m_pfEntryPoint;
00087
00088
                    m_pvArq;
          void*
00089 #if KERNEL_ROUND_ROBIN
00090
          uint16_t m_u16Quantum;
00091 #endif // #if KERNEL_ROUND_ROBIN 00092 #if KERNEL_EVENT_FLAGS
00093 uint16_t m_u16FlagMask;
00094 uint8_t m_eFlagMode;
00095 #endif // #if KERNEL_EVENT_FLAGS
00096 Fake_Timer m_clTimer;
00097 bool m_bExpired
                    m_bExpired;
00098 } Fake_Thread;
00099
00100 //---
00101 typedef struct {
      Fake_ThreadList thread_list;
00103
          00104
          uint16 t
                           m_u16Value;
                          m_u16MaxValue;
00105
          uint16 t
00106 } Fake_Semaphore;
00107
00109 typedef struct {
        Fake_ThreadList thread_list;
00110
          00111
00112
                         m_ubReady;
m_bRecursive;
m_u8MaxPri;
m_pclOwner;
          bool
00113
        bool
          uint8_t
00115
00116
          void*
00117 } Fake_Mutex;
00118
00119 //---
00120 typedef struct {
00121 Fake_LinkedListNode list_node;
                     m_pvData;
m_u16Code;
00122
          void*
00123
          uint16_t
00124 } Fake_Message;
00125
00126 //----
00127 typedef struct {
00128 Fake_Semaphore m_clSemaphore;
00129 Fake_LinkedList m_clLinkList;
00130 } Fake_MessageQueue;
00131
00132 //----
00133 typedef struct {
          Fake_LinkedList m_clList;
00134
00135 } Fake_MessagePool;
00136
00137 //----
00138 typedef struct {
          uint16_t
                          m_u16Head;
          uint16_t
                          m_u16Tail;
00140
                           m_u16Count;
00141
          uint16_t
00142
          uint16_t
                          m_u16Free;
                          m_u16ElementSize;
00143
          uint16_t
00144
                          m_pvBuffer;
          void*
```

```
Fake_Semaphore m_clRecvSem;
           Fake_Semaphore m_clSendSem;
00147 } Fake_Mailbox;
00148
00149 //----
00150 typedef struct {
00151
           Fake_ThreadList thread_list;
          uint8_t m_u8Initialized;
bool m_bPending;
00153
00154 } Fake_Notify;
00155
00156 //----
00157 typedef struct {
00158 Fake_ThreadList thread_list;
00159 uint8_t m_u8Initialized;
00160 uint16_t m_u16EventFlag;
00161 } Fake_EventFlag;
00162
00163 //---
00164 typedef struct {
        Fake_Mutex m_clGlobalMutex;
Fake_Mutex m_clReaderMutex;
uint8_t m_u8ReadCount;
00165
00166
00167
00168 } Fake_ReaderWriterLock;
00169
00170 //---
00171 typedef struct {
00172 Fake_Mutex m_clMutex;
00173 Fake_Semaphore m_clSemaphore;
00174
           uint8 t
                            m u8Waiters:
00175 } Fake_ConditionVariable;
00177 #if defined(__cplusplus)
00178 }
00179 #endif
00180
```

# 20.5 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/mark3c.h File Reference

Header providing C-language API bindings for the Mark3 kernel.

```
#include "mark3cfg.h"
#include "fake_types.h"
#include <stdint.h>
#include <stdbool.h>
```

#### **Macros**

- #define THREAD\_SIZE (sizeof(Fake\_Thread))
- #define TIMER SIZE (sizeof(Fake Timer))
- #define SEMAPHORE\_SIZE (sizeof(Fake\_Semaphore))
- #define MUTEX\_SIZE (sizeof(Fake\_Mutex))
- #define MESSAGE\_SIZE (sizeof(Fake\_Message))
- #define MESSAGEQUEUE\_SIZE (sizeof(Fake\_MessageQueue))
- #define MAILBOX\_SIZE (sizeof(Fake\_Mailbox))
- #define NOTIFY\_SIZE (sizeof(Fake\_Notify))
- #define MESSAGEPOOL\_SIZE (sizeof(Fake\_MessagePool))
- #define CONDITIONVARIABLE\_SIZE (sizeof(Fake\_ConditionVariable))
- #define READERWRITERLOCK SIZE (sizeof(Fake ReaderWriterLock))
- #define TOKEN\_1(x, y) x##y
- #define TOKEN\_2(x, y) TOKEN\_1(x, y)
- #define WORD\_ROUND(x) (((x) + (sizeof(K\_WORD) 1)) / sizeof(K\_WORD))

```
    #define DECLARE_THREAD(name)

    #define DECLARE_TIMER(name)

    • #define DECLARE SEMAPHORE(name)

    #define DECLARE_MUTEX(name)

    • #define DECLARE_MESSAGE(name)
    • #define DECLARE_MESSAGEPOOL(name)
    • #define DECLARE MESSAGEQUEUE(name)

    #define DECLARE MAILBOX(name)

    #define DECLARE NOTIFY(name)

    #define DECLARE_CONDITIONVARIABLE(name)

    #define DECLARE_READERWRITERLOCK(name)

Typedefs

    typedef void * Mailbox_t

         Mailbox opaque handle data type.
    typedef void * Message_t
         Message opaque handle data type.

    typedef void * MessagePool_t

         MessagePool opaque handle data type.

    typedef void * MessageQueue_t

         MessageQueue opaque handle data type.
    typedef void * Mutex_t
         Mutex opaque handle data type.
    typedef void * Notify_t
         Notification object opaque handle data type.
    typedef void * Semaphore_t
         Semaphore opaque handle data type.
    typedef void * Thread_t
         Thread opaque handle data type.
    typedef void * Timer_t
         Timer opaque handle data type.

    typedef void * ConditionVariable_t

         Condition Variable opaque handle data type.

    typedef void * ReaderWriterLock t

         Reader-writer-lock opaque handle data type.

    typedef void(* kernel_debug_print_t) (const char *szString_)

    typedef void(* panic_func_t) (uint16_t u16PanicCode_)

    typedef void(* thread entry func t) (void *pvArg )

    typedef void(* timer_callback_t) (Thread_t hOwner_, void *pvData_)
```

#### **Enumerations**

```
    enum thread_state_t {
        THREAD_STATE_EXIT = 0, THREAD_STATE_READY, THREAD_STATE_BLOCKED, THREAD_STATE
        _STOP,
        THREAD_STATE_INVALID }
```

#### **Functions**

void \* Alloc\_Memory (size\_t eSize\_)

Alloc\_Memory.

void Free\_Memory (void \*pvObject\_)

Free\_Memory.

• Semaphore\_t Alloc\_Semaphore (void)

Alloc Semaphore.

- void Free Semaphore (Semaphore t handle)
- Mutex\_t Alloc\_Mutex (void)

Alloc\_Mutex.

- void Free\_Mutex (Mutex\_t handle)
- Message t Alloc Message (void)

Alloc\_Message.

- void Free\_Message (Message\_t handle)
- MessageQueue\_t Alloc\_MessageQueue (void)

Alloc\_MessageQueue.

- void Free\_MessageQueue (MessageQueue\_t handle)
- MessagePool t Alloc MessagePool (void)
- void Free\_MessagePool (MessagePool\_t handle)
- Notify\_t Alloc\_Notify (void)

Alloc\_Notify.

- void Free\_Notify (Notify\_t handle)
- Mailbox\_t Alloc\_Mailbox (void)

Alloc Mailbox.

- void Free\_Mailbox (Mailbox\_t handle)
- Thread\_t Alloc\_Thread (void)

Alloc\_Thread.

- · void Free Thread (Thread thandle)
- Timer\_t Alloc\_Timer (void)

Alloc\_Timer.

- void Free\_Timer (Timer\_t handle)
- void Kernel\_Init (void)

Kernel\_Init.

void Kernel\_Start (void)

Kernel Start.

bool Kernel\_IsStarted (void)

Kernel IsStarted.

void Kernel\_SetPanic (panic\_func\_t pfPanic\_)

Kernel\_SetPanic.

bool Kernel\_IsPanic (void)

Kernel\_IsPanic.

• void Kernel Panic (uint16 t u16Cause )

Kernel\_Panic.

• uint32\_t Kernel\_GetTicks (void)

Kernel GetTicks.

void Scheduler\_Enable (bool bEnable\_)

Scheduler\_Enable.

• bool Scheduler\_IsEnabled (void)

Scheduler\_IsEnabled.

Thread\_t Scheduler\_GetCurrentThread (void)

Scheduler\_GetCurrentThread.

```
    void Thread_Init (Thread_t handle, K_WORD *pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE u←

  XPriority_, thread_entry_func_t pfEntryPoint_, void *pvArg_)
     Thread_Init.

    void Thread Start (Thread t handle)

     Thread_Start.

    void Thread_Stop (Thread_t handle)

     Thread Stop.

    PORT_PRIO_TYPE Thread_GetPriority (Thread_t handle)

     Thread_GetPriority.

    PORT_PRIO_TYPE Thread_GetCurPriority (Thread_t handle)

     Thread_GetCurPriority.

    void Thread_SetPriority (Thread_t handle, PORT_PRIO_TYPE uXPriority_)

     Thread_SetPriority.
void Thread_Exit (Thread_t handle)
     Thread Exit.

    void Thread_Sleep (uint32_t u32TimeMs_)

     Thread_Sleep.

    void Thread_Yield (void)

     Thread Yield.

    void Thread_CoopYield (void)

     Thread_CoopYield.

    void Thread SetID (Thread t handle, uint8 t u8ID )

     Thread_SetID.

    uint8_t Thread_GetID (Thread_t handle)

     Thread GetID.

    thread_state_t Thread_GetState (Thread_t handle)

     Thread GetState.

    void Timer_Init (Timer_t handle)

     Timer Init.
• void Timer_Start (Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_, timer_callback_t pfCallback_,
  void *pvData_)
     Timer Start.

    void Timer_Restart (Timer_t handle)

     Timer Restart.

    void Timer_Stop (Timer_t handle)

     Timer Stop.
• void Semaphore_Init (Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_)
     Semaphore Init.

    void Semaphore_Post (Semaphore_t handle)

     Semaphore_Post.
· void Semaphore Pend (Semaphore t handle)
     Semaphore Pend.
• bool Semaphore_TimedPend (Semaphore_t handle, uint32_t u32WaitTimeMS_)
     Semaphore_TimedPend.

    void Mutex Init (Mutex t handle)

     Mutex Init.

    void Mutex Claim (Mutex t handle)

     Mutex_Claim.

    void Mutex Release (Mutex t handle)

     Mutex Release.

    bool Mutex_TimedClaim (Mutex_t handle, uint32_t u32WaitTimeMS_)
```

```
Mutex_TimedClaim.

    void Notify_Init (Notify_t handle)

     Notify Init.

    void Notify Signal (Notify t handle)

     Notify_Signal.

    void Notify_Wait (Notify_t handle, bool *pbFlag_)

     Notify Wait.
• bool Notify TimedWait (Notify t handle, uint32 t u32WaitTimeMS, bool *pbFlag)
     Notify TimedWait.

    uint8_t Atomic_Set8 (uint8_t *pu8Source_, uint8_t u8Val_)

    uint16_t Atomic_Set16 (uint16_t *pu16Source_, uint16_t u16Val_)

     Atomic Set16.
• uint32_t Atomic_Set32 (uint32_t *pu32Source_, uint32_t u32Val_)
     Atomic_Set32.
• uint8 t Atomic Add8 (uint8 t *pu8Source , uint8 t u8Val )
     Atomic_Add8.
uint16_t Atomic_Add16 (uint16_t *pu16Source_, uint16_t u16Val_)
     Atomic Add16.

    uint32 t Atomic Add32 (uint32 t *pu32Source , uint32 t u32Val )

     Atomic Add32.
• uint8_t Atomic_Sub8 (uint8_t *pu8Source_, uint8_t u8Val )
     Atomic Sub8.

    uint16_t Atomic_Sub16 (uint16_t *pu16Source_, uint16_t u16Val_)

     Atomic_Sub16.
uint32_t Atomic_Sub32 (uint32_t *pu32Source_, uint32_t u32Val_)
     Atomic_Sub32.

    bool Atomic_TestAndSet (bool *pbLock)

     Atomic_TestAndSet.

    void Message Init (Message t handle)

     Message Init.

    void Message SetData (Message t handle, void *pvData )

     Message SetData.

    void * Message_GetData (Message_t handle)

     Message GetData.

    void Message_SetCode (Message_t handle, uint16_t u16Code_)

     Message_SetCode.
• uint16_t Message_GetCode (Message_t handle)
     Message_GetCode.

    void MessageQueue_Init (MessageQueue_t handle)

     MessageQueue_Init.
• Message t MessageQueue Receive (MessageQueue t handle)
     MessageQueue_Receive.

    Message_t MessageQueue_TimedReceive (MessageQueue_t handle, uint32_t u32TimeWaitMS_)

     MessageQueue TimedReceive.

    void MessageQueue_Send (MessageQueue_t handle, Message_t hMessage_)

     MessageQueue Send.

    uint16_t MessageQueue_GetCount (MessageQueue_t handle)

     MessageQueue_GetCount.
• void MessagePool_Init (MessagePool_t handle)
     MessagePool_Init.
```

```
    void MessagePool_Push (MessagePool_t handle, Message_t msg)

     MessagePool_Push.

    Message t MessagePool Pop (MessagePool t handle)

     MessagePool_Pop.

    void Mailbox_Init (Mailbox_t handle, void *pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)

     Mailbox Init.

    bool Mailbox Send (Mailbox t handle, void *pvData )

     Mailbox Send.

    bool Mailbox_SendTail (Mailbox_t handle, void *pvData_)

     Mailbox_SendTail.

    bool Mailbox TimedSend (Mailbox t handle, void *pvData , uint32 t u32TimeoutMS )

     Mailbox_TimedSend.

    bool Mailbox_TimedSendTail (Mailbox_t handle, void *pvData_, uint32_t u32TimeoutMS_)

     Mailbox TimedSendTail.

    void Mailbox Receive (Mailbox t handle, void *pvData )

     Mailbox Receive.

    void Mailbox_ReceiveTail (Mailbox_t handle, void *pvData_)

     Mailbox ReceiveTail.

    bool Mailbox TimedReceive (Mailbox t handle, void *pvData , uint32 t u32TimeoutMS )

     Mailbox_TimedReceive.

    bool Mailbox TimedReceiveTail (Mailbox t handle, void *pvData , uint32 t u32TimeoutMS )

     Mailbox_TimedReceiveTail.

    uint16 t Mailbox GetFreeSlots (Mailbox t handle)

     Mailbox GetFreeSlots.

    bool Mailbox_IsFull (Mailbox_t handle)

     Mailbox IsFull.

    bool Mailbox_IsEmpty (Mailbox_t handle)

     Mailbox_IsEmpty.

    void ConditionVariable_Init (ConditionVariable_t handle)

     ConditionVariable_Init.

    void ConditionVariable Wait (ConditionVariable t handle, Mutex t hMutex )

     ConditionVariable Wait.
• void ConditionVariable_Signal (ConditionVariable_t handle)
     ConditionVariable Signal.

    void ConditionVariable_Broadcast (ConditionVariable_t handle)

     ConditionVariable Broadcast.

    bool ConditionVariable_TimedWait (ConditionVariable_t handle, Mutex_t hMutex_, uint32_t u32WaitTime

 MS)
     ConditionVariable_TimedWait.

    void ReaderWriterLock_Init (ReaderWriterLock_t handle)

     ReaderWriterLock_Init.

    void ReaderWriterLock AcquireReader (ReaderWriterLock t handle)

     ReaderWriterLock_AcquireReader.

    void ReaderWriterLock_ReleaseReader (ReaderWriterLock_t handle)

     ReaderWriterLock_ReleaseReader.

    void ReaderWriterLock AcquireWriter (ReaderWriterLock t handle)

     ReaderWriterLock_AcquireWriter.

    void ReaderWriterLock ReleaseWriter (ReaderWriterLock t handle)

     ReaderWriterLock_ReleaseWriter.

    bool ReaderWriterLock TimedAcquireWriter (ReaderWriterLock t handle, uint32 t u32TimeoutMs)

     ReaderWriterLock TimedAcquireWriter.
```

```
    bool ReaderWriterLock_TimedAcquireReader (ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)
    ReaderWriterLock_TimedAcquireReader.
```

• void Kernel\_SetDebugPrintFunction (kernel\_debug\_print\_t pfPrintFunction\_)

Kernel\_SetDebugPrintFunction.

void Kernel DebugPrint (const char \*szString )

KernelDebug\_DebugPrint.

## 20.5.1 Detailed Description

Header providing C-language API bindings for the Mark3 kernel.

Definition in file mark3c.h.

## 20.5.2 Macro Definition Documentation

#### 20.5.2.1 CONDITIONVARIABLE\_SIZE

```
#define CONDITIONVARIABLE_SIZE (sizeof(Fake_ConditionVariable))
```

Definition at line 79 of file mark3c.h.

#### 20.5.2.2 DECLARE\_CONDITIONVARIABLE

## Value:

Definition at line 161 of file mark3c.h.

#### 20.5.2.3 DECLARE\_MAILBOX

## Value:

Definition at line 147 of file mark3c.h.

#### 20.5.2.4 DECLARE\_MESSAGE

#### Value:

Definition at line 135 of file mark3c.h.

## 20.5.2.5 DECLARE\_MESSAGEPOOL

```
\begin{tabular}{ll} \# define \ \ DECLARE\_MESSAGEPOOL (\\ name \ ) \end{tabular}
```

#### Value:

Definition at line 139 of file mark3c.h.

#### 20.5.2.6 DECLARE\_MESSAGEQUEUE

```
\begin{tabular}{ll} \# define & DECLARE\_MESSAGEQUEUE ( \\ & name & ) \end{tabular}
```

#### Value:

Definition at line 143 of file mark3c.h.

## 20.5.2.7 DECLARE\_MUTEX

```
\begin{tabular}{ll} \#define & DECLARE\_MUTEX\,(\\ & \textit{name} \end{tabular} \label{eq:define}
```

#### Value:

```
K_WORD TOKEN_2(__mutex_, name)[WORD_ROUND(MUTEX_SIZE)];
Mutex_t name = (Mutex_t)TOKEN_2(__mutex_, name);
```

Definition at line 131 of file mark3c.h.

#### 20.5.2.8 DECLARE\_NOTIFY

#### Value:

```
K_WORD TOKEN_2(__notify_, name)[WORD_ROUND(NOTIFY_SIZE)];
Notify_t name = (Notify_t)TOKEN_2(__notify_, name);
```

Definition at line 151 of file mark3c.h.

## 20.5.2.9 DECLARE\_READERWRITERLOCK

## Value:

Definition at line 165 of file mark3c.h.

## 20.5.2.10 DECLARE\_SEMAPHORE

#### Value:

Definition at line 127 of file mark3c.h.

## 20.5.2.11 DECLARE\_THREAD

#### Value:

```
K_WORD TOKEN_2(__thread_, name)[WORD_ROUND(THREAD_SIZE)];
Thread_t name = (Thread_t)TOKEN_2(__thread_, name);
```

Definition at line 119 of file mark3c.h.

# 20.5.2.12 DECLARE\_TIMER

```
\begin{tabular}{ll} \# define & DECLARE\_TIMER ( \\ & \textit{name} \end{tabular} ) \end{tabular}
```

## Value:

Definition at line 123 of file mark3c.h.

#### 20.5.2.13 MAILBOX\_SIZE

```
#define MAILBOX_SIZE (sizeof(Fake_Mailbox))
```

Definition at line 73 of file mark3c.h.

```
20.5.2.14 MESSAGE_SIZE
#define MESSAGE_SIZE (sizeof(Fake_Message))
Definition at line 71 of file mark3c.h.
20.5.2.15 MESSAGEPOOL_SIZE
#define MESSAGEPOOL_SIZE (sizeof(Fake_MessagePool))
Definition at line 78 of file mark3c.h.
20.5.2.16 MESSAGEQUEUE_SIZE
#define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
Definition at line 72 of file mark3c.h.
20.5.2.17 MUTEX_SIZE
#define MUTEX_SIZE (sizeof(Fake_Mutex))
Definition at line 70 of file mark3c.h.
20.5.2.18 NOTIFY SIZE
#define NOTIFY_SIZE (sizeof(Fake_Notify))
Definition at line 74 of file mark3c.h.
20.5.2.19 READERWRITERLOCK_SIZE
#define READERWRITERLOCK_SIZE (sizeof(Fake_ReaderWriterLock))
```

Definition at line 80 of file mark3c.h.

```
20.5.2.20 SEMAPHORE_SIZE
```

```
#define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
```

Definition at line 69 of file mark3c.h.

#### 20.5.2.21 THREAD\_SIZE

```
#define THREAD_SIZE (sizeof(Fake_Thread))
```

Definition at line 67 of file mark3c.h.

#### 20.5.2.22 TIMER\_SIZE

```
#define TIMER_SIZE (sizeof(Fake_Timer))
```

Definition at line 68 of file mark3c.h.

## 20.5.2.23 TOKEN\_1

```
#define TOKEN_1(  x, \\ y ) \  \  x \# y
```

Definition at line 112 of file mark3c.h.

# 20.5.2.24 TOKEN\_2

Definition at line 113 of file mark3c.h.

## 20.5.2.25 WORD\_ROUND

```
#define WORD_ROUND(  x \ ) \ (((x) \ + \ (sizeof(K_WORD) \ - \ 1)) \ / \ sizeof(K_WORD))
```

Definition at line 117 of file mark3c.h.

# 20.5.3 Typedef Documentation

Definition at line 41 of file mark3c.h.

```
20.5.3.1 ConditionVariable_t
typedef void* ConditionVariable_t
Condition Variable opaque handle data type.
Definition at line 48 of file mark3c.h.
20.5.3.2 kernel_debug_print_t
typedef void(* kernel_debug_print_t) (const char *szString_)
Definition at line 62 of file mark3c.h.
20.5.3.3 Mailbox_t
typedef void* Mailbox_t
Mailbox opaque handle data type.
Definition at line 39 of file mark3c.h.
20.5.3.4 Message_t
typedef void* Message_t
Message opaque handle data type.
Definition at line 40 of file mark3c.h.
20.5.3.5 MessagePool_t
typedef void* MessagePool_t
MessagePool opaque handle data type.
```

```
20.5.3.6 MessageQueue_t
typedef void* MessageQueue_t
MessageQueue opaque handle data type.
Definition at line 42 of file mark3c.h.
20.5.3.7 Mutex_t
typedef void* Mutex_t
Mutex opaque handle data type.
Definition at line 43 of file mark3c.h.
20.5.3.8 Notify_t
typedef void* Notify_t
Notification object opaque handle data type.
Definition at line 44 of file mark3c.h.
20.5.3.9 panic_func_t
typedef void(* panic_func_t) (uint16_t u16PanicCode_)
Definition at line 282 of file mark3c.h.
20.5.3.10 ReaderWriterLock_t
typedef void* ReaderWriterLock_t
Reader-writer-lock opaque handle data type.
Definition at line 49 of file mark3c.h.
```

```
20.5.3.11 Semaphore_t
typedef void* Semaphore_t
Semaphore opaque handle data type.
Definition at line 45 of file mark3c.h.
20.5.3.12 thread_entry_func_t
typedef void(* thread_entry_func_t) (void *pvArg_)
Definition at line 401 of file mark3c.h.
20.5.3.13 Thread_t
typedef void* Thread_t
Thread opaque handle data type.
Definition at line 46 of file mark3c.h.
20.5.3.14 timer_callback_t
typedef void(* timer_callback_t) (Thread_t hOwner_, void *pvData_)
Definition at line 570 of file mark3c.h.
20.5.3.15 Timer_t
typedef void* Timer_t
Timer opaque handle data type.
Definition at line 47 of file mark3c.h.
20.5.4 Enumeration Type Documentation
20.5.4.1 thread_state_t
enum thread_state_t
```

Define the various states that a thread can be in

#### Enumerator

THREAD_STATE_EXIT	
THREAD_STATE_READY	!< Thread has terminated via exit path
THREAD_STATE_BLOCKED	!< Thread is ready to run
THREAD_STATE_STOP	!< Thread is blocked on a blocking call
THREAD_STATE_INVALID	!< Thread has been manually stopped !< Invalid thread state

Definition at line 101 of file mark3c.h.

# 20.5.5 Function Documentation

```
20.5.5.1 Alloc_Mailbox()
```

Alloc\_Mailbox.

See also

Mailbox\* AutoAlloc::NewMailbox()

#### Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 123 of file mark3c.cpp.

## 20.5.5.2 Alloc\_Memory()

Alloc\_Memory.

See also

void\* AutoAlloc::NewRawData(size\_t sSize\_)

#### **Parameters**

e⊷	Size in bytes to allocate from the one-time-allocate heap
Size⊷	
_	

#### Returns

Pointer to an allocated blob of memory, or nullptr if heap exhausted

#### 20.5.5.3 Alloc\_Message()

Alloc\_Message.

See also

AutoAlloc::NewMessage()

#### **Returns**

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 76 of file mark3c.cpp.

## 20.5.5.4 Alloc\_MessagePool()

Definition at line 100 of file mark3c.cpp.

#### 20.5.5.5 Alloc\_MessageQueue()

 $Alloc\_Message Queue.$ 

See also

MesageQueue\* AutoAlloc::NewMessageQueue()

#### Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 88 of file mark3c.cpp.

```
20.5.5.6 Alloc_Mutex()
Mutex_t Alloc_Mutex (
              void )
Alloc_Mutex.
See also
     Mutex* AutoAlloc::NewMutex()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 52 of file mark3c.cpp.
20.5.5.7 Alloc_Notify()
Notify_t Alloc_Notify (
              void )
Alloc_Notify.
See also
     Notify* AutoAlloc::NewNotify()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 112 of file mark3c.cpp.
20.5.5.8 Alloc_Semaphore()
Semaphore_t Alloc_Semaphore (
              void )
Alloc_Semaphore.
See also
     Semaphore* AutoAlloc::NewSemaphore()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
```

Definition at line 40 of file mark3c.cpp.

```
20.5.5.9 Alloc_Thread()
Thread_t Alloc_Thread (
              void )
Alloc_Thread.
See also
     Thread* AutoAlloc::NewThread()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 156 of file mark3c.cpp.
20.5.5.10 Alloc_Timer()
Timer_t Alloc_Timer (
              void )
Alloc_Timer.
See also
     Timer* AutoAlloc::NewTimer()
Returns
     Handle to an allocated object, or nullptr if heap exhausted
Definition at line 167 of file mark3c.cpp.
20.5.5.11 Atomic_Add16()
uint16_t Atomic_Add16 (
              uint16_t * pu16Source_,
              uint16_t u16Val_ )
Atomic_Add16.
See also
```

uint16\_t Atomic::Add(uint16\_t \*pu16Source\_, uint16\_t u16Val\_)

#### **Parameters**

pu16⇔	Pointer to a variable
Source_	
u16Val_	Value to add to the variable

## Returns

Previously-held value in pu16Source\_

Definition at line 638 of file mark3c.cpp.

## 20.5.5.12 Atomic\_Add32()

Atomic\_Add32.

#### See also

```
uint32_t Atomic::Add(uint32_t *pu32Source_, uint32_t u32Val_)
```

## Parameters

pu32←	Pointer to a variable
Source_	Value to add to the variable
u32Val_	Value to add to the variable

## Returns

Previously-held value in pu32Source\_

Definition at line 644 of file mark3c.cpp.

## 20.5.5.13 Atomic\_Add8()

Atomic\_Add8.

## See also

```
uint8_t Atomic::Add(uint8_t *pu8Source_, uint8_t u8Val_)
```

#### **Parameters**

pu8←	Pointer to a variable
Source_	
u8Val_	Value to add to the variable

## Returns

Previously-held value in pu8Source\_

Definition at line 632 of file mark3c.cpp.

## 20.5.5.14 Atomic\_Set16()

Atomic\_Set16.

## See also

```
uint16_t Atomic::Set(uint16_t *pu16Source_, uint16_t u16Val_)
```

## Parameters

pu16↔ Source_	Pointer to a variable to set the value of
u16Val_	New value to set in the variable

#### Returns

Previously-set value

Definition at line 620 of file mark3c.cpp.

## 20.5.5.15 Atomic\_Set32()

Atomic\_Set32.

## See also

```
uint32_t Atomic::Set(uint32_t *pu32Source_, uint32_t u32Val_)
```

#### **Parameters**

pu32← Source_	Pointer to a variable to set the value of
u32Val_	New value to set in the variable

## Returns

Previously-set value

Definition at line 626 of file mark3c.cpp.

# 20.5.5.16 Atomic\_Set8()

Atomic\_Set8.

# See also

```
uint8_t Atomic::Set(uint8_t *pu8Source_, uint8_t u8Val_)
```

## **Parameters**

pu8← Source_	Pointer to a variable to set the value of
u8Val_	New value to set in the variable

## Returns

Previously-set value

Definition at line 614 of file mark3c.cpp.

# 20.5.5.17 Atomic\_Sub16()

 $Atomic\_Sub16.$ 

## See also

```
uint16_t Atomic::Sub(uint16_t *pu16Source_, uint16_t u16Val_)
```

## **Parameters**

<i>pu</i> 16⇔	Pointer to a variable
Source_	
u16Val_	Value to subtract from the variable

# Returns

Previously-held value in pu16Source\_

Definition at line 656 of file mark3c.cpp.

# 20.5.5.18 Atomic\_Sub32()

Atomic\_Sub32.

## See also

```
uint32_t Atomic::Sub(uint32_t *pu32Source_, uint32_t u32Val_)
```

# Parameters

pu32⇔ Source_	Pointer to a variable
u32Val_	Value to subtract from the variable

## Returns

Previously-held value in pu32Source\_

Definition at line 662 of file mark3c.cpp.

# 20.5.5.19 Atomic\_Sub8()

Atomic\_Sub8.

## See also

```
uint8_t Atomic::Sub(uint8_t *pu8Source_, uint8_t u8Val_)
```

#### **Parameters**

pu8←	Pointer to a variable
Source_	
u8Val_	Value to subtract from the variable

# Returns

Previously-held value in pu8Source\_

Definition at line 650 of file mark3c.cpp.

## 20.5.5.20 Atomic\_TestAndSet()

 $Atomic\_TestAndSet.$ 

#### See also

bool Atomic::TestAndSet(bool \*pbLock)

#### **Parameters**

pbLock	Pointer to a value to test against. This will always be set to "true" at the end of a call to TestAndSet.
--------	---

# Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 668 of file mark3c.cpp.

# 20.5.5.21 ConditionVariable\_Broadcast()

```
\begin{tabular}{ll} \begin{tabular}{ll} void & ConditionVariable\_Broadcast & ( \\ & ConditionVariable\_t & handle & ) \end{tabular}
```

 $Condition Variable\_Broadcast.$ 

## See also

void ConditionVariable::Broadcast()

#### **Parameters**

handle Handle of the condition variable object
--

Definition at line 874 of file mark3c.cpp.

## 20.5.5.22 ConditionVariable\_Init()

ConditionVariable\_Init.

See also

void ConditionVariable::Init()

#### **Parameters**

	handle	Handle of the condition variable object
--	--------	---

Definition at line 855 of file mark3c.cpp.

# 20.5.5.23 ConditionVariable\_Signal()

ConditionVariable\_Signal.

See also

void ConditionVariable::Signal()

#### **Parameters**

Handle of the condition variable object
---

Definition at line 868 of file mark3c.cpp.

## 20.5.5.24 ConditionVariable\_TimedWait()

ConditionVariable\_TimedWait.

## See also

bool ConditionVariable::Wait(Mutex\* pclMutex\_, uint32\_t u32WaitTimeMS\_)

#### **Parameters**

handle	Handle of the condition variable object
hMutex_	Handle of the mutex to lock on acquisition of the condition variable
u32WaitTimeM⊷ S_	Maximum time to wait for object

## Returns

true on success, false on timeout

Definition at line 880 of file mark3c.cpp.

## 20.5.5.25 ConditionVariable\_Wait()

ConditionVariable\_Wait.

## See also

void ConditionVariable::Wait(Mutex\* pclMutex\_)

# Parameters

handle	Handle of the condition variable object
h⊷ Mutex⊷	Handle of the mutex to lock on acquisition of the condition variable

Definition at line 861 of file mark3c.cpp.

```
20.5.5.26 Free_Mailbox()
```

Definition at line 128 of file mark3c.cpp.

# 20.5.5.27 Free\_Memory()

```
void Free_Memory (
     void * pvObject_ )
```

Free Memory.

#### **Parameters**

pv⇔	Pointer to previously allocated block of memory
Object_	

Definition at line 34 of file mark3c.cpp.

## 20.5.5.28 Free\_Message()

Definition at line 82 of file mark3c.cpp.

# 20.5.5.29 Free\_MessagePool()

Definition at line 106 of file mark3c.cpp.

# 20.5.5.30 Free\_MessageQueue()

Definition at line 94 of file mark3c.cpp.

```
20.5.5.31 Free_Mutex()
void Free_Mutex (
             Mutex_t handle )
Definition at line 58 of file mark3c.cpp.
20.5.5.32 Free_Notify()
void Free_Notify (
             Notify_t handle )
Definition at line 118 of file mark3c.cpp.
20.5.5.33 Free_Semaphore()
void Free_Semaphore (
             Semaphore_t handle )
Definition at line 46 of file mark3c.cpp.
20.5.5.34 Free_Thread()
void Free_Thread (
              Thread_t handle )
Definition at line 161 of file mark3c.cpp.
20.5.5.35 Free_Timer()
void Free_Timer (
              Timer_t handle )
Definition at line 172 of file mark3c.cpp.
20.5.5.36 Kernel_DebugPrint()
void Kernel_DebugPrint (
              const char * szString_ )
KernelDebug_DebugPrint.
See also
```

void DebugPrint(const char\* szString\_)

## **Parameters**

SZ⊷	String to print to debug interface
String_	

Definition at line 282 of file mark3c.cpp.

```
20.5.5.37 Kernel_GetTicks()
```

Kernel\_GetTicks.

See also

Kernel::GetTicks()

## Returns

Number of kernel ticks that have elapsed since boot

Definition at line 216 of file mark3c.cpp.

# 20.5.5.38 Kernel\_Init()

```
void Kernel_Init (
    void )
```

Kernel\_Init.

See also

void Kernel::Init()

Definition at line 180 of file mark3c.cpp.

```
20.5.5.39 Kernel_IsPanic()
bool Kernel_IsPanic (
              void )
Kernel_IsPanic.
See also
     bool Kernel::IsPanic()
Returns
     Whether or not the kernel is in a panic state
Definition at line 204 of file mark3c.cpp.
20.5.5.40 Kernel_IsStarted()
bool Kernel_IsStarted (
               void )
Kernel_IsStarted.
See also
     bool Kernel::IsStarted()
Returns
     Whether or not the kernel has started - true = running, false = not started
Definition at line 192 of file mark3c.cpp.
20.5.5.41 Kernel_Panic()
void Kernel_Panic (
             uint16_t u16Cause_ )
Kernel Panic.
See also
```

void Kernel::Panic(uint16\_t u16Cause\_)

## **Parameters**

u16 <b></b>	Reason for the kernel panic
Cause_	

Definition at line 210 of file mark3c.cpp.

## 20.5.5.42 Kernel\_SetDebugPrintFunction()

 $Kernel\_SetDebugPrintFunction.$ 

See also

void Kernel::SetDebugPrintFunction()

#### **Parameters**

pfPrint⇔	Function to use to print debug information from the kernel
Function_	

Definition at line 276 of file mark3c.cpp.

## 20.5.5.43 Kernel\_SetPanic()

Kernel\_SetPanic.

See also

void Kernel::SetPanic(PanicFunc\_t pfPanic\_)

# **Parameters**

pf⇔	Panic function pointer
Panic⊷	

Definition at line 198 of file mark3c.cpp.

```
20.5.5.44 Kernel_Start()
void Kernel_Start (
             void )
Kernel_Start.
See also
     void Kernel::Start()
Definition at line 186 of file mark3c.cpp.
20.5.5.45 Mailbox_GetFreeSlots()
uint16_t Mailbox_GetFreeSlots (
             Mailbox_t handle )
Mailbox_GetFreeSlots.
See also
     uint16_t Mailbox::GetFreeSlots()
Parameters
 handle
           Handle of the mailbox object
Returns
     Number of free slots in the mailbox
Definition at line 832 of file mark3c.cpp.
20.5.5.46 Mailbox_Init()
void Mailbox_Init (
              Mailbox_t handle,
              void * pvBuffer_,
              uint16_t u16BufferSize_,
              uint16_t u16ElementSize_ )
Mailbox_Init.
```

void Mailbox::Init(void \*pvBuffer\_, uint16\_t u16BufferSize\_, uint16\_t u16ElementSize\_)

See also

## **Parameters**

handle	Handle of the mailbox object
pvBuffer_	Pointer to the static buffer to use for the mailbox
u16BufferSize_	Size of the mailbox buffer, in bytes
u16Element⊷ Size	Size of each envelope, in bytes

Definition at line 769 of file mark3c.cpp.

# 20.5.5.47 Mailbox\_IsEmpty()

Mailbox\_IsEmpty.

See also

bool Mailbox::IsEmpty()

## **Parameters**

handle	Handle of the mailbox object
--------	------------------------------

# Returns

true if the mailbox is empty, false otherwise

Definition at line 846 of file mark3c.cpp.

# 20.5.5.48 Mailbox\_lsFull()

Mailbox\_IsFull.

See also

bool Mailbox::IsFull()

## **Parameters**

handle	Handle of the mailbox object
--------	------------------------------

## Returns

true if the mailbox is full, false otherwise

Definition at line 839 of file mark3c.cpp.

### 20.5.5.49 Mailbox\_Receive()

Mailbox\_Receive.

#### See also

void Mailbox::Receive(void \*pvData\_)

## **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
Data_	

Definition at line 804 of file mark3c.cpp.

#### 20.5.5.50 Mailbox\_ReceiveTail()

Mailbox\_ReceiveTail.

## See also

void Mailbox::ReceiveTail(void \*pvData\_)

# **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
GenDelateed by Doxygen	

Definition at line 811 of file mark3c.cpp.

## 20.5.5.51 Mailbox\_Send()

Mailbox\_Send.

## See also

bool Mailbox::Send(void \*pvData\_)

#### **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to the data object to send to the mailbox.
Data_	

## Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 776 of file mark3c.cpp.

# 20.5.5.52 Mailbox\_SendTail()

 $Mailbox\_SendTail.$ 

#### See also

bool Mailbox::SendTail(void \*pvData\_)

# **Parameters**

handle	Handle of the mailbox object
pv⊷	Pointer to the data object to send to the mailbox.
Data_	

#### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 783 of file mark3c.cpp.

## 20.5.5.53 Mailbox\_TimedReceive()

 $Mailbox\_TimedReceive.$ 

#### See also

bool Mailbox::Receive(void \*pvData\_, uint32\_t u32TimeoutMS\_)

#### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM⊷ S	Maximum time to wait for delivery.

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 818 of file mark3c.cpp.

## 20.5.5.54 Mailbox\_TimedReceiveTail()

Mailbox\_TimedReceiveTail.

## See also

bool Mailbox::ReceiveTail(void \*pvData\_, uint32\_t u32TimeoutMS\_)

#### **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
u32TimeoutM↔ S_	Maximum time to wait for delivery.

#### Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 825 of file mark3c.cpp.

## 20.5.5.55 Mailbox\_TimedSend()

Mailbox\_TimedSend.

# See also

bool Mailbox::Send(void \*pvData\_, uint32\_t u32TimeoutMS\_)

## **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM⊷ S_	Maximum time to wait for a free transmit slot

## Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 790 of file mark3c.cpp.

# 20.5.5.56 Mailbox\_TimedSendTail()

Mailbox\_TimedSendTail.

## See also

bool Mailbox::Send(void \*pvData\_, uint32\_t u32TimeoutMS\_)

## **Parameters**

handle	Handle of the mailbox object
pvData_	Pointer to the data object to send to the mailbox.
u32TimeoutM⊷	Maximum time to wait for a free transmit slot
S_	

#### Returns

true - envelope was delivered, false - mailbox is full.

Definition at line 797 of file mark3c.cpp.

# 20.5.5.57 Message\_GetCode()

Message\_GetCode.

#### See also

uint16\_t Message::GetCode()

#### **Parameters**

	handle	Handle of the message object
--	--------	------------------------------

## Returns

user code set in the object

Definition at line 704 of file mark3c.cpp.

## 20.5.5.58 Message\_GetData()

 $Message\_GetData.$ 

## See also

void\* Message::GetData()

## **Parameters**

handle	Handle of the message object
--------	------------------------------

## Returns

Pointer to the data set in the message object

Definition at line 690 of file mark3c.cpp.

## 20.5.5.59 Message\_Init()

Message\_Init.

See also

void Message::Init()

#### **Parameters**

	handle	Handle of the message object
--	--------	------------------------------

Definition at line 676 of file mark3c.cpp.

## 20.5.5.60 Message\_SetCode()

 $Message\_SetCode.$ 

See also

void Message::SetCode(uint16\_t u16Code\_)

## **Parameters**

handle	Handle of the message object
<i>u</i> 16⇔	Data code to set in the object
Code	

Definition at line 697 of file mark3c.cpp.

## 20.5.5.61 Message\_SetData()

Message\_SetData.

#### See also

void Message::SetData(void \*pvData\_)

#### **Parameters**

handle	Handle of the message object
pv⊷	Pointer to the data object to send in the message
Data_	

Definition at line 683 of file mark3c.cpp.

## 20.5.5.62 MessagePool\_Init()

MessagePool\_Init.

## See also

void MessagePool::Init()

### **Parameters**

handle	Handle of the message pool object
--------	-----------------------------------

Definition at line 725 of file mark3c.cpp.

# 20.5.5.63 MessagePool\_Pop()

MessagePool\_Pop.

See also

Message\* MessagePool::Pop()

#### **Parameters**

handle	Handle of the message pool object
--------	-----------------------------------

#### Returns

Handle to a Message object, or nullptr on allocation error

Definition at line 739 of file mark3c.cpp.

## 20.5.5.64 MessagePool\_Push()

MessagePool\_Push.

See also

void MessagePool::Push(Message\* pclMessage\_)

#### **Parameters**

handle	Handle of the message pool object
msg	Message object to return back to the pool

Definition at line 732 of file mark3c.cpp.

# 20.5.5.65 MessageQueue\_GetCount()

 $Message Queue\_Get Count.$ 

See also

uint16\_t MessageQueue::GetCount()

#### Returns

Count of pending messages in the queue.

Definition at line 760 of file mark3c.cpp.

## 20.5.5.66 MessageQueue\_Init()

MessageQueue Init.

See also

void MessageQueue::Init()

## **Parameters**

handle Handle to the m	essage queue to initialize
------------------------	----------------------------

Definition at line 711 of file mark3c.cpp.

## 20.5.5.67 MessageQueue\_Receive()

MessageQueue\_Receive.

See also

Message\_t MessageQueue::Receive()

## **Parameters**

handle Handle of the message queue object
---

# Returns

Pointer to a message object at the head of the queue

Definition at line 718 of file mark3c.cpp.

## 20.5.5.68 MessageQueue\_Send()

MessageQueue\_Send.

## See also

void MessageQueue::Send(Message \*pclMessage\_)

#### **Parameters**

handle	Handle of the message queue object
h⇔	Handle to the message to send to the given queue
Message←	

Definition at line 753 of file mark3c.cpp.

# 20.5.5.69 MessageQueue\_TimedReceive()

MessageQueue\_TimedReceive.

## See also

Message\_t MessageQueue::TimedReceive(uint32\_t u32TimeWaitMS\_)

## **Parameters**

handle	Handle of the message queue object
u32TimeWaitM⊷ S_	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 nullptr on timeout.

Definition at line 746 of file mark3c.cpp.

```
20.5.5.70 Mutex_Claim()
```

Mutex\_Claim.

See also

void Mutex::Claim()

## **Parameters**

handle Handle of the m
------------------------

Definition at line 515 of file mark3c.cpp.

## 20.5.5.71 Mutex\_Init()

Mutex\_Init.

See also

void Mutex::Init()

# **Parameters**

handle	Handle of the mutex
Hallule	I landle of the mutex

Definition at line 508 of file mark3c.cpp.

# 20.5.5.72 Mutex\_Release()

Mutex\_Release.

See also

void Mutex::Release()

#### **Parameters**

nande Hande of the mutex	handle	Handle of the mutex
--------------------------	--------	---------------------

Definition at line 522 of file mark3c.cpp.

# 20.5.5.73 Mutex\_TimedClaim()

 $Mutex\_TimedClaim.$ 

## See also

bool Mutex::Claim(uint32\_t u32WaitTimeMS\_)

## **Parameters**

handle	Handle of the mutex
u32WaitTimeM⊷	Time to wait before aborting
S_	

## Returns

true if mutex was claimed, false on timeout

Definition at line 529 of file mark3c.cpp.

# 20.5.5.74 Notify\_Init()

Notify\_Init.

## See also

void Notify::Init()

## **Parameters**

handle Handle of the notification object
--

Definition at line 584 of file mark3c.cpp.

## 20.5.5.75 Notify\_Signal()

Notify\_Signal.

See also

void Notify::Signal()

#### **Parameters**

handle	Handle of the notification object
--------	-----------------------------------

Definition at line 591 of file mark3c.cpp.

## 20.5.5.76 Notify\_TimedWait()

Notify\_TimedWait.

See also

bool Notify::Wait(uint32\_t u32WaitTimeMS\_, bool \*pbFlag\_)

# **Parameters**

handle	Handle of the notification object
u32WaitTimeM←	Maximum time to wait for notification in ms
S_	
pbFlag_	Flag to set to true on notification

# Returns

true on unblock, false on timeout

Definition at line 605 of file mark3c.cpp.

## 20.5.5.77 Notify\_Wait()

Notify\_Wait.

See also

void Notify::Wait(bool \*pbFlag\_)

#### **Parameters**

handle	Handle of the notification object
pb⇔	Flag to set to true on notification
Flag_	

Definition at line 598 of file mark3c.cpp.

# 20.5.5.78 ReaderWriterLock\_AcquireReader()

ReaderWriterLock\_AcquireReader.

See also

void ReaderWriterLock::AcquireReader()

## **Parameters**

handle	Handle of the reader-writer object
Hariule	riandle of the reader-writer object

Definition at line 897 of file mark3c.cpp.

# 20.5.5.79 ReaderWriterLock\_AcquireWriter()

 $ReaderWriterLock\_AcquireWriter.$ 

See also

void ReaderWriterLock::AcquireWriter()

#### **Parameters**

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 911 of file mark3c.cpp.

#### 20.5.5.80 ReaderWriterLock\_Init()

ReaderWriterLock\_Init.

See also

void ReaderWriterLock::Init()

#### **Parameters**

handle	Handle of the reader-writer object
--------	------------------------------------

Definition at line 890 of file mark3c.cpp.

# 20.5.5.81 ReaderWriterLock\_ReleaseReader()

ReaderWriterLock\_ReleaseReader.

See also

void ReaderWriterLock::ReleaseReader()

# **Parameters**

handle	Handle of the reader-writer object

Definition at line 904 of file mark3c.cpp.

## 20.5.5.82 ReaderWriterLock\_ReleaseWriter()

ReaderWriterLock\_ReleaseWriter.

See also

void ReaderWriterLock::ReleaseWriter()

#### **Parameters**

Definition at line 918 of file mark3c.cpp.

## 20.5.5.83 ReaderWriterLock\_TimedAcquireReader()

ReaderWriterLock\_TimedAcquireReader.

See also

bool ReaderWriterLock::AcquireReader(uint32\_t u32TimeoutMs\_)

#### **Parameters**

handle	Handle of the reader-writer object
u32Timeout <i>⊷</i>	Maximum time to wait for the reader lock before bailing
Ms_	

## Returns

true on success, false on timeout

Definition at line 932 of file mark3c.cpp.

## 20.5.5.84 ReaderWriterLock\_TimedAcquireWriter()

ReaderWriterLock\_TimedAcquireWriter.

## See also

bool ReaderWriterLock::AcquireWriter(uint32\_t u32TimeoutMs\_)

## **Parameters**

handle	Handle of the reader-writer object
u32Timeout⇔	Maximum time to wait for the writer lock before bailing
Ms_	

#### Returns

true on success, false on timeout

Definition at line 925 of file mark3c.cpp.

# 20.5.5.85 Scheduler\_Enable()

```
void Scheduler_Enable (
          bool bEnable_ )
```

Scheduler\_Enable.

# See also

void Scheduler::SetScheduler(bool bEnable\_)

#### **Parameters**

b⇔	true to enable, false to disable the scheduler
Enable←	

Definition at line 289 of file mark3c.cpp.

#### 20.5.5.86 Scheduler\_GetCurrentThread()

 $Scheduler\_GetCurrentThread.$ 

## See also

Thread\* Scheduler::GetCurrentThread()

#### Returns

Handle of the currently-running thread

Definition at line 301 of file mark3c.cpp.

```
20.5.5.87 Scheduler_IsEnabled()
```

```
bool Scheduler_IsEnabled ( \mbox{void} \ \ \mbox{)}
```

Scheduler\_IsEnabled.

See also

bool Scheduler::IsEnabled()

## Returns

true - scheduler enabled, false - disabled

Definition at line 295 of file mark3c.cpp.

# 20.5.5.88 Semaphore\_Init()

Semaphore\_Init.

## See also

void Semaphore::Init(uint16\_t u16InitVal\_, uint16\_t u16MaxVal\_)

## **Parameters**

handle	Handle of the semaphore
u16InitVal↔	Initial value of the semaphore
_	
u16Max⊷	Maximum value that can be held for a semaphore
Val	

Definition at line 478 of file mark3c.cpp.

```
20.5.5.89 Semaphore_Pend()
```

Semaphore\_Pend.

See also

void Semaphore::Pend()

#### **Parameters**

handle	Handle of the semaphore
--------	-------------------------

Definition at line 492 of file mark3c.cpp.

# 20.5.5.90 Semaphore\_Post()

Semaphore\_Post.

See also

void Semaphore::Post()

# **Parameters**

handle	Handle of the semaphore

Definition at line 485 of file mark3c.cpp.

# 20.5.5.91 Semaphore\_TimedPend()

 $Semaphore\_TimedPend.$ 

See also

bool Semaphore::Pend(uint32\_t u32WaitTimeMS\_)

## **Parameters**

handle	Handle of the semaphore
u32WaitTimeM⊷	Time in ms to wait
S_	

#### Returns

true if semaphore was acquired, false on timeout

Definition at line 499 of file mark3c.cpp.

```
20.5.5.92 Thread_CoopYield()
```

 $Thread\_CoopYield.$ 

See also

void Thread::CoopYield()

Definition at line 415 of file mark3c.cpp.

## 20.5.5.93 Thread\_Exit()

Thread\_Exit.

See also

void Thread::Exit()

## **Parameters**

handle	Handle of the thread
Handle	nande of the thread

Definition at line 381 of file mark3c.cpp.

```
20.5.5.94 Thread_GetCurPriority()
```

Thread\_GetCurPriority.

See also

```
PORT_PRIO_TYPE Thread::GetCurPriority()
```

**Parameters** 

```
handle Handle of the thread
```

## Returns

Current priority of the thread considering priority inheritence

Definition at line 354 of file mark3c.cpp.

```
20.5.5.95 Thread_GetID()
```

Thread\_GetID.

See also

```
uint8_t Thread::GetID()
```

Parameters

```
handle Handle of the thread
```

Returns

Return ID assigned to the thread

Definition at line 426 of file mark3c.cpp.

## 20.5.5.96 Thread\_GetPriority()

Thread\_GetPriority.

## See also

```
PORT_PRIO_TYPE Thread::GetPriority()
```

## **Parameters**

```
handle Handle of the thread
```

## Returns

Current priority of the thread not considering priority inheritence

Definition at line 348 of file mark3c.cpp.

# 20.5.5.97 Thread\_GetState()

Thread\_GetState.

See also

ThreadState Thread::GetState()

## **Parameters**

```
handle Handle of the thread
```

## Returns

The thread's current execution state

Definition at line 438 of file mark3c.cpp.

#### 20.5.5.98 Thread\_Init()

Thread\_Init.

## See also

```
void Thread::Init(K_WORD *pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, Thread← Entry_t pfEntryPoint_, void *pvArg_)
```

#### **Parameters**

handle	Handle of the thread to initialize
pwStack_	Pointer to the stack to use for the thread
u16Stack⊷	Size of the stack (in bytes)
Size_	
uXPriority_	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 310 of file mark3c.cpp.

#### 20.5.5.99 Thread\_SetID()

Thread\_SetID.

#### See also

void Thread::SetID(uint8\_t u8ID\_)

# **Parameters**

handle	Handle of the thread
u8l⊷	ID To assign to the thread
D_	

Definition at line 420 of file mark3c.cpp.

# 20.5.5.100 Thread\_SetPriority()

Thread\_SetPriority.

## See also

void Thread::SetPriority(PORT\_PRIO\_TYPE uXPriority\_)

## **Parameters**

handle	Handle of the thread
uX⇔	New priority level
Priority_	

Definition at line 374 of file mark3c.cpp.

```
20.5.5.101 Thread_Sleep()
```

Thread\_Sleep.

See also

void Thread::Sleep(uint32\_t u32TimeMs\_)

#### **Parameters**

u32Time←	Time in ms to block the thread for
Ms_	

Definition at line 388 of file mark3c.cpp.

# 20.5.5.102 Thread\_Start()

Thread\_Start.

See also

void Thread::Start()

# **Parameters**

handle	Handle of the thread to start

Definition at line 322 of file mark3c.cpp.

```
20.5.5.103 Thread_Stop()
```

Thread\_Stop.

See also

void Thread::Stop()

**Parameters** 

handle Handle of the thread to stop

Definition at line 329 of file mark3c.cpp.

# 20.5.5.104 Thread\_Yield()

```
void Thread_Yield (
     void )
```

Thread\_Yield.

See also

void Thread::Yield()

Definition at line 410 of file mark3c.cpp.

# 20.5.5.105 Timer\_Init()

Timer\_Init.

See also

void Timer::Init()

# **Parameters**

handle Handle of the timer

Definition at line 448 of file mark3c.cpp.

# 

 $Timer\_Restart.$ 

See also

void Timer::Start()

# **Parameters**

handle	Handle of the timer to restart.
--------	---------------------------------

Definition at line 469 of file mark3c.cpp.

# 20.5.5.107 Timer\_Start()

Timer\_Start.

See also

void Timer::Start(bool bRepeat\_, uint32\_t u32IntervalMs\_, uint32\_t u32ToleranceMs\_, TimerCallbackC\_t pf  $\leftarrow$  Callback\_, void \*pvData\_)

### **Parameters**

handle	Handle of the timer
bRepeat_	Restart the timer continuously on expiry
u32Interval←	Time in ms to expiry
Ms_	
pfCallback_	Callback to run on timer expiry
pvData_	Data to pass to the callback on expiry

Definition at line 455 of file mark3c.cpp.

20.6 mark3c.h 419

#### 20.5.5.108 Timer\_Stop()

Timer\_Stop.

See also

void Timer::Stop()

#### **Parameters**

handle Handle of the timer

Definition at line 462 of file mark3c.cpp.

# 20.6 mark3c.h

```
00001 /*-----
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00021 #pragma once
00022
00023 #include "mark3cfg.h"
00024 #include "fake_types.h"
00025
00026 #include <stdint.h>
00027 #include <stdbool.h>
00028
00029 #if defined(__cplusplus)
00030 extern "C" {
00031 #endif
00032
00033 //
00034 // Define a series of handle types to be used in place of the underlying classes
00035 // of Mark3.
00036 #if KERNEL_EVENT_FLAGS
00037 typedef void* EventFlag_t;
                                        // #if KERNEL_EVENT_FLAGS
00038 #endif
00039 typedef void* Mailbox_t;
00040 typedef void* Message_t;
00041 typedef void* MessagePool_t;
00042 typedef void* MessageQueue_t;
00043 typedef void* Mutex_t;
00044 typedef void* Notify_t;
00045 typedef void* Semaphore_t;
00046 typedef void* Thread_t;
00047 typedef void* Timer_t;
00048 typedef void* ConditionVariable_t;
00049 typedef void* ReaderWriterLock_t;
00050
00051 //----
00052 // Function pointer types used by Kernel APIs
00053 #if KERNEL_THREAD_CREATE_CALLOUT
```

```
00054 typedef void (*thread_create_callout_t)(Thread_t hThread_);
00055 #endif // #if KERNEL_THREAD_CREATE_CALLOUT
00057 typedef void (*thread_exit_callout_t)(Thread_t hThread_);
00058 #endif // #if KERNEL_THREAD_EXIT_CALLOUT 00059 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00060 typedef void (*thread_context_callout_t)(Thread_t hThread_);
00061 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOT
00062 typedef void (*kernel_debug_print_t)(const char* szString_);
00063
00064 //----
00065 // Use the sizes of the structs in fake_types.h to generate opaque object-blobs
00066 // that get instantiated as kernel objects (from the C++ code) later.
00067 #define THREAD_SIZE (sizeof(Fake_Thread))
00068 #define TIMER_SIZE (sizeof(Fake_Timer))
00069 #define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
00070 #define MUTEX_SIZE (sizeof(Fake_Mutex))
00071 #define MESSAGE_SIZE (sizeof(Fake_Message))
00072 #define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
00073 #define MAILBOX_SIZE (sizeof(Fake_Mailbox))
00074 #define NOTIFY_SIZE (sizeof(Fake_Notify))
00075 #if KERNEL_EVENT_FLAGS
00076 #define EVENTFLAG_SIZE (sizeof(Fake_EventFlag))
00077 #endif // #if KERNEL_EVENT_FLAGS
00078 #define MESSAGEPOOL_SIZE (sizeof(Fake_MessagePool))
00079 #define CONDITIONVARIABLE_SIZE (sizeof(Fake_ConditionVariable))
00080 #define READERWRITERLOCK_SIZE (sizeof(Fake_ReaderWriterLock))
00081
00082 #if KERNEL_EVENT_FLAGS
00083 //----
00088 typedef enum {
00089
           EVENT_FLAG_ALL_SET,
00090
           EVENT_FLAG_ANY_SET,
00091
           EVENT_FLAG_ALL_CLEAR,
00092
          EVENT_FlAG_ANY_CLEAR,
          EVENT FLAG_PENDING_UNBLOCK
00093
00094 } event_flag_operation_t;
00095 #endif // #if KERNEL_EVENT_FLAGS
00096
00097 //---
00101 typedef enum {
           THREAD_STATE_EXIT = 0,
THREAD_STATE_READY,
00102
00103
00104
           THREAD_STATE_BLOCKED,
           THREAD_STATE_STOP,
00105
00106
           THREAD_STATE_INVALID
00107 } thread_state_t;
00108
00109 //----
00110 // Macros for declaring opaque buffers of an appropriate size for the given
00111 // kernel objects
00112 #define TOKEN_1(x, y) x##y
00113 #define TOKEN_2(x, y) TOKEN_1(x, y)
00114
00115 \// Ensure that opaque buffers are sized to the nearest word - which is
00116 // a platform-dependent value.
00117 #define WORD_ROUND(x) (((x) + (sizeof(K_WORD) - 1)) / sizeof(K_WORD))
00118
00119 #define DECLARE_THREAD(name)
00120
           K WORD
                     TOKEN_2(__thread_, name)[WORD_ROUND(THREAD_SIZE)];
00121
           Thread_t name = (Thread_t)TOKEN_2(__thread_, name);
00123 #define DECLARE_TIMER(name)
                   TOKEN_2(__timer_, name)[WORD_ROUND(TIMER_SIZE)];
00124
           K WORD
00125
           Timer_t name = (Timer_t)TOKEN_2(__timer_, name);
00126
00127 #define DECLARE_SEMAPHORE(name)
00128
           K WORD
                        TOKEN_2(__semaphore_, name)[WORD_ROUND(SEMAPHORE_SIZE)];
00129
           Semaphore t name = (Semaphore t) TOKEN 2 ( semaphore , name);
00130
00131 #define DECLARE_MUTEX(name)
00132
           K_WORD TOKEN_2(__mutex_, name)[WORD_ROUND(MUTEX_SIZE)];
00133
           Mutex t name = (Mutex t) TOKEN 2 ( mutex , name);
00134
00135 #define DECLARE MESSAGE (name)
00136
           K WORD
                      TOKEN_2(__message_, name)[WORD_ROUND(MESSAGE_SIZE)];
00137
           Message t name = (Message t) TOKEN 2( message , name);
```

20.6 mark3c.h 421

```
00138
00139 #define DECLARE_MESSAGEPOOL (name)
00140
          K WORD
                        TOKEN_2(__messagepool_, name)[WORD_ROUND(MESSAGEPOOL_SIZE)];
00141
          MessagePool t name = (MessagePool t)TOKEN 2( messagepool , name);
00142
00143 #define DECLARE_MESSAGEQUEUE(name)
00144
          K WORD
                         TOKEN_2(__messagequeue_, name)[WORD_ROUND(MESSAGEQUEUE_SIZE)];
00145
          MessageQueue_t name = (MessageQueue_t)TOKEN_2(__messagequeue , name);
00146
00147 #define DECLARE_MAILBOX(name)
00148
          K WORD
                    TOKEN_2(__mailbox_, name)[WORD_ROUND(MAILBOX_SIZE)];
00149
          Mailbox_t name = (Mailbox_t)TOKEN_2(__mailbox_, name);
00150
00151 #define DECLARE_NOTIFY(name)
00152
          K_WORD
                   TOKEN_2(__notify_, name)[WORD_ROUND(NOTIFY_SIZE)];
00153
          Notify_t name = (Notify_t)TOKEN_2(__notify_, name);
00154
00155 #if KERNEL_EVENT_FLAGS
00156 #define DECLARE_EVENTFLAG(name)
00157
          K WORD
                      TOKEN_2(__eventflag_, name)[WORD_ROUND(EVENTFLAG_SIZE)];
00158
          EventFlag t name = (EventFlag t) TOKEN 2( eventflag , name);
00159 #endif // #if KERNEL_EVENT_FLAGS
00160
00161 #define DECLARE_CONDITIONVARIABLE(name)
00162
          K WORD
                               TOKEN_2 (__condvar_, name) [WORD_ROUND (EVENTFLAG_SIZE)];
00163
          ConditionVariable_t name = (ConditionVariable_t) TOKEN_2 (__condvar_, name);
00164
00165 #define DECLARE_READERWRITERLOCK(name)
00166
          K WORD
                              TOKEN_2(__readerwriterlock_, name)[WORD_ROUND(EVENTFLAG_SIZE)];
00167
          ReaderWriterLock_t name = (ReaderWriterLock_t)TOKEN_2(__readerwriterlock_, name);
00168
00169 //-
00170 // Allocate-once Memory managment APIs
00177 void* Alloc_Memory(size_t eSize_);
00178
00183 void Free Memory (void* pvObject );
00184
00190 Semaphore_t Alloc_Semaphore(void);
00191 void
                  Free_Semaphore(Semaphore_t handle);
00192
00198 Mutex_t Alloc_Mutex(void);
00199 void
             Free_Mutex(Mutex_t handle);
00200
00201 #if KERNEL_EVENT_FLAGS
00202
00207 EventFlag_t Alloc_EventFlag(void);
00208 void
                  Free_EventFlag(EventFlag_t handle);
00209 #endif // #if KERNEL_EVENT_FLAGS
00210
00216 Message_t Alloc_Message(void);
00217 void
               Free_Message (Message_t handle);
00218
00224 MessageQueue_t Alloc_MessageQueue(void);
00225 void
                     Free_MessageQueue (MessageQueue_t handle);
00226
00227 MessagePool_t Alloc_MessagePool(void);
00228 void
                    Free_MessagePool(MessagePool_t handle);
00229
00235 Notify_t Alloc_Notify(void);
00236 void
              Free_Notify(Notify_t handle);
00237
00243 Mailbox_t Alloc_Mailbox(void);
00244 void
               Free_Mailbox(Mailbox_t handle);
00245
00251 Thread_t Alloc_Thread(void);
00252 void
              Free_Thread(Thread_t handle);
00253
00259 Timer_t Alloc_Timer(void);
00260 void
             Free_Timer(Timer_t handle);
00261
00262 /
00263 // Kernel APIs
00268 void Kernel Init(void);
```

```
00273 void Kernel_Start(void);
00280 bool Kernel_IsStarted(void);
00281
00282 typedef void (*panic_func_t) (uint16_t u16PanicCode_);
00288 void Kernel_SetPanic(panic_func_t pfPanic_);
00294 bool Kernel_IsPanic(void);
00300 void Kernel_Panic(uint16_t u16Cause_);
00301
00302 #if KERNEL_THREAD_CREATE_CALLOUT
00303
00308 void Kernel SetThreadCreateCallout(thread_create_callout_t pfCreate_);
00309 #endif // #if KERNEL_THREAD_CREATE_CALLOUT
00310
00311 #if KERNEL_THREAD_EXIT_CALLOUT
00312
00317 void Kernel_SetThreadExitCallout(thread_exit_callout_t pfExit_);
00318 #endif // #if KERNEL_THREAD_EXIT_CALLOUT
00319
00320 #if KERNEL CONTEXT SWITCH CALLOUT
00326 void Kernel_SetThreadContextSwitchCallout(thread_context_callout_t pfContext_);
00327 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00328
00329 #if KERNEL THREAD CREATE CALLOUT
00330
00335 thread_create_callout_t Kernel_GetThreadCreateCallout(void);
00336 #endif // #if KERNEL_THREAD_CREATE_CALLOUT
00337
00338 #if KERNEL THREAD EXIT CALLOUT
00339
00344 thread_exit_callout_t Kernel_GetThreadExitCallout(void); 00345 #endif // #if KERNEL_THREAD_EXIT_CALLOUT
00346
00347 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00348
00353 thread_context_callout_t Kernel_GetThreadContextSwitchCallout(void);
00354 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00356 #if KERNEL STACK CHECK
00357
00363 void Kernel_SetStackGuardThreshold(uint16_t u16Threshold_);
00364
00370 uint16_t Kernel_GetStackGuardThreshold(void);
00371 #endif // #if KERNEL_STACK_CHECK
00372
00378 uint32_t Kernel_GetTicks(void);
00379
00380 //----
00381 // Scheduler APIs
00387 void Scheduler_Enable(bool bEnable_);
00393 bool Scheduler_IsEnabled(void);
00399 Thread_t Scheduler_GetCurrentThread(void);
00400
00401 typedef void (*thread_entry_func_t)(void* pvArg_);
00402 //----
00403 // Thread APIs
00417 void Thread_Init(Thread_t
00418
00419
                        uint16_t
                                              u16StackSize_,
                        PORT_PRIO_TYPE
                                              uXPriority_,
00420
00421
                        thread_entry_func_t pfEntryPoint_,
00422
                        void*
                                              pvArg_);
00428 void Thread_Start(Thread_t handle);
00434 void Thread_Stop(Thread_t handle);
00435
00436 #if KERNEL_NAMED_THREADS
00437
00443 void Thread_SetName(Thread_t handle, const char* szName_);
00450 const char* Thread_GetName(Thread_t handle);
00451 #endif // #if KERNEL_NAMED_THREADS
00452
00459 PORT_PRIO_TYPE Thread_GetPriority(Thread_t handle);
00466 PORT_PRIO_TYPE Thread_GetCurPriority(Thread_t handle);
00467
00468 #if KERNEL ROUND ROBIN
00469
00475 void Thread_SetQuantum(Thread_t handle, uint16_t u16Quantum_);
00482 uint16_t Thread_GetQuantum(Thread_t handle); 00483 #endif // #if KERNEL_ROUND_ROBIN
00484
00491 void Thread SetPriority (Thread t handle, PORT PRIO TYPE uXPriority );
00497 void Thread_Exit(Thread_t handle);
00503 void Thread_Sleep(uint32_t u32TimeMs_);
00504
00505 #if KERNEL_EXTENDED_CONTEXT
00506
00512 void* Thread GetExtendedContext(Thread t handle);
```

20.6 mark3c.h 423

```
00520 void Thread_SetExtendedContext(Thread_t handle, void* pvData_);
00521 #endif // #if KERNEL_EXTENDED_CONTEXT
00522
00527 void Thread Yield (void);
00528
00533 void Thread_CoopYield(void);
00534
00541 void Thread_SetID(Thread_t handle, uint8_t u8ID_);
00548 uint8_t Thread_GetID(Thread_t handle);
00549
00550 #if KERNEL STACK CHECK
00551
00557 uint16_t Thread_GetStackSlack(Thread_t handle);
00558 #endif // #if KERNEL_STACK_CHECK
00559
00566 thread_state_t Thread_GetState(Thread_t handle);
00567
00569 // Timer APIs
00570 typedef void (*timer_callback_t)(Thread_t hOwner_, void* pvData_);
00576 void Timer_Init(Timer_t handle);
00587 void Timer_Start(Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_,
     timer_callback_t pfCallback_, void* pvData_);
00588
00594 void Timer_Restart(Timer_t handle);
00595
00601 void Timer_Stop(Timer_t handle);
00602
00603 //----
00604 // Semaphore APIs
00612 void Semaphore_Init (Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_);
00618 void Semaphore_Post(Semaphore_t handle);
00624 void Semaphore_Pend(Semaphore_t handle);
00632 bool Semaphore_TimedPend(Semaphore_t handle, uint32_t u32WaitTimeMS_);
00633
00634 //-
00635 // Mutex APIs
00641 void Mutex_Init(Mutex_t handle);
00647 void Mutex_Claim(Mutex_t handle);
00653 void Mutex_Release(Mutex_t handle);
00661 bool Mutex_TimedClaim(Mutex_t handle, uint32_t u32WaitTimeMS_);
00662
00663 #if KERNEL_EVENT_FLAGS
00665 // EventFlag APIs
00671 void EventFlag_Init(EventFlag_t handle);
00680 uint16_t EventFlag_Wait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_);
00690 uint16_t EventFlag_TimedWait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_, uint32_t
      u32TimeMS ):
00697 void EventFlag_Set(EventFlag_t handle, uint16_t u16Mask_);
00704 void EventFlag_Clear(EventFlag_t handle, uint16_t u16Mask_);
00711 uint16_t EventFlag_GetMask(EventFlag_t handle);
00712 #endif // #if KERNEL_EVENT_FLAGS
00713
00714 //-
00715 // Notification APIs
00721 void Notify_Init(Notify_t handle);
00727 void Notify_Signal(Notify_t handle);
00734 void Notify_Wait(Notify_t handle, bool* pbFlag_);
00743 bool Notify_TimedWait(Notify_t handle, uint32_t u32WaitTimeMS_, bool* pbFlag_);
00744
00746 // Atomic Functions
00754 uint8_t Atomic_Set8(uint8_t* pu8Source_, uint8_t u8Val_);
00762 uint16_t Atomic_Set16(uint16_t* pu16Source_, uint16_t u16Val_);
00770 uint32_t Atomic_Set32(uint32_t* pu32Source_, uint32_t u32Val_);
00778 uint8_t Atomic_Add8(uint8_t* pu8Source_, uint8_t u8Val_);
00786 uint16_t Atomic_Add16(uint16_t* pu16Source_, uint16_t u16Val_);
00794 uint32_t Atomic_Add32(uint32_t* pu32Source_, uint32_t u32Val_);
00802 uint8_t Atomic_Sub8(uint8_t* pu8Source_, uint8_t u8Val_);
00810 uint16_t Atomic_Sub16(uint16_t* pu16Source_, uint16_t u16Val_);
00818 uint32_t Atomic_Sub32(uint32_t* pu32Source_, uint32_t u32Val_);
00827 bool Atomic_TestAndSet(bool* pbLock);
00828
00829 //-
00830 // Message/Message Queue APIs
00836 void Message_Init(Message_t handle);
00843 void Message_SetData(Message_t handle, void* pvData_);
00850 void* Message_GetData(Message_t handle);
00857 void Message_SetCode(Message_t handle, uint16_t u16Code_);
00864 uint16_t Message_GetCode(Message_t handle);
00870 void MessageQueue_Init(MessageQueue_t handle);
00877 Message_t MessageQueue_Receive(MessageQueue_t handle);
00888 Message_t MessageQueue_TimedReceive (MessageQueue_t handle, uint32_t u32TimeWaitMS_
00889
```

```
00896 void MessageQueue_Send(MessageQueue_t handle, Message_t hMessage_);
00903 uint16_t MessageQueue_GetCount (MessageQueue_t handle);
00904
00910 void MessagePool Init (MessagePool t handle);
00911
00918 void MessagePool_Push(MessagePool_t handle, Message_t msg);
00919
00926 Message_t MessagePool_Pop(MessagePool_t handle);
00927
00928 //----
00929 // Mailbox APIs
00930
00939 void Mailbox_Init(Mailbox_t handle, void* pvBuffer_, uint16_t u16BufferSize_, uint16_t
     u16ElementSize_);
00940
00948 bool Mailbox Send(Mailbox t handle, void* pvData );
00949
00957 bool Mailbox_SendTail(Mailbox_t handle, void* pvData_);
00967 bool Mailbox_TimedSend(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
00968
00977 bool Mailbox_TimedSendTail(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
00978
00986 void Mailbox_Receive(Mailbox_t handle, void* pvData_);
00995 void Mailbox_ReceiveTail(Mailbox_t handle, void* pvData_);
00996
01006 bool Mailbox_TimedReceive (Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
01007
01017 bool Mailbox TimedReceiveTail (Mailbox t handle, void* pvData, uint32 t
      u32TimeoutMS_);
01018
01025 uint16_t Mailbox_GetFreeSlots(Mailbox_t handle);
01026
01033 bool Mailbox_IsFull(Mailbox_t handle);
01034
01041 bool Mailbox_IsEmpty(Mailbox_t handle);
01042
01043 //-
01044 // Condition Variables
01050 void ConditionVariable_Init(ConditionVariable_t handle);
01051
01058 void ConditionVariable_Wait(ConditionVariable_t handle, Mutex_t hMutex_);
01065 void ConditionVariable_Signal(ConditionVariable_t handle);
01066
01072 void ConditionVariable_Broadcast (ConditionVariable_t handle);
01073
01082 bool ConditionVariable_TimedWait(ConditionVariable_t handle, Mutex_t hMutex_,
     uint32_t u32WaitTimeMS_);
01083
01084 //--
01085 // Reader-writer locks
01091 void ReaderWriterLock_Init(ReaderWriterLock_t handle);
01092
01098 void ReaderWriterLock_AcquireReader(ReaderWriterLock_t handle);
01099
01105 void ReaderWriterLock_ReleaseReader(ReaderWriterLock_t handle);
01106
01112 void ReaderWriterLock AcquireWriter(ReaderWriterLock t handle);
01113
01119 void ReaderWriterLock_ReleaseWriter(ReaderWriterLock_t handle);
01128 bool ReaderWriterLock_TimedAcquireWriter(ReaderWriterLock_t handle,
      uint32_t u32TimeoutMs_);
01129
01137 bool ReaderWriterLock TimedAcquireReader(ReaderWriterLock t handle,
      uint32 t u32TimeoutMs );
01144 void Kernel_SetDebugPrintFunction(
      kernel_debug_print_t pfPrintFunction_);
01145
01151 void Kernel_DebugPrint(const char* szString_);
01152
01153 #if defined(__cplusplus)
01154
01155 #endif
```

# 20.7 /home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp File Reference

Implementation of memory, string, and conversion routines.

20.8 memutil.cpp 425

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

# **Namespaces**

• Mark3

### 20.7.1 Detailed Description

Implementation of memory, string, and conversion routines.

Definition in file memutil.cpp.

# 20.8 memutil.cpp

```
00001
00002
00003
00004
00005
00006 1
00007
00009
      --[Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00021 #include "kerneltypes.h'
00022 #include "mark3cfg.h"
00023 #include "memutil.h"
00024 #include "kerneldebug.h"
00025
00026 namespace Mark3
00027 {
00028 //---
00029 void MemUtil::DecimalToHex(uint8_t u8Data_, char* szText_)
00030 {
00031
          uint8_t u8Tmp = u8Data_;
00032
          uint8_t u8Max;
00033
00034
          KERNEL_ASSERT(szText_);
00035
00036
          if (u8Tmp >= 0x10) {
00037
              u8Max = 2;
          } else {
00038
00039
              u8Max = 1;
00040
          }
00041
00042
          u8Tmp
                          = u8Data_;
          szText_[u8Max] = 0;
while ((u8Max--) != 0u) {
00043
00044
              if ((u8Tmp & 0x0F) <= 9) {
    szText_[u8Max] = '0' + (u8Tmp & 0x0F);
00045
00046
00047
              } else {
00048
                 szText_[u8Max] = 'A' + ((u8Tmp & 0x0F) - 10);
00049
00050
              u8Tmp >>= 4;
          }
00051
00052 }
00053
00054 //--
00055 void MemUtil::DecimalToHex(uint16_t u16Data_, char* szText_)
00056 {
00057
          uint16_t u16Tmp
                               = u16Data_;
00058
          uint16_t u16Max
00059
          uint16_t u16Compare = 0x0010;
```

```
00060
00061
          KERNEL_ASSERT (szText_);
00062
00063
          while (u16Data_ > u16Compare && u16Max < 4) {</pre>
00064
              u16Max++;
00065
              u16Compare <<= 4;
00066
00067
00068
          u16Tmp
                           = u16Data_;
          szText_[u16Max] = 0;
while ((u16Max--) != 0u) {
00069
00070
             if ((u16Tmp & 0x0F) <= 9) {
    szText_[u16Max] = '0' + (u16Tmp & 0x0F);
00071
00072
               } else {
00073
00074
                   szText_[u16Max] = 'A' + ((u16Tmp & 0x0F) - 10);
00075
00076
               u16Tmp >>= 4;
00077
          }
00078 }
00079
00080 //---
00081 void MemUtil::DecimalToHex(uint32_t u32Data_, char* szText_)
00082 {
                               = u32Data_;
= 1;
          uint32_t u32Tmp
uint32_t u32Max
00083
00084
00085
          uint32_t u32Compare = 0 \times 0010;
00086
00087
          KERNEL_ASSERT (szText_);
00088
          while (u32Data_ > u32Compare && u32Max < 8) {
00089
00090
              u32Max++;
00091
              u32Compare <<= 4;
00092
00093
                           = u32Data_;
00094
          u32Tmp
          szText_[u32Max] = 0;
while ((u32Max--) != 0u) {
00095
00096
              if ((u32Tmp & 0xOF) <= 9) {
    szText_[u32Max] = '0' + (u32Tmp & 0xOF);
00098
00099
00100
                   szText_[u32Max] = 'A' + ((u32Tmp & 0x0F) - 10);
00101
              u32Tmp >>= 4;
00102
00103
          }
00104 }
00105
00106 //---
00107 void MemUtil::DecimalToHex(uint64_t u64Data_, char* szText_)
00108 {
00109
          uint64_t u64Tmp
                                = u64Data :
          uint64_t u64Max
00110
00111
          uint64_t u64Compare = 0x0010;
00112
00113
          KERNEL ASSERT (szText );
00114
00115
          while (u64Data_ > u64Compare && u64Max < 8) {</pre>
00116
             u64Max++;
00117
              u64Compare <<= 4;
00118
00119
00120
          1164Tmp
                           = u64Data ;
          szText_[u64Max] = 0;
00121
00122
          while ((u64Max--) != 0u) {
            if ((u64Tmp & 0x0F) <= 9) {
    szText_[u64Max] = '0' + (u64Tmp & 0x0F);
00123
00124
00125
               } else {
                   szText_[u64Max] = 'A' + ((u64Tmp & 0x0F) - 10);
00126
00127
00128
               u64Tmp >>= 4;
00129
          }
00130 }
00131
00132 //---
00133 void MemUtil::DecimalToString(uint8_t u8Data_, char* szText_)
00134 {
00135
          uint8_t u8Tmp = u8Data_;
00136
          uint8_t u8Max;
00137
          KERNEL_ASSERT(szText_);
00138
00139
00140
          // Find max index to print...
          if (u8Data_ >= 100) {
    u8Max = 3;
00141
00142
00143
          } else if (u8Data_ >= 10) {
00144
              u8Max = 2;
          } else {
00145
00146
              u8Max = 1;
```

20.8 memutil.cpp 427

```
00147
           }
00148
           szText_[u8Max] = 0;
while ((u8Max--) != 0u) {
    szText_[u8Max] = '0' + (u8Tmp % 10);
00149
00150
00151
00152
               u8Tmp /= 10;
00153
00154 }
00155
00156 //---
00157 void MemUtil::DecimalToString(uint16_t u16Data_, char* szText_)
00158 {
00159
           uint16_t u16Tmp
                              = u16Data_;
= 1;
00160
           uint16_t u16Max
00161
           uint16_t u16Compare = 10;
00162
           KERNEL ASSERT (szText ):
00163
00164
00165
           while (u16Data_ >= u16Compare && u16Max < 5) {</pre>
00166
               u16Compare *= 10;
00167
               u16Max++;
00168
           }
00169
           szText_[u16Max] = 0;
while ((u16Max--) != 0u) {
    szText_[u16Max] = '0' + (u16Tmp % 10);
00170
00171
00172
00173
               u16Tmp /= 10;
00174
           }
00175 }
00176
00177 //-
00178 void MemUtil::DecimalToString(uint32_t u32Data_, char* szText_)
00179 {
                               = u32Data_;
= 1;
00180
           uint32_t u32Tmp
           uint32_t u32Max = 1;
uint32_t u32Compare = 10;
00181
00182
00183
00184
           KERNEL_ASSERT(szText_);
00185
00186
           while (u32Data_ >= u32Compare && u32Max < 12) {</pre>
00187
               u32Compare *= 10;
               u32Max++;
00188
00189
           }
00190
00191
           szText_[u32Max] = 0;
           while ((u32Max--) != 0u) {
   szText_[u32Max] = '0' + (u32Tmp % 10);
00192
00193
00194
               u32Tmp /= 10;
00195
           }
00196 }
00197
00198 //---
00199 void MemUtil::DecimalToString(uint64_t u64Data_, char* szText_)
00200 {
           uint64_t u64Tmp = u64
uint64_t u64Max = 1;
uint64_t u64Compare = 10;
00201
                               = u64Data_;
00202
00203
00204
00205
           KERNEL_ASSERT(szText_);
00206
           while (u64Data_ >= u64Compare && u64Max < 12) {
    u64Compare *= 10;</pre>
00207
00208
00209
               u64Max++;
00210
00211
00212
           szText_[u64Max] = 0;
           while ((u64Max--) != 0u) {
    szText_[u64Max] = '0' + (u64Tmp % 10);
    u64Tmp /= 10;
00213
00214
00215
00216
           }
00217 }
00218
00219 //---
00220 bool MemUtil::StringToDecimal8(const char* szText_, uint8_t* pu8Out_)
00221 {
00222
           uint8_t u8Tmp = 0;
00223
          uint8_t u8Len = 0;
00224
           00225
00226
00227
00228
                         return false;
00229
00230
                    u8Len = i;
00231
                    break;
00232
               }
00233
           }
```

```
00234
           for (uint8_t i = 0; i < u8Len; i++) {
    if ((szText_[i] < '0') || (szText_[i] > '9')) {
00235
00236
00237
                   return false;
00238
00239
               u8Tmp *= 10;
00240
               u8Tmp += szText_[i] - '0';
00241
00242
           *pu80ut_ = u8Tmp;
00243
00244
           return true;
00245 }
00246
00247 //---
00248 bool MemUtil::StringToDecimal16(const char* szText_, uint16_t* pu16Out_)
00249 {
           uint16_t u16Tmp = 0;
00250
00251
           uint16_t u16Len = 0;
00253
           for (uint8_t i = 0; i < 6; i++) {</pre>
00254
               if (szText_[i] == 0) {
                    if (i == 0) {
00255
00256
                        return false;
00257
00258
                   u16Len = i;
00259
                   break;
00260
               }
00261
          }
00262
           for (uint8_t i = 0; i < u16Len; i++) {
    if ((szText_[i] < '0') || (szText_[i] > '9')) {
00263
00264
00265
                    return false;
00266
00267
               u16Tmp *= 10;
               u16Tmp += szText_[i] - '0';
00268
00269
00270
           *pu16Out_ = u16Tmp;
00271
00272
           return true;
00273 }
00274
00275 //---
00276 bool MemUtil::StringToDecimal32(const char* szText_, uint32_t* pu320ut_)
00277 {
00278
           uint32_t u32Tmp = 0;
00279
           uint32_t u32Len = 0;
00280
           for (uint8_t i = 0; i < 11; i++) {
   if (szText_[i] == 0) {
      if (i == 0) {</pre>
00281
00282
00283
00284
                        return false;
00285
00286
                    u32Len = i;
00287
                   break;
00288
               }
00289
          }
00290
00291
           for (uint8_t i = 0; i < u32Len; i++) {</pre>
           if ((szText_[i] < '0') || (szText_[i] > '9')) {
00292
00293
                    return false;
00294
               u32Tmp *= 10;
00295
00296
               u32Tmp += szText_[i] - '0';
00297
00298
           *pu32Out_ = u32Tmp;
00299
00300
           return true;
00301 }
00302
00303 //--
00304 bool MemUtil::StringToDecimal64(const char* szText_, uint64_t* pu640ut_)
00305 {
          uint64_t u64Tmp = 0;
uint64_t u64Len = 0;
00306
00307
00308
00309
           for (uint8_t i = 0; i < 21; i++) {</pre>
               if (szText_[i] == 0) {
    if (i == 0) {
00310
00311
00312
                        return false;
00313
00314
                   u64Len = i;
00315
                    break;
00316
              }
00317
           }
00318
           for (uint8_t i = 0; i < u64Len; i++) {
    if ((szText_[i] < '0') || (szText_[i] > '9')) {
00319
00320
```

20.8 memutil.cpp 429

```
return false;
00322
00323
              u64Tmp *= 10;
             u64Tmp += szText_[i] - '0';
00324
00325
00326
          *pu640ut_ = u64Tmp;
00327
00328
          return true;
00329 }
00330
00331 //----
00332 // Basic checksum routines
00333 uint8_t MemUtil::Checksum8(const void* pvSrc_, uint16_t u16Len_)
00334 {
00335
          uint8_t u8Ret = 0;
00336
          uint8_t* pcData = (uint8_t*)pvSrc_;
00337
00338
          KERNEL ASSERT (pvSrc );
00339
00340
          // 8-bit CRC, computed byte at a time
00341
          while ((u16Len_--) != 0u) { u8Ret += *pcData++; }
00342
          return u8Ret;
00343 }
00344
00345 //-
00346 uint16_t MemUtil::Checksum16(const void* pvSrc_, uint16_t u16Len_)
00347 {
00348
          uint16_t u16Ret = 0;
          uint8_t* pcData = (uint8_t*)pvSrc_;
00349
00350
00351
          KERNEL_ASSERT (pvSrc_);
00352
00353
          // 16-bit CRC, computed byte at a time
00354
          while ((u16Len_--) != 0u) { u16Ret += *pcData++; }
00355
          return u16Ret;
00356 }
00357
00358 //--
00359 // Basic string routines
00360 uint16_t MemUtil::StringLength(const char* szStr_)
00361 {
          uint8_t* pcData = (uint8_t*)szStr_;
uint16_t u16Len = 0;
00362
00363
00364
00365
          KERNEL_ASSERT(szStr_);
00366
00367
          while (*pcData++ != 0u) { u16Len++; }
00368
          return u16Len;
00369 }
00370
00371 //--
00372 bool MemUtil::CompareStrings(const char* szStr1_, const char* szStr2_)
00373 {
          char* szTmp1 = (char*)szStr1_;
char* szTmp2 = (char*)szStr2_;
00374
00375
00376
00377
          KERNEL_ASSERT (szStr1_);
00378
          KERNEL_ASSERT(szStr2_);
00379
00380
          while ((*szTmp1 != 0) && (*szTmp2 != 0)) {
          if (*szTmp1++ != *szTmp2++) {
00381
00382
                  return false;
00383
              }
00384
          }
00385
00386
          \ensuremath{//} Both terminate at the same length
00387
          return ((*szTmp1) == 0) && ((*szTmp2) == 0);
00388 }
00389
00390 //--
00391 bool MemUtil::CompareStrings(const char* szStr1_, const char* szStr2_, uint16_t
      u16Length_)
00392 {
00393
          char* szTmp1 = (char*)szStr1_;
00394
          char* szTmp2 = (char*)szStr2_;
00395
00396
          while (((*szTmp1 != 0) && (*szTmp2 != 0)) && (u16Length_ != 0u)) {
00397
           if (*szTmp1++ != *szTmp2++) {
00398
                  return false;
00399
00400
              u16Length_--;
00401
          }
00402
00403
          // Both terminate at the same length
00404
          return (((*szTmp1) == 0) && ((*szTmp2) == 0)) || (u16Length_ == 0u);
00405 }
00406
```

```
00408 void MemUtil::CopyMemory(void* pvDst_, const void* pvSrc_, uint16_t u16Len_)
00409 {
          char* szDst = (char*)pvDst_;
char* szSrc = (char*)pvSrc_;
00410
00411
00412
          KERNEL_ASSERT (pvDst_);
00413
          KERNEL_ASSERT (pvSrc_);
00414
00415
00416
          // Run through the strings verifying that each character matches
00417
          // and the lengths are the same.
          while ((u16Len_--) != 0u) { *szDst++ = *szSrc++; }
00418
00419 }
00420
00421 //--
00422 void MemUtil::CopyString(char* szDst_, const char* szSrc_) 00423 {
          char* szDst = (char*)szDst_;
char* szSrc = (char*)szSrc_;
00424
00426
00427
          KERNEL_ASSERT (szDst_);
00428
          KERNEL_ASSERT(szSrc_);
00429
00430
          // Run through the strings verifying that each character matches
00431
          // and the lengths are the same.
          while (*szSrc != 0) { *szDst++ = *szSrc++; }
00432
00433 }
00434
00435 //---
00436 int16_t MemUtil::StringSearch(const char* szBuffer_, const char* szPattern_)
00437 {
          char* szTmpPat = (char*)szPattern_;
int16_t i16Idx = 0;
00438
00439
00440
          int16_t i16Start;
00441
          KERNEL_ASSERT(szBuffer_);
          KERNEL_ASSERT(szPattern_);
00442
00443
          // Run through the big buffer looking for a match of the pattern
00445
          while (szBuffer_[i16Idx] != 0) {
           // Reload the pattern
00446
              i16Start = i16Idx;
szTmpPat = (char*)szPattern_;
00447
00448
              while ((*szTmpPat != 0) && (szBuffer_[i16Idx] != 0)) {
00449
                  if (*szTmpPat != szBuffer_[i16Idx]) {
00450
00451
                      break;
00452
                  }
00453
                   szTmpPat++;
00454
                  i16Idx++;
00455
               // Made it to the end of the pattern, it's a match.
00456
               if (*szTmpPat == '\0') {
00457
00458
                   return il6Start;
00459
00460
              i16Tdx++:
00461
          }
00462
          return -1;
00464 }
00465
00466 //---
00467 bool MemUtil::CompareMemory(const void* pvMem1_, const void* pvMem2_, uint16_t
     u16Len_)
00468 {
00469
          char* szTmp1 = (char*)pvMem1_;
00470
          char* szTmp2 = (char*)pvMem2_;
00471
00472
          KERNEL_ASSERT (pvMem1_);
          KERNEL_ASSERT (pvMem2_);
00473
00474
00475
          // Run through the strings verifying that each character matches
00476
          // and the lengths are the same.
          while ((u16Len_--) != 0u) {
   if (*szTmp1++ != *szTmp2++) {
00477
00478
00479
                   return false;
00480
              }
00481
00482
          return true;
00483 }
00484
00485 //---
00486 void MemUtil::SetMemory(void* pvDst_, uint8_t u8Val_, uint16_t u16Len_)
00487 {
00488
          char* szDst = (char*)pvDst_;
00489
00490
          KERNEL_ASSERT (pvDst_);
00491
00492
          while ((u16Len --) != 0u) { *szDst++ = u8Val ; }
```

```
00493 }
00494
00495 //-
00496 uint8_t MemUtil::Tokenize(const char* szBuffer_, Token_t* pastTokens_, uint8_t
      u8MaxTokens_)
00497 {
00498
           uint8_t u8CurrArg = 0;
00499
           uint8_t u8LastArg = 0;
00500
           uint8_t i
00501
00502
           bool bEscape = false;
00503
00504
           KERNEL_ASSERT(szBuffer_);
00505
           KERNEL_ASSERT (pastTokens_);
00506
00507
           while (szBuffer_[i] != 0) {
              //-- Handle unescaped quotes
if (szBuffer_[i] == '\"') {
   bEscape = !bEscape;
00508
00509
00510
00511
00512
                    continue;
00513
               }
00514
               //-- Handle all escaped chars - by ignoring them
if (szBuffer_[i] == '\\') {
00515
00516
00518
                    if (szBuffer_[i] != 0) {
00519
00520
00521
                    continue:
00522
              }
00523
00524
               //-- Process chars based on current escape characters
00525
00526
                  // Everything within the quote is treated as literal, but escaped chars are still treated the
same
00527
                   i++;
                   continue;
00529
00530
              //-- Non-escaped case
if (szBuffer_[i] != ' ') {
00531
00532
00533
               i++;
00534
                   continue;
00535
00536
            pastTokens_[u8CurrArg].pcToken = &(szBuffer_[u8LastArg]);
pastTokens_[u8CurrArg].u8Len = i - u8LastArg;
u8CurrArg++;
00537
00538
00539
               u8CurrArg++;
00540
               if (u8CurrArg >= u8MaxTokens_) {
00541
                    return u8MaxTokens_;
00542
00543
00544
               while (szBuffer_[i] == ' ') { i++; }
00545
00546
00547
               u8LastArg = i;
00548
00549
         if ((i != 0u) && (szBuffer_[i] == 0) && ((i - u8LastArg) != 0)) {
               pastTokens_[u8CurrArg].pcToken = &(szBuffer_[u8LastArg]);
pastTokens_[u8CurrArg].u8Len = i - u8LastArg;
00550
00551
00552
               u8CurrArg++;
00554
           return u8CurrArg;
00555 }
00556 } // namespace Mark3
```

# 20.9 /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h File Reference

Utility class containing memory, string, and conversion routines.

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

#### **Classes**

struct Mark3::Token t

Token descriptor struct format.

· class Mark3::MemUtil

String and Memory manipu32ation class.

# **Namespaces**

• Mark3

# 20.9.1 Detailed Description

Utility class containing memory, string, and conversion routines.

Definition in file memutil.h.

# 20.10 memutil.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00021 #pragma once
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h'
00025
00026 namespace Mark3
00027 {
00028 //----
00032 typedef struct {
          const char* pcToken;
00033
          uint8_t
00035 } Token_t;
00036
00037 //----
00046 class MemUtil
00047 {
00048 public:
00059
           static void DecimalToHex(uint8_t u8Data_, char* szText_);
00060
           static void DecimalToHex(uint16_t u16Data_, char* szText_);
00061
           static void DecimalToHex(uint32_t u32Data_, char* szText_);
00062
           static void DecimalToHex(uint64_t u64Data_, char* szText_);
00063
00064
00073
           static void DecimalToString(uint8_t u8Data_, char* szText_);
           static void DecimalToString(uint16_t u16Data_, char* szText_); static void DecimalToString(uint32_t u32Data_, char* szText_);
00074
00075
00076
           static void DecimalToString(uint64_t u64Data_, char* szText_);
00077
00078
00089
           static bool StringToDecimal8(const char* szText_, uint8_t* pu8Out_);
           static bool StringToDecimal16(const char* szText_, uint16_t* pu16Out_); static bool StringToDecimal32(const char* szText_, uint32_t* pu32Out_);
00090
00091
00092
           static bool StringToDecimal64(const char* szText_, uint64_t* pu640ut_);
00093
00094
00104
           static uint8_t Checksum8(const void* pvSrc_, uint16_t u16Len_);
```

```
00105
00116
          static uint16_t Checksum16(const void* pvSrc_, uint16_t u16Len_);
00117
00118
00128
          static uint16_t StringLength(const char* szStr_);
00129
00130
          static bool CompareStrings(const char* szStr1_, const char* szStr2_); static bool CompareStrings(const char* szStr1_, const char* szStr2_, uint16_t u16Length_)
00140
00141
00142
00143
00153
          static void CopyMemory(void* pvDst_, const void* pvSrc_, uint16_t u16Len_);
00154
00155
00164
          static void CopyString(char* szDst_, const char* szSrc_);
00165
00166
00176
          static int16_t StringSearch(const char* szBuffer_, const char* szPattern_);
00177
00178
00190
          static bool CompareMemory(const void* pvMem1_, const void* pvMem2_, uint16_t u16Len_);
00191
00192
          static void SetMemory(void* pvDst_, uint8_t u8Val_, uint16_t u16Len_);
00203
00204
          static uint8_t Tokenize(const char* szBuffer_, Token_t* pastTokens_, uint8_t
00214
      u8MaxTokens_);
00215 };
00216 } // namespace Mark3
```

# 20.11 /home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h File Reference

Thread/Interrupt-safe byte-based data streaming.

```
#include "kerneltypes.h"
#include "mark3.h"
```

# Classes

· class Mark3::Streamer

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

#### **Namespaces**

• Mark3

# 20.11.1 Detailed Description

Thread/Interrupt-safe byte-based data streaming.

Definition in file streamer.h.

# 20.12 streamer.h

```
00001 /*
00002
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00020 #include "kerneltypes.h"
00021 #include "mark3.h"
00022
00023 #pragma once
00024
00025 namespace Mark3
00026 {
00027 //-
00035 class Streamer
00036 {
00037 public:
          void Init(uint8_t* pau8Buffer_, uint16_t u16Size_);
00046
00047
00056
          bool Read(uint8 t* pu8Data );
00057
00067
          uint16_t Read(uint8_t* pu8Data_, uint16_t u16Len_);
00068
00077
          bool Write(uint8_t u8Data_);
00078
00087
          uint16_t Write(uint8_t* pu8Data_, uint16_t u16Len_);
00088
00109
          bool Claim(uint8_t** pu8Addr_);
00110
00120
          void Lock(uint8_t* pu8LockAddr_);
00121
          void Unlock (void):
00128
00129
00134
          uint16_t GetAvailable(void) { return m_u16Size; }
00139
          bool CanRead(void);
00140
00145
          bool CanWrite (void);
00146
          bool IsEmpty(void);
00151
00152
00153 private:
00154
          uint8_t* m_pau8Buffer;
00155
          uint8_t* m_pu8LockAddr;
00156
          uint16_t m_u16Size;
00157
          uint16_t m_u16Avail;
00159
          uint16_t m_u16Head;
00160
          uint16_t m_u16Tail;
00161 };
00162 } // namespace Mark3
```

# 20.13 /home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp File Reference

Thread/Interrupt-safe byte-based data streaming.

```
#include "kerneltypes.h"
#include "mark3.h"
#include "streamer.h"
```

# **Namespaces**

• Mark3

20.14 streamer.cpp 435

# 20.13.1 Detailed Description

Thread/Interrupt-safe byte-based data streaming.

Definition in file streamer.cpp.

# 20.14 streamer.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =========
00020 #include "kerneltypes.h"
00021 #include "mark3.h"
00022 #include "streamer.h"
00023
00024 namespace Mark3
00025 {
00026 //--
00027 void Streamer::Init(uint8_t* pau8Buffer_, uint16_t u16Size_)
00028 {
00029
          m\_u16Head
                        = 0;
00030
          m\_ulfail
                        = 0;
                        = u16Size_;
00031
          m u16Size
00032
          m_u16Avail
                        = m_u16Size;
          m_pau8Buffer = pau8Buffer_;
m_pu8LockAddr = 0;
00033
00034
00035 }
00036
00037 //--
00038 bool Streamer::Read(uint8_t* pu8Data_)
00039 {
00040
          auto rc = true;
00041
00042
          const auto cs = CriticalGuard{};
00043
00044
          if (m_u16Avail == m_u16Size) {
00045
              rc = false;
00046
          } else {
00047
              auto* pu8Dest = &m_pau8Buffer[m_u16Tail];
              if (pu8Dest == m_pu8LockAddr) {
00048
00049
                   rc = false;
00050
              } else {
                  *pu8Data_ = *pu8Dest;
m_u16Tail++;
00051
00052
00053
                   if (m_u16Tail >= m_u16Size) {
00054
                       m_u16Tail = 0;
00055
00056
                  m_u16Avail++;
00057
00058
          }
00059
00060
          return rc;
00061 }
00062
00063 //--
00064 uint16_t Streamer::Read(uint8_t* pu8Data_, uint16_t u16Len_)
00065 {
00066
          uint16_t u16ToRead;
00067
00068
          if (m_pu8LockAddr != 0) {
00069
              return 0;
00070
00071
00072
          uint16_t u16Allocated;
00073
          uint16_t u16PreWrap;
00074
          uint8_t* pu8Src;
00075
          uint8_t* pu8Dst;
00076
          { // Begin critical section
```

```
const auto cs = CriticalGuard{};
00079
               u16Allocated = m_u16Size - m_u16Avail;
00080
00081
               if (u16Allocated > u16Len_) {
00082
                  u16ToRead = u16Len_;
               } else {
00083
                   u16ToRead = u16Allocated;
00085
               }
00086
00087
               u16PreWrap = m_u16Size - m_u16Tail;
00088
00089
               pu8Src = &m_pau8Buffer[m_u16Tail];
              pu8Dst = pu8Data_;
00090
00091
00092
               Lock (pu8Src);
00093
          } // end critical section
00094
00095
          if (u16Allocated != 0u) {
              if (u16PreWrap >= u16ToRead) {
00096
00097
                   for (uint16_t i = 0; i < u16ToRead; i++) { *pu8Dst++ = *pu8Src++; }</pre>
00098
00099
                   for (uint16_t i = 0; i < u16PreWrap; i++) { *pu8Dst++ = *pu8Src++; }</pre>
00100
                   pu8Src = m_pau8Buffer;
                   for (uint16_t i = u16PreWrap; i < u16ToRead; i++) { *pu8Dst++ = *pu8Src++; }</pre>
00101
00102
              }
00103
          }
00104
00105
          { // Begin critical section
              const auto cs = CriticalGuard{};
m_ul6Avail += ul6ToRead;
00106
00107
              if (u16PreWrap >= u16ToRead) {
    m_u16Tail += u16ToRead;
00108
00109
00110
00111
                   m_u16Tail += u16ToRead - m_u16Size;
00112
              }
00113
00114
              Unlock();
          } // end critical section
00115
00116
          return u16ToRead;
00117 }
00118
00119 //---
00120 bool Streamer::Write(uint8_t u8Data_)
00121 {
00122
          auto rc = true;
00123
00124
          const auto cs = CriticalGuard{};
          if (m_u16Avail == 0u) {
00125
00126
              rc = false;
00127
          } else {
              if (m_pu8LockAddr == &m_pau8Buffer[m_u16Head]) {
00128
00129
00130
              } else {
00131
                  m_pau8Buffer[m_u16Head] = u8Data_;
00132
                   m_u16Head++;
                   if (m_u16Head >= m_u16Size) {
00133
00134
                      m_u16Head = 0;
00135
00136
                   m_u16Avail--;
00137
              }
00138
          }
00139
00140
          return rc;
00141 }
00142
00143 //----
00144 uint16_t Streamer::Write(uint8_t* pu8Data_, uint16_t u16Len_)
00145 {
00146
          uint16_t u16ToWrite;
00147
00148
          \ensuremath{//} Bail if the buffer is currently locked.
00149
          if (m_pu8LockAddr != 0) {
             return 0;
00150
00151
00152
00153
          // Update the buffer metadata in a critical section, and lock it so that
00154
          // we can safely write to it with interrupts enabled.
00155
          uint16_t u16PreWrap;
00156
          uint8_t* pu8Src;
uint8_t* pu8Dst;
00157
00158
00159
          { // Begin critical section
00160
00161
              const auto cs = CriticalGuard{};
              if (m_ul6Avail > ul6Len_) {
    ul6ToWrite = ul6Len_;
00162
00163
              } else {
00164
```

20.14 streamer.cpp 437

```
00165
                  u16ToWrite = m_u16Avail;
00166
00167
00168
              u16PreWrap = m_u16Size - m_u16Head;
00169
00170
              pu8Src = pu8Data_;
00171
              pu8Dst = &m_pau8Buffer[m_u16Head];
00172
00173
              m_u16Avail -= u16ToWrite;
00174
              if (u16PreWrap >= u16ToWrite) {
00175
                  m_u16Head += u16ToWrite;
00176
00177
              } else {
00178
                 m_ul6Head += ul6ToWrite - m_ul6Size;
00179
              }
00180
00181
              Lock(pu8Dst);
         } // End critical section
00182
00183
00184
          // Perform the buffer writes with interrupts enabled, buffers locked.
          if (u16ToWrite != 0u) {
00185
00186
              if (u16PreWrap >= u16ToWrite) {
                  for (uint16_t i = 0; i < u16ToWrite; i++) { *pu8Dst++ = *pu8Src++; }</pre>
00187
00188
              } else {
00189
                  for (uint16_t i = 0; i < u16PreWrap; i++) { *pu8Dst++ = *pu8Src++; }</pre>
00190
                  pu8Dst = m_pau8Buffer;
00191
                   for (uint16_t i = u16PreWrap; i < u16ToWrite; i++) { *pu8Dst++ = *pu8Src++; }
00192
00193
         }
00194
00195
          Unlock();
00196
          return u16ToWrite;
00197 }
00198
00199 //--
00200 bool Streamer::CanRead(void)
00201 {
          auto bRc = true;
          const auto cs = CriticalGuard{};
if (m_u16Avail == m_u16Size) {
00203
00204
00205
             bRc = false;
00206
00207
          return bRc:
00208 }
00209
00210 //---
00211 bool Streamer::CanWrite(void)
00212 {
00213
          auto bRc = false;
00214
00215
          const auto cs = CriticalGuard{};
00216
         if (m_u16Avail != 0u) {
00217
             bRc = true;
00218
00219
00220
          return bRc;
00221 }
00222
00223 //---
00224 bool Streamer::Claim(uint8_t** pu8Addr_)
00225 {
00226
          auto rc = true;
00227
00228
          const auto cs = CriticalGuard{};
00229
          if (m_u16Avail == 0u) {
00230
             rc = false;
00231
          } else {
00232
              if (m_pu8LockAddr == &m_pau8Buffer[m_u16Head]) {
00233
                  rc = false:
00234
              } else {
00235
                 *pu8Addr_ = &m_pau8Buffer[m_u16Head];
00236
                  if (m_pu8LockAddr == 0) {
                      m_pu8LockAddr = &m_pau8Buffer[m_u16Head];
00237
00238
00239
                  m_u16Head++;
00240
                  if (m_u16Head >= m_u16Size) {
00241
                      m_u16Head = 0;
00242
00243
                  m_u16Avail--;
             }
00244
00245
         }
00246
          return rc;
00247 }
00248
00249 //---
00250 void Streamer::Lock(uint8_t* pu8LockAddr_)
00251 {
```

```
const auto cs = CriticalGuard{};
00253
          m_pu8LockAddr = pu8LockAddr_;
00254 }
00255
00256 //---
00257 void Streamer::Unlock(void)
00258 {
00259
          const auto cs = CriticalGuard{};
00260
          m_pu8LockAddr = 0;
00261 }
00262
00263 //--
00264 bool Streamer::IsEmpty(void)
00265 {
00266
          const auto cs = CriticalGuard{};
00267
          return m_u16Avail == m_u16Size;
00268 3
00269 } // namespace Mark3
```

# 20.15 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp File Reference

Kernel Software interrupt implementation for ATMega1284p.

```
#include "kerneltypes.h"
#include "kernelswi.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

# **Namespaces**

Mark3

# 20.15.1 Detailed Description

Kernel Software interrupt implementation for ATMega1284p.

Definition in file kernelswi.cpp.

# 20.16 kernelswi.cpp

```
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00022 #include "kerneltypes.h"
00023 #include "kernelswi.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 namespace Mark3
```

```
00029 {
00030 //--
00031 void KernelSWI::Config(void)
00032 {
                                   // Clear INT2
// Set PortB, bit 2 (INT2) As Output
00038 //----
00039 void KernelSWI::Start(void)
00040 {
        EIFR &= \sim (1 << INTF2); // Clear any pending interrupts on INT2 EIMSK |= (1 << INT2); // Enable INT2 interrupt (as int32_t as I-bit is set)
00041
00043 }
00044
00045 //---
00046 void KernelSWI::Trigger(void)
00047 {
         // if(Thread_IsSchedulerEnabled())
00049
00050
             PORTB &= \sim 0 \times 04;
00051
00052 }
            PORTB |= 0x04;
00053 }
00054 } // namespace Mark3
```

# 20.17 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp File Reference

Kernel Timer Implementation for ATMega1284p.

```
#include "mark3.h"
#include <avr/common.h>
#include <avr/io.h>
#include <avr/interrupt.h>
```

# **Namespaces**

Mark3

# **Macros**

- #define TCCR1B INIT ((1 << WGM12) | (1 << CS12))</li>
- #define TIMER\_IMSK (1 << OCIE1A)</li>
- #define TIMER\_IFR (1 << OCF1A)</li>

# **Functions**

- static void Mark3::KernelTimer\_Task (void \*unused)
- ISR (TIMER1\_COMPA\_vect)

# 20.17.1 Detailed Description

Kernel Timer Implementation for ATMega1284p.

Definition in file kerneltimer.cpp.

# 20.17.2 Macro Definition Documentation

```
20.17.2.1 TCCR1B_INIT
\#define TCCR1B_INIT ((1 << WGM12) | (1 << CS12))
Definition at line 27 of file kerneltimer.cpp.
20.17.2.2 TIMER_IFR
#define TIMER_IFR (1 << OCF1A)</pre>
Definition at line 29 of file kerneltimer.cpp.
20.17.2.3 TIMER_IMSK
\#define TIMER_IMSK (1 << OCIE1A)
Definition at line 28 of file kerneltimer.cpp.
20.17.3 Function Documentation
20.17.3.1 ISR()
```

Definition at line 103 of file kerneltimer.cpp.

TIMER1\_COMPA\_vect )

ISR (

20.18 kerneltimer.cpp 441

# 20.18 kerneltimer.cpp

```
00001 /*=======
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00021 #include "mark3.h"
00023 #include <avr/common.h>
00024 #include <avr/io.h>
00025 #include <avr/interrupt.h>
00026
00027 #define TCCR1B_INIT ((1 << WGM12) | (1 << CS12))
00028 #define TIMER_IMSK (1 << OCIE1A)
00029 #define TIMER_IFR (1 << OCF1A)
00030
00031 namespace
00032 {
00033 using namespace Mark3;
00035 ^{\prime\prime} Static objects implementing the timer thread and its synchronization objects
00038 Semaphore s_clTimerSemaphore;
00039 }
00040
00041 namespace Mark3
00042 {
00043 //----
00044 static void KernelTimer_Task(void* unused)
00045 {
00046
          (void) unused;
00047
         while (1) {
00048
              s_clTimerSemaphore.Pend();
00049 #if KERNEL_ROUND_ROBIN
00050 Quantum::SetInTimer();
00051 #endif // #if KERNEL_ROUND_ROBIN
00052 TimerScheduler::Process();
00053 #if KERNEL_ROUND_ROBIN
00054
             Quantum::ClearInTimer();
00055 #endif // #if KERNEL_ROUND_ROBIN
00056
        }
00057 }
00058
00059 //--
00060 void KernelTimer::Config(void)
00061 {
00062
          TCCR1B = TCCR1B_INIT;
00063
         s_clTimerSemaphore.Init(0, 1);
          s_clTimerThread.Init(s_clTimerThreadStack,
00064
00065
                               sizeof(s_clTimerThreadStack) / sizeof(K_WORD),
00066
                               KERNEL_TIMERS_THREAD_PRIORITY,
00067
                               KernelTimer_Task,
00068
                               0);
00069 #if KERNEL_ROUND_ROBIN
00070
         Ouantum::SetTimerThread(&s clTimerThread);
00071 #endif // #if KERNEL_ROUND_ROBIN
         s_clTimerThread.Start();
00073 }
00074
00075 //---
00076 void KernelTimer::Start(void)
00077 {
          TCCR1B = ((1 << WGM12) | (1 << CS11) | (1 << CS10));
00079
          OCR1A = ((PORT_SYSTEM_FREQ / 1000) / 64);
08000
         TCNT1 = 0;
TIFR1 &= ~TIMER_IFR;
00081
          TIMSK1 |= TIMER_IMSK;
00082
00083 }
00084
00086 void KernelTimer::Stop(void)
00087 {
00088
          TIFR1 &= ~TIMER_IFR;
          TIMSK1 &= ~TIMER_IMSK;
00089
00090
          TCCR1B &= ~(1 << CS12); // Disable count...
          TCNT1 = 0;
```

# 20.19 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/portcfg.h File Reference

Mark3 Port Configuration.

#### **Macros**

```
    #define KERNEL_NUM_PRIORITIES (16)
```

- #define KERNEL\_TIMERS\_THREAD\_PRIORITY (KERNEL\_NUM\_PRIORITIES 1)
- #define THREAD\_QUANTUM\_DEFAULT (4)
- #define KERNEL\_STACK\_GUARD\_DEFAULT (32)
- #define PORT\_COROUTINE\_PRIORITIES (8)
- #define AVR (1)
- #define K WORD uint8 t

Size of a data word.

• #define K ADDR uint16 t

Size of an address (pointer size)

- #define K\_INT int32\_t
- #define PORT PRIO TYPE uint8 t

Type used for bitmap in the PriorityMap class.

• #define PORT\_PRIO\_MAP\_WORD\_SIZE (1)

size of PORT\_PRIO\_TYPE in bytes

• #define PORT\_SYSTEM\_FREQ ((uint32\_t)16000000)

CPU Frequency in Hz.

#define PORT\_TIMER\_FREQ ((uint32\_t)(PORT\_SYSTEM\_FREQ / 1000))

Fixed timer interrupt frequency.

- #define PORT\_KERNEL\_DEFAULT\_STACK\_SIZE ((K\_ADDR)384)
- #define PORT\_KERNEL\_TIMERS\_THREAD\_STACK ((K\_ADDR)384)
- #define PORT\_TIMER\_COUNT\_TYPE uint16\_t

Timer counter type.

- #define PORT\_MIN\_TIMER\_TICKS (0)
- #define PORT\_OVERLOAD\_NEW (1)
- #define PORT\_STACK\_GROWS\_DOWN (1)
- #define PORT\_USE\_HW\_CLZ (1)

# 20.19.1 Detailed Description

#### Mark3 Port Configuration.

This file is used to configure the kernel for your specific target CPU in order to provide the optimal set of features for a given use case.

!! NOTE: This file must ONLY be included from mark3cfg.h

Definition in file portcfg.h.

#### 20.19.2 Macro Definition Documentation

```
20.19.2.1 AVR
```

```
#define AVR (1)
```

Define a macro indicating the CPU architecture for which this port belongs.

This may also be set by the toolchain, but that's not guaranteed.

Definition at line 55 of file portcfg.h.

#### 20.19.2.2 K\_ADDR

```
#define K_ADDR uint16_t
```

Size of an address (pointer size)

#### **Examples:**

lab9\_dynamic\_threads/main.cpp.

Definition at line 63 of file portcfg.h.

#### 20.19.2.3 K\_INT

```
#define K_INT int32_t
```

Definition at line 64 of file portcfg.h.

# 20.19.2.4 K\_WORD

```
#define K_WORD uint8_t
```

Size of a data word.

Define types that map to the CPU Architecture's default data-word and address size.

# Examples:

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 62 of file portcfg.h.

#### 20.19.2.5 KERNEL\_NUM\_PRIORITIES

```
#define KERNEL_NUM_PRIORITIES (16)
```

Define the number of thread priorities that the kernel's scheduler will support. The number of thread priorities is limited only by the memory of the host CPU, as a ThreadList object is statically-allocated for each thread priority.

In practice, systems rarely need more than 32 priority levels, with the most complex having the capacity for 256.

Definition at line 35 of file portcfg.h.

#### 20.19.2.6 KERNEL STACK GUARD DEFAULT

```
#define KERNEL_STACK_GUARD_DEFAULT (32)
```

Definition at line 41 of file portcfg.h.

#### 20.19.2.7 KERNEL\_TIMERS\_THREAD\_PRIORITY

```
#define KERNEL_TIMERS_THREAD_PRIORITY (KERNEL_NUM_PRIORITIES - 1)
```

Definition at line 37 of file portcfg.h.

### 20.19.2.8 PORT\_COROUTINE\_PRIORITIES

```
#define PORT_COROUTINE_PRIORITIES (8)
```

Set the number of priorities supported by the coroutine scheduler. The number of coroutine priorities is limited by the memory of the host CPU.

Definition at line 47 of file portcfg.h.

### 20.19.2.9 PORT\_KERNEL\_DEFAULT\_STACK\_SIZE

```
#define PORT_KERNEL_DEFAULT_STACK_SIZE ((K_ADDR)384)
```

Define the default/minimum size of a thread stack

#### **Examples:**

lab10\_notifications/main.cpp, lab11\_mailboxes/main.cpp, lab1\_kernel\_setup/main.cpp, lab2\_coroutines/main.⇔ cpp, lab3\_round\_robin/main.cpp, lab4\_semaphores/main.cpp, lab5\_mutexes/main.cpp, lab6\_timers/main.⇔ cpp, lab7\_events/main.cpp, lab8\_messages/main.cpp, and lab9\_dynamic\_threads/main.cpp.

Definition at line 96 of file portcfg.h.

```
20.19.2.10 PORT_KERNEL_TIMERS_THREAD_STACK
```

```
#define PORT_KERNEL_TIMERS_THREAD_STACK ((K_ADDR)384)
```

Define the size of the kernel-timer thread stack (if one is configured)

Definition at line 101 of file portcfg.h.

#### 20.19.2.11 PORT\_MIN\_TIMER\_TICKS

```
#define PORT_MIN_TIMER_TICKS (0)
```

Minimum number of timer ticks for any delay or sleep, required to ensure that a timer cannot be initialized to a negative value.

Definition at line 112 of file portcfg.h.

#### 20.19.2.12 PORT\_OVERLOAD\_NEW

```
#define PORT_OVERLOAD_NEW (1)
```

Set this to 1 to overload the system's New/Free functions with the kernel's allocator functions. A user must configure the Kernel's allocator functions to point to a real heap implementation backed with real memory in order to use dynamic object creation.

Definition at line 119 of file portcfg.h.

```
20.19.2.13 PORT_PRIO_MAP_WORD_SIZE
```

```
#define PORT_PRIO_MAP_WORD_SIZE (1)
```

size of PORT\_PRIO\_TYPE in bytes

Definition at line 74 of file portcfg.h.

#### 20.19.2.14 PORT\_PRIO\_TYPE

```
#define PORT_PRIO_TYPE uint8_t
```

Type used for bitmap in the PriorityMap class.

Set a base datatype used to represent each element of the scheduler's priority bitmap.

PORT\_PRIO\_MAP\_WORD\_SIZE should map to the *size* of an element of type PORT\_PROI\_TYPE.

Definition at line 73 of file portcfg.h.

# 20.19.2.15 PORT\_STACK\_GROWS\_DOWN

```
#define PORT_STACK_GROWS_DOWN (1)
```

Set this to 1 if the stack grows down in the target architecture, or 0 if the stack grows up

Definition at line 124 of file portcfg.h.

#### 20.19.2.16 PORT\_SYSTEM\_FREQ

```
#define PORT_SYSTEM_FREQ ((uint32_t)16000000)
```

CPU Frequency in Hz.

Define the running CPU frequency. This may be an integer constant, or an alias for another variable which holds the CPU's current running frequency.

Definition at line 81 of file portcfg.h.

# 20.19.2.17 PORT\_TIMER\_COUNT\_TYPE

```
#define PORT_TIMER_COUNT_TYPE uint16_t
```

Timer counter type.

Define the native type corresponding to the kernel timer hardware's counter register.

Definition at line 106 of file portcfg.h.

# 20.19.2.18 PORT\_TIMER\_FREQ

```
#define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))
```

Fixed timer interrupt frequency.

Set the timer frequency. If running in tickless mode, this is simply the frequency at which the free-running kernel timer increments.

In tick-based mode, this is the frequency at which the fixed-frequency kernel tick interrupt occurs.

Definition at line 91 of file portcfg.h.

20.20 portcfg.h 447

#### 20.19.2.19 PORT\_USE\_HW\_CLZ

```
#define PORT_USE_HW_CLZ (1)
```

Set this to 1 if the target CPU/toolchain supports an optimized Count-leading-zeros instruction, or count-leading-zeros intrinsic. If such functionality is not available, a general-purpose implementation will be used.

Definition at line 131 of file portcfg.h.

#### 20.19.2.20 THREAD QUANTUM DEFAULT

```
#define THREAD_QUANTUM_DEFAULT (4)
```

Definition at line 39 of file portcfg.h.

# 20.20 portcfg.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----*/
00024 #pragma once
00025
00035 #define KERNEL_NUM_PRIORITIES (16)
00036
00037 #define KERNEL_TIMERS_THREAD_PRIORITY (KERNEL_NUM_PRIORITIES - 1)
00038
00039 #define THREAD_QUANTUM_DEFAULT (4)
00040
00041 #define KERNEL_STACK_GUARD_DEFAULT (32) // words
00042
00047 #define PORT_COROUTINE_PRIORITIES (8)
00048
00054 #ifndef AVR
00055 #define AVR (1)
00056 #endif
00057
00062 #define K_WORD uint8_t
00063 #define K_ADDR uint16_t
00064 #define K_INT int32_t
00065
00073 #define PORT_PRIO_TYPE uint8_t
00074 #define PORT_PRIO_MAP_WORD_SIZE (1)
00075
00076
00080 #if !defined(PORT_SYSTEM_FREQ)
00081 #define PORT_SYSTEM_FREQ ((uint32_t)16000000)
00082 #endif
00083
00091 #define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))
00092
00093
00096 #define PORT_KERNEL_DEFAULT_STACK_SIZE ((K_ADDR)384)
00097
00101 #define PORT_KERNEL_TIMERS_THREAD_STACK ((K_ADDR)384)
00102
00106 #define PORT TIMER COUNT TYPE uint16 t
00107
00108
00112 #define PORT_MIN_TIMER_TICKS (0)
00113
00119 #define PORT_OVERLOAD_NEW (1)
00120
00124 #define PORT_STACK_GROWS_DOWN (1)
00125
00131 #define PORT_USE_HW_CLZ
```

# 20.21 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/threadport.h File Reference

# ATMega1284p Multithreading support.

```
#include "portcfg.h"
#include "kerneltypes.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

# **Namespaces**

· Mark3

#### **Macros**

- #define ASM(x) asm volatile(x);
- #define PORT\_TOP\_OF\_STACK(x, y) (reinterpret\_cast<K\_WORD\*>(reinterpret\_cast<K\_ADDR>(x) + (static\_cast<K\_ADDR>(y) 1)))

Macro to find the top of a stack given its size and top address.

• #define PORT\_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.

#### **Functions**

- uint8\_t Mark3::PORT\_CLZ (uint8\_t in\_)
- void Mark3::PORT\_IRQ\_ENABLE ()
- void Mark3::PORT IRQ DISABLE ()
- void Mark3::PORT\_CS\_ENTER ()
- void Mark3::PORT CS EXIT ()
- K\_WORD Mark3::PORT\_CS\_NESTING ()

#### **Variables**

- static constexpr auto Mark3::SR\_ = uint8\_t{0x3F}
- K\_WORD Mark3::g\_kwSFR
- K\_WORD Mark3::g\_kwCriticalCount

# 20.21.1 Detailed Description

ATMega1284p Multithreading support.

Definition in file threadport.h.

# 20.21.2 Macro Definition Documentation

# 20.21.2.1 ASM

```
#define ASM( x ) asm volatile(x);
```

Definition at line 32 of file threadport.h.

#### 20.21.2.2 PORT\_PUSH\_TO\_STACK

Push a value y to the stack pointer x and decrement the stack pointer.

Definition at line 38 of file threadport.h.

#### 20.21.2.3 PORT\_TOP\_OF\_STACK

Macro to find the top of a stack given its size and top address.

Definition at line 36 of file threadport.h.

# 20.21.2.4 Thread\_RestoreContext

```
#define Thread_RestoreContext( )
```

Restore the context of the Thread.

Definition at line 91 of file threadport.h.

#### 20.21.2.5 Thread\_SaveContext

```
#define Thread_SaveContext()
```

Save the context of the Thread.

Definition at line 42 of file threadport.h.

# 20.22 threadport.h

```
00001 /*----
00002
00003
00004
00005
00006 |
00007
00008
00009 -- [Mark3 Realtime Platform]-----
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00020 #pragma once
00021
00022 #include "portcfg.h"
00023 #include "kerneltypes.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 namespace Mark3
00029 {
00030 // clang-format off
00031 //--
00032 #define ASM(x)
                             asm volatile(x);
00033
00034 //----
00036 #define PORT_TOP_OF_STACK(x, y) (reinterpret_cast<K_WORD*>(reinterpret_cast<K_ADDR>(x) +
         (static_cast<K_ADDR>(y) - 1)))
00037 #define PORT_PUSH_TO_STACK(x, y)
00039
00040 //----
00042 #define Thread_SaveContext() \
00042 #GETINE INCOMES TO 1); \
00044 ASM("push r0"); \
00044 ASM("in r0, __SREG__"); \
00045 ASM("cli");
00046 ASM("push r0");
00047 ASM("push r1");
00048 ASM("clr r1");
00049 ASM("push r2");
00050 ASM("push r3");
00051 ASM("push r4");
00052 ASM("push r5");
00053 ASM("push r6");
00054 ASM("push r7");
00055 ASM("push r8");
00056 ASM("push r9");
00057 ASM("push r10");
00058 ASM("push r11");
00059 ASM("push r12");
00060 ASM("push r13");
00061 ASM("push r14");
00062 ASM("push r15");
00063 ASM("push r16");
00064 ASM("push r17");
00065 ASM("push r18");
00066 ASM("push r19");
00066 ASM("push r20");
00067 ASM("push r20");
00068 ASM("push r21");
00069 ASM("push r22");
00070 ASM("push r23");
00071 ASM("push r24");
00072 ASM("push r25");
00073 ASM("push r26");
00074 ASM("push r27");
00075 ASM("push r28");
00076 ASM("push r29");
```

20.22 threadport.h 451

```
00077 ASM("push r30");
00078 ASM("push r31");
00079 ASM("in r0, 0x
                 r0, 0x3B"); \
00082 ASM("lds r27, g_pclCurrent + 1"); \
00083 ASM("adiw r26, 4"); \
00084 ASM("in r0, 0x3D"); \
00085 ASM("in r0, 0x3E"); \
00087 ASM("st x+, r0"); \
00088
00089 //----
00091 #define Thread_RestoreContext() \
00092 ASM("lds r26, g_pclCurrent");
00093 ASM("lds r27, g_pclCurrent + 1");\
00094 ASM("adiw r26, 4"); \
00095 ASM("ld
                 r28, x+");
00096 ASM("out 0x3D, r28"); \
00097 ASM("ld
                 r29, x+");
00098 ASM("out 0x3E, r29"); \
00099 ASM("pop r0"); \setminus
00100 ASM("out 0x3B, r0"); \
00101 ASM("pop r31");
00102 ASM("pop r30");
00103 ASM("pop r29");
00104 ASM("pop r28");
00105 ASM("pop r27");
00106 ASM("pop r26");
00107 ASM("pop r25");
00108 ASM("pop r24");
00109 ASM("pop r23");
00110 ASM("pop r22");
00111 ASM("pop r21");
00112 ASM("pop r20");
00113 ASM("pop r19");
00114 ASM("pop r18");
00115 ASM("pop r17");
00116 ASM("pop r16");
00117 ASM("pop r15");
00118 ASM("pop r14");
00119 ASM("pop r13");
00120 ASM("pop r12");
00121 ASM("pop r11");
00122 ASM("pop r10");
00123 ASM("pop r9");
00124 ASM("pop r8");
00125 ASM("pop r7");
00126 ASM("pop r6");
00127 ASM("pop r5");
00128 ASM("pop r4");
00129 ASM("pop r3");
00130 ASM("pop r2");
00131 ASM("pop r1");
00132 ASM("pop r0"); \
00132 ASM( pop 10 ), \
00133 ASM("out __SREG__, r0"); \
00134 ASM("pop r0");
00135
00136 //----
00137 static constexpr auto SR_ = uint8_t{0x3F};
00138 extern "C" {
00139 extern K_WORD g_kwSFR;
00140
         extern K_WORD g_kwCriticalCount;
00141 }
00142
00143 //----
00144 inline uint8_t PORT_CLZ(uint8_t in_)
00145 {
00146
          static const uint8_t u8Lookup[] = {4, 3, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
          uint8_t hi = __builtin_avr_swap(in_) & 0x0F;
00148
          <u>if</u> (hi) {
00149
             return u8Lookup[hi];
00150
          return 4 + u8Lookup[in_];
00151
00152 }
00153
00154 //--
00155 inline void PORT_IRQ_ENABLE()
00156 {
          ASM("sei"):
00157
00158 }
00160 //---
00161 inline void PORT_IRQ_DISABLE()
00162 {
          ASM("cli");
00163
00164 }
```

```
00167 inline void PORT_CS_ENTER()
00168 {
         auto u8SFR = \_SFR\_IO8(SR\_);
00169
        ASM("cli");
if (!g_kwCriticalCount) {
00170
00171
            g_kwSFR = u8SFR;
00172
00176
00177 //--
00178 inline void PORT_CS_EXIT()
00179 {
00180
         g_kwCriticalCount--;
        if (!g_kwCriticalCount) {
00181
           _SFR_IO8(SR_) = g_kwSFR;
00182
00183
00184 }
00185
00186 //----
00187 inline K_WORD PORT_CS_NESTING()
00188 {
00189
         return g_kwCriticalCount;
00190 }
00191 } // namespace Mark3
```

# 20.23 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp File Reference

# ATMega1284p Multithreading.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "threadport.h"
#include "kernelswi.h"
#include "timerlist.h"
#include "quantum.h"
#include "kernel.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

#### Namespaces

• Mark3

#### **Functions**

- static void Mark3::Thread\_Switch (void)
- Mark3::ISR (INT2\_vect) \_\_attribute\_\_((signal

ISR(INT2\_vect) SWI using INT2 - used to trigger a context switch.

# Variables

- K\_WORD g\_kwSFR = 0
- K\_WORD g\_kwCriticalCount = 0
- · Mark3::naked

20.24 threadport.cpp 453

### 20.23.1 Detailed Description

ATMega1284p Multithreading.

Definition in file threadport.cpp.

#### 20.23.2 Variable Documentation

#### 20.23.2.1 g\_kwCriticalCount

```
K_WORD g_kwCriticalCount = 0
```

Definition at line 36 of file threadport.cpp.

#### 20.23.2.2 g\_kwSFR

```
K_WORD g_kwSFR = 0
```

Definition at line 35 of file threadport.cpp.

### 20.24 threadport.cpp

```
00001 /*=
00002
00003
00004
00005
00006 |
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024 #include "thread.h"
00024 #include "threadport.h"
00026 #include "kernelswi.h"
00027 #include "kerneltimer.h"
00028 #include "timerlist.h"
00029 #include "quantum.h"
00030 #include "kernel.h"
00031 #include <avr/io.h>
00032 #include <avr/interrupt.h>
00033
00034 extern "C" {
00035 K_WORD g_kwSFR = 0;
00036 K_WORD g_kwCriticalCount = 0;
00037 } // extern "C"
00038
00039 //---
00040 namespace Mark3
00041 {
00042 //-
00043 void ThreadPort::InitStack(Thread* pclThread_)
00044 {
```

```
// Initialize the stack for a Thread
00046
           uint16_t u16Addr;
00047
          uint8_t* pu8Stack;
00048
          uint16 t i;
00049
00050
           // Get the address of the thread's entry function
00051
          u16Addr = (uint16_t) (pclThread_->m_pfEntryPoint);
00052
00053
           \ensuremath{//} Start by finding the bottom of the stack
00054
          pu8Stack = (uint8_t*)pclThread_->m_pwStackTop;
00055
00056 #if KERNEL STACK CHECK
          // clear the stack, and initialize it to a known-default value (easier // to debug when things go sour with stack corruption or overflow)
00057
00058
00059
           for (i = 0; i < pclThread_->m_ul6StackSize; i++) { pclThread_->
m_pwStack[i] = 0xFF; }
00060 #endif // #if KERNEL_STACK_CHECK
00061
00062
           // Our context starts with the entry function
00063
           PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)(u16Addr & 0x00FF));
00064
           PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)((u16Addr >> 8) & 0x00FF));
00065
00066
           // R0
          PORT_PUSH_TO_STACK(pu8Stack, 0x00); // R0
00067
00068
00069
           // Push status register and R1 (which is used as a constant zero)
00070
           PORT_PUSH_TO_STACK(pu8Stack, 0x80); // SR
00071
          PORT_PUSH_TO_STACK(pu8Stack, 0x00); // R1
00072
00073
           // Push other registers
          for (i = 2; i <= 23; i++) // R2-R23
00074
00075
          {
00076
               PORT_PUSH_TO_STACK(pu8Stack, i);
00077
00078
00079
           // Assume that the argument is the only stack variable
          PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)(((uint16_t)(pclThread_->vArg)) & 0x00FF)); // R24
00080
      m_pvArg)) & 0x00FF));
00081
          PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)((((uint16_t)(pc1Thread_->
      m_pvArg)) >> 8) & 0x00FF)); // R25
00082
00083
          // Push the rest of the registers in the context
for (i = 26; i <= 31; i++) { PORT_PUSH_TO_STACK(pu8Stack, i); }</pre>
00084
00085
00086
          PORT_PUSH_TO_STACK(pu8Stack, 0x00); // RAMPZ
00087
           // Set the top o' the stack.
00088
          pclThread_->m_pwStackTop = (uint8_t*)pu8Stack;
00089
00090
           // That's it! the thread is ready to run now.
00091 }
00092
00093 //---
00094 static void Thread_Switch(void)
00095 {
00096
           g_pclCurrent = (Thread*)g_pclNext;
00097 }
00098
00099 //---
00100 void ThreadPort::StartThreads()
00101 {
          KernelSWI::Config(); // configure the task switch SWI
00102
          KernelTimer::Config(); // configure the kernel timer
00103
00104
00105
           // Tell the kernel that we're ready to start scheduling threads
00106
           // for the first time.
00107
          Kernel::CompleteStart();
00108
           Scheduler::SetScheduler(1); // enable the scheduler
00109
                                        // run the scheduler - determine the first thread to run
00110
          Scheduler::Schedule();
00111
00112
           Thread_Switch(); // Set the next scheduled thread to the current thread
00113
          KernelTimer::Start(); // enable the kernel timer
KernelSWI::Start(); // enable the task switch SWI
00114
00115
00116
00117 #if KERNEL_ROUND_ROBIN
00118
          // Restart the thread quantum timer, as any value held prior to starting
00119
           // the kernel will be invalid. This fixes a bug where multiple threads
00120
           \ensuremath{//} started with the highest priority before starting the kernel causes problems
           // until the running thread voluntarily blocks.
00121
           Quantum::Update(g_pclCurrent);
00122
00123 #endif // #if KERNEL_ROUND_ROBIN
00124
           // Restore the context...
00125
           Thread_RestoreContext(); // restore the context of the first running thread
00126
                                      \ensuremath{//} return from interrupt - will return to the first scheduled thread
00127
           ASM("reti");
00128 }
```

# 20.25 /home/moslevin/projects/m3-repo/kernel/src/atomic.cpp File Reference

Basic Atomic Operations.

```
#include "mark3.h"
```

#### **Namespaces**

Mark3

#### 20.25.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.cpp.

### 20.26 atomic.cpp

```
00001 /*==========
00002
00003
00004
00006 |_
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00021 #include "mark3.h"
00022
00023 namespace Mark3
00024 {
00025 //---
00026 bool Atomic::TestAndSet(bool* pbLock_)
00027 {
00028
         KERNEL_ASSERT (nullptr != pbLock_);
00029
00030
         auto cs = CriticalGuard{};
00031
         auto bRet = *pbLock_;
00032
         if (!bRet) {
            *pbLock_ = 1;
00033
00034
00035
         return bRet;
00036 }
00037 } // namespace Mark3
```

# 20.27 /home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp File Reference

Automatic memory allocation for kernel objects.

```
#include "mark3.h"
#include <stdint.h>
```

#### **Namespaces**

Mark3

#### 20.27.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.cpp.

# 20.28 autoalloc.cpp

```
00001 /*==
00002
00003
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00019 #include "mark3.h"
00020
00021 #include <stdint.h>
00022
00023 //----
00024 // Override new() and delete() using functions provided to AutoAlloc
00025 //--
00026 #if PORT_OVERLOAD_NEW
00027 using namespace Mark3;
00028 void* operator new(size_t n)
00029 {
00030
          return AutoAlloc::NewRawData(n);
00031 }
00032
00033 //-
00034 void* operator new[](size_t n)
00035 {
00036
          return AutoAlloc::NewRawData(n);
00037 }
00038
00039 //-
00040 void operator delete(void* p)
00041 {
00042
         AutoAlloc::DestroyRawData(p);
00043 }
00044
00045 //-
00046 void operator delete[](void* p)
00047 {
00048
         AutoAlloc::DestroyRawData(p);
00049 }
00050 #endif
00051
00052 namespace Mark3
00053 {
```

```
00054 AutoAllocAllocator_t AutoAlloc::m_pfAllocator;
00055 AutoAllocFree_t
                        AutoAlloc::m_pfFree;
00056
00057 //-----
00058 void* AutoAlloc::Allocate(AutoAllocType eType_, size_t sSize_)
00059 {
       return nullptr;
         if (!m_pfAllocator) {
00061
.
00063     return m_pfAllocator(eType_, sSize_);
00064 }
00062
00065
00066 //---
00067 void AutoAlloc::Free(AutoAllocType eType_, void* pvObj_)
00068 {
00069
         if (!m_pfFree) {
        return;
00070
00071
         m_pfFree(eType_, pvObj_);
00073 }
00074
00075 //---
00076 void AutoAlloc::SetAllocatorFunctions(
     AutoAllocAllocator_t pfAllocator_, AutoAllocFree_t pfFree_)
00077 {
00078
         m_pfAllocator = pfAllocator_;
00079
        m_pfFree = pfFree_;
00080 }
00081
00082 //-----
00083 void AutoAlloc::Init()
00084 {
00085 m_pfAllocator = nurry,
00086 m_pfFree = nullptr;
00088
00089 //-
00090 void* AutoAlloc::NewUserTypeAllocation(uint8_t eType_)
00091 {
00092
         return Allocate(static_cast<AutoAllocType>(eType_), 0);
00093 }
00094 //---
00095 void AutoAlloc::DestroyUserTypeAllocation(uint8_t eUserType_, void*
     pvObj_)
00096 {
00097
         Free(static_cast<AutoAllocType>(eUserType_), pvObj_);
00098 }
00099 //---
00100 void* AutoAlloc::NewRawData(size_t sSize_)
00101 {
         return Allocate(AutoAllocType::Raw, sSize_);
00103 }
00104 //--
00105 void AutoAlloc::DestroyRawData(void* pvData_)
00106 {
00107
         Free (AutoAllocType::Raw, pvData_);
00109
00110 } // namespace Mark3
```

### 20.29 /home/moslevin/projects/m3-repo/kernel/src/blocking.cpp File Reference

Implementation of base class for blocking objects.

```
#include "mark3.h"
```

#### **Namespaces**

• Mark3

#### 20.29.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

### 20.30 blocking.cpp

```
00002
00003
00004
00005 1
                 1.11
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00021 #include "mark3.h"
00022
00023 namespace Mark3
00024 {
00025 //-
00026 void BlockingObject::Block(Thread* pclThread_)
00027 {
00028
          KERNEL_ASSERT(nullptr != pclThread_);
00029
00030
          // Remove the thread from its current thread list (the "owner" list)
00031
          // ... And add the thread to this object's block list
          Scheduler::Remove(pclThread_);
00032
00033
          m_clBlockList.Add(pclThread_);
00034
00035
          // Set the "current" list location to the blocklist for this thread
          pclThread_->SetCurrent(&m_clBlockList);
00036
00037
          pclThread_->SetState(ThreadState::Blocked);
00038 }
00039
00040 //
00041 void BlockingObject::BlockPriority(Thread* pclThread_)
00042 {
00043
          KERNEL_ASSERT (nullptr != pclThread_);
00044
00045
             Remove the thread from its current thread list (the "owner" list)
00046
          // ... And add the thread to this object's block list
00047
          Scheduler::Remove(pclThread_);
00048
          m_clBlockList.AddPriority(pclThread_);
00049
00050
          // Set the "current" list location to the blocklist for this thread
00051
          pclThread_->SetCurrent(&m_clBlockList);
00052
          pclThread_->SetState(ThreadState::Blocked);
00053 }
00054
00055 //--
00056 void BlockingObject::UnBlock(Thread* pclThread_)
00057 {
00058
          KERNEL_ASSERT(nullptr != pclThread_);
00059
00060
          // Remove the thread from its current thread list (the "owner" list)
00061
          pclThread_->GetCurrent()->Remove(pclThread_);
00062
          // Put the thread back in its active owner's list. This is usually // the ready-queue at the thread's original priority.
00063
00064
00065
          Scheduler::Add(pclThread_);
00066
00067
          // Tag the thread's current list location to its owner
00068
          pclThread_->SetCurrent(pclThread_->GetOwner());
          pclThread_->SetState(ThreadState::Ready);
00069
00070 }
00071 } // namespace Mark3
```

### 20.31 /home/moslevin/projects/m3-repo/kernel/src/colist.cpp File Reference

CoRoutine List structure implementation.

```
#include "colist.h"
```

#### **Namespaces**

• Mark3

#### 20.31.1 Detailed Description

CoRoutine List structure implementation.

Definition in file colist.cpp.

# 20.32 colist.cpp

```
00001 /*======
00002
00003
00004
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00020 #include "colist.h"
00021
00022 namespace Mark3 {
00023 //-
00024 void CoList::SetPrioMap(CoPrioMap* pclPrioMap_)
00025 {
00026
          m_pclPrioMap = pclPrioMap_;
00027 }
00028
00029 //-
00030 void CoList::SetPriority(PORT_PRIO_TYPE uPriority)
00031 {
00032
          m_uPriority = uPriority;
00033 }
00034
00035 //-
00036 void CoList::Add(Coroutine* pclCoroutine_)
00037 {
00038
          TypedCircularLinkList<Coroutine>::Add(pclCoroutine_);
00039
         PivotForward();
         if (m_pclPrioMap) {
00040
00041
             m_pclPrioMap->Set (m_uPriority);
00042
00043 }
00044
00045 //---
00046 void CoList::Remove(Coroutine* pclCoroutine_)
00047 {
00048
          TypedCircularLinkList<Coroutine>::Remove (pclCoroutine_);
00049
          if (m_pclPrioMap) {
00050
             if (nullptr == GetHead()) {
                  m_pclPrioMap->Clear(m_uPriority);
00051
00052
00053
         }
00054 }
00055 } // namespace Mark3
```

# 20.33 /home/moslevin/projects/m3-repo/kernel/src/condvar.cpp File Reference

Condition Variable implementation.

```
#include "mark3.h"
```

#### **Namespaces**

• Mark3

#### 20.33.1 Detailed Description

Condition Variable implementation.

Definition in file condvar.cpp.

### 20.34 condvar.cpp

```
00001
00002
00003
00004
00006
00007
80000
00009 -- [Mark3 Realtime Platform] --
00010
00011 Copyright (c) 2012 - 2018 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00020 #include "mark3.h"
00021
00022 namespace Mark3
00023 {
00024 //---
00025 void ConditionVariable::Init()
00026 {
00027
          m_clMutex.Init();
00028
          m_clSemaphore.Init(0, 255);
00029 }
00030
00031 //--
00032 void ConditionVariable::Wait (Mutex* pclMutex_)
00033 {
          KERNEL_ASSERT(nullptr != pclMutex_);
00034
00035
00036
          m_clMutex.Claim();
00037
00038
          pclMutex_->Release();
00039
          m_u8Waiters++;
00040
00041
          m clMutex.Release();
00042
00043
          m_clSemaphore.Pend();
00044
          pclMutex_->Claim();
00045 }
00046
00047 //-
00048 bool ConditionVariable::Wait(Mutex* pclMutex_, uint32_t u32WaitTimeMS_)
00049 {
00050
          KERNEL_ASSERT (nullptr != pclMutex_);
00051
00052
          m_clMutex.Claim();
00053
00054
          pclMutex_->Release();
00055
          m_u8Waiters++;
```

```
00056
00057
          m_clMutex.Release();
00058
         if (!m_clSemaphore.Pend(u32WaitTimeMS_)) {
00059
         return false;
}
00060
00061
00062
         return pclMutex_->Claim(u32WaitTimeMS_);
00063 }
00064
00065 //----
00066 void ConditionVariable::Signal()
00067 {
00068 m_clMutex.Claim();
00069 if (m_u8Waiters) {
         m_u8Waiters--;
m_clSemaphore.Post();
00070
00070
00071
00072
00075
00076 //---
00077 void ConditionVariable::Broadcast()
00078 {
00079
          m clMutex.Claim();
08000
00081
00082
        while (m_u8Waiters > 0) {
         m_u8Waiters > 0) {
    m_u8Waiters--;
    m_c1Semaphore.Post();
}
00083
00084
00085
00086
         m_clMutex.Release();
00087 }
00088
00089 } // namespace Mark3
```

# 20.35 /home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp File Reference

Coroutine object implementation.

```
#include "coroutine.h"
#include "cosched.h"
#include "criticalguard.h"
#include "kernel.h"
```

#### Namespaces

• Mark3

#### 20.35.1 Detailed Description

Coroutine object implementation.

Definition in file coroutine.cpp.

### 20.36 coroutine.cpp

```
00001 /*=======
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00020 #include "coroutine.h'
00021 #include "cosched.h"
00022 #include "criticalguard.h"
00023 #include "kernel.h"
00024
00025 namespace Mark3 {
00026
00027 //----
00028 Coroutine::~Coroutine()
00029 {
00030
           if (m_pclOwner != CoScheduler::GetStopList()) {
               Kernel::Panic(PANIC_ACTIVE_COROUTINE_DESCOPED);
00031
00032
00033
00034
          const auto cs = CriticalGuard{};
00035
          m_pclOwner->Remove(this);
00036 }
00037
00038 //---
00039 void Coroutine::Init(PORT_PRIO_TYPE uPriority_,
      CoroutineHandler pfHandler_, void* pvContext_)
00040 {
00041
          m_bQueued = false;
          m_pfHandler = pfHandler_;
m_pvContext = pvContext_;
00042
00043
          m_uPriority = uPriority_;
00044
00045
00046
          m_pclOwner = CoScheduler::GetStopList();
00047
          const auto cs = CriticalGuard{};
00048
          m_pclOwner->Add(this);
00049
00050 }
00051
00052 //--
00053 void Coroutine::Run()
00054 {
          { // Begin critical section
  const auto cs = CriticalGuard{};
00055
00056
               m_pclOwner->Remove(this);
00057
00058
               m_pclOwner = CoScheduler::GetStopList();
00059
               m_pclOwner->Add(this);
               m bOueued = false:
00060
00061
          } // end critical section
00062
00063
          m_pfHandler(this, m_pvContext);
00064 }
00065
00066 //--
00067 void Coroutine::Activate()
00068 {
00069
          const auto cs = CriticalGuard{};
00070
00071
           if (m_bQueued) {
00072
              return;
00073
00074
00075
          m pclOwner->Remove(this);
00076
          m_pclOwner = CoScheduler::GetCoList(
      m_uPriority);
00077
          m_pclOwner->Add(this);
00078
          m_bQueued = true;
00079 }
08000
00081 //--
00082 void Coroutine::SetPriority(PORT_PRIO_TYPE uPriority_)
00083 {
00084
          const auto cs = CriticalGuard{};
00085
00086
          m pclOwner->Remove(this);
00087
          m_uPriority = uPriority_;
88000
          if (m_bQueued) {
```

```
00089
             m_pclOwner = CoScheduler::GetCoList(
      m_uPriority);
00090
        } else {
00091
             m_pclOwner = CoScheduler::GetStopList();
00092
00093
         m_pclOwner->Add(this);
00095
00096 //--
00097 PORT_PRIO_TYPE Coroutine::GetPriority()
00098 {
00099
          return m_uPriority;
00100 }
00101 } // namespace Mark3
```

# 20.37 /home/moslevin/projects/m3-repo/kernel/src/cosched.cpp File Reference

CoRoutine Scheduler implementation.

```
#include "cosched.h"
#include "criticalguard.h"
```

#### **Namespaces**

• Mark3

#### 20.37.1 Detailed Description

CoRoutine Scheduler implementation.

Definition in file cosched.cpp.

# 20.38 cosched.cpp

```
00001 /*=========
00002
00003
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00020 #include "cosched.h"
00021 #include "criticalguard.h"
00023 namespace Mark3 {
00024 CoList CoScheduler::m_aclPriorities[
      PORT_COROUTINE_PRIORITIES];
00025 CoList CoScheduler::m_clStopList;
00026 CoPrioMap CoScheduler::m_clPrioMap;
00028 //--
00029 void CoScheduler::Init()
00030 {
          m_clStopList.Init();
for (auto i = 0; i < PORT_COROUTINE_PRIORITIES; i++) {
    m_aclPriorities[i].SetPriority(i);</pre>
00031
00032
00033
00034
              m_aclPriorities[i].SetPrioMap(&m_clPrioMap);
```

```
00035
00036 }
00037
00038 //---
00039 CoPrioMap* CoScheduler::GetPrioMap()
00040 {
00041
          return &m_clPrioMap;
00042 }
00043
00044 //----
00045 CoList* CoScheduler::GetStopList()
00046 {
00047
          return &m clStopList;
00048 }
00049
00050 //---
00051 CoList* CoScheduler::GetCoList(PORT_PRIO_TYPE uPriority_)
00052 {
          if (uPriority_ >= PORT_COROUTINE_PRIORITIES) {
00053
00054
             return nullptr;
00055
00056
          return &m_aclPriorities[uPriority_];
00057 }
00058
00059 //--
00060 Coroutine* CoScheduler::Schedule()
00061 {
00062
          const auto cs = CriticalGuard{};
00063
00064
          auto uPriority = m_clPrioMap.HighestPriority();
00065
          if (0 == uPriority) {
00066
             return nullptr;
00067
00068
          uPriority--;
00069
00070 }
          return m_aclPriorities[uPriority].GetHead();
00071 } // namespace Mark3
```

### 20.39 /home/moslevin/projects/m3-repo/kernel/src/eventflag.cpp File Reference

Event Flag Blocking Object/IPC-Object implementation.

```
#include "mark3.h"
```

#### 20.39.1 Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

# 20.40 eventflag.cpp

20.40 eventflag.cpp 465

```
00021 #if KERNEL_EVENT_FLAGS
00022
00023 namespace Mark3
00024 {
00025 namespace
00026 {
00038
          void TimedEventFlag_Callback(Thread* pclOwner_, void* pvData_)
00039
00040
              KERNEL_ASSERT (nullptr != pclOwner_);
              KERNEL_ASSERT (nullptr != pvData_);
00041
00042
00043
              auto* pclEventFlag = static_cast<EventFlag*>(pvData_);
00044
00045
              pclOwner_->SetExpired(true);
00046
              pclOwner_->SetEventFlagMask(0);
00047
00048
              pclEventFlag->WakeMe(pclOwner_);
00049
              if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
     GetCurPriority()) {
00050
                 Thread::Yield();
00051
              }
00052
00053 } // anonymous namespace
00054 //---
00055 EventFlag::~EventFlag()
00056 {
00057
           // If there are any threads waiting on this object when it goes out
          // of scope, set a kernel panic.
if (nullptr != m_clBlockList.HighestWaiter()) {
00058
00059
00060
              Kernel::Panic(PANIC_ACTIVE_EVENTFLAG_DESCOPED);
00061
00062 }
00063
00064 //--
00065 void EventFlag::Init()
00066 {
00067
          KERNEL_ASSERT(!m_clBlockList.GetHead());
00068
          m_u16SetMask = 0;
00069
          SetInitialized();
00070 }
00071
00072 //--
00073 void EventFlag::WakeMe(Thread* pclChosenOne_)
00074 {
00075
          KERNEL_ASSERT(IsInitialized());
00076
          KERNEL_ASSERT(nullptr != pclChosenOne_);
00077
00078
          UnBlock (pclChosenOne_);
00079 }
08000
00081 //----
00082 uint16_t EventFlag::Wait_i(uint16_t u16Mask_,
      EventFlagOperation eMode_, uint32_t u32TimeMS_)
00083 {
00084
          KERNEL ASSERT(eMode <= EventFlagOperation::Pending Unblock
      );
00085
          KERNEL_ASSERT(IsInitialized());
00086
00087
          auto bThreadYield = false;
00088
          auto bMatch
                            = false:
00089
00090
          auto clEventTimer = Timer {};
00091
                             = false;
          auto bUseTimer
00092
00093
           // Ensure we're operating in a critical section while we determine
00094
          // whether or not we need to block the current thread on this object.
00095
00096
          { // Begin critical section
00097
              const auto cs = CriticalGuard{};
00098
00099
              // Check to see whether or not the current mask matches any of the
              // desired bits.
00100
00101
              g_pclCurrent->SetEventFlagMask(u16Mask_);
00102
              if ((EventFlagOperation::All_Set == eMode_) || (
      EventFlagOperation::All_Clear == eMode_)) {
00104
                   // Check to see if the flags in their current state match all of
                  // the set flags in the event flag group, with this mask.
if ((m_u16SetMask & u16Mask_) == u16Mask_) {
00105
00106
                      bMatch = true;
00107
00108
                       g_pclCurrent->SetEventFlagMask(u16Mask_);
00109
00110
                       if (EventFlagOperation::All_Clear == eMode_) {
00111
                           m_u16SetMask &= ~u16Mask_;
00112
                           g_pclCurrent->SetExpired(false);
00113
```

```
} else if ((EventFlagOperation::Any_Set == eMode_) || (
     00116
                  // the event flag group with this mask
if ((m_u16SetMask & u16Mask_) != 0) {
00117
00118
                      bMatch = true;
00119
00120
                       g_pclCurrent->SetEventFlagMask(
     m_u16SetMask & u16Mask_);
00121
00122
                      if (EventFlagOperation::Any_Clear == eMode_) {
                           m_u16SetMask &= ~u16Mask_;
00123
00124
                           g_pclCurrent->SetExpired(false);
00125
00126
                  }
00127
              }
00128
00129
              // We're unable to match this pattern as-is, so we must block.
00130
              if (!bMatch) {
00131
                  // Reset the current thread's event flag mask & mode
00132
                  g_pclCurrent->SetEventFlagMask(u16Mask_);
00133
                  g_pclCurrent->SetEventFlagMode(eMode_);
00134
                  if (Ou != u32TimeMS_) {
   g_pclCurrent->SetExpired(false);
00135
00136
00137
                       clEventTimer.Init();
00138
                       clEventTimer.Start(false, u32TimeMS_, TimedEventFlag_Callback, this);
00139
                      bUseTimer = true;
00140
                  }
00141
00142
                   // Add the thread to the object's block-list.
00143
                  BlockPriority(g_pclCurrent);
00144
00145
                  // Trigger that
00146
                  bThreadYield = true;
             }
00147
00148
              // If bThreadYield is set, it means that we've blocked the current thread,
00150
              // and must therefore rerun the scheduler to determine what thread to
00151
              // switch to.
00152
              if (bThreadYield) {
                   // Switch threads immediately
00153
00154
                  Thread::Yield();
00155
         } // end critical section
00156
00157
00162
          if (bUseTimer && bThreadYield) {
00163
              clEventTimer.Stop();
00164
00165
00166
          return g_pclCurrent->GetEventFlagMask();
00167 }
00168
00169 //--
00170 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation eMode_)
00171 {
          KERNEL_ASSERT(eMode_ <= EventFlagOperation::Pending_Unblock</pre>
     );
00173
          return Wait_i(u16Mask_, eMode_, 0);
00174 }
00175
00176 //
00177 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation eMode_,
      uint32 t u32TimeMS )
00178 {
00179
          KERNEL_ASSERT(eMode_ <= EventFlagOperation::Pending_Unblock</pre>
);
00180
          return Wait_i(u16Mask_, eMode_, u32TimeMS_);
00181 }
00182
00183 //--
00184 void EventFlag::Set(uint16_t u16Mask_)
00185 {
          KERNEL ASSERT(IsInitialized());
00186
00187
00188
          auto bReschedule = false;
00189
          const auto cs = CriticalGuard{};
00190
          // 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
00191
00192
          // the threads involved.
00193
00194
          m_u16SetMask |= u16Mask_;
00195
00196
          auto u16NewMask = m_u16SetMask;
00197
          // Start at the head of the list, and iterate through until we hit the
00198
00199
          // "head" element in the list again. Ensure that we handle the case where
```

20.40 eventflag.cpp 467

```
// we remove the first or last elements in the list, or if there's only
           // one element in the list.
00201
00202
00203
           auto* pclCurrent = m_clBlockList.GetHead();
           \ensuremath{//} Do nothing when there are no objects blocking.
00204
00205
           if (nullptr != pclCurrent) {
               // First loop - process every thread in the block-list and check to
00207
               // see whether or not the current flags match the event-flag conditions
00208
               // on the thread.
00209
               auto* pclPrev = (Thread*){};
00210
               do {
                   pclPrev = pclCurrent;
00211
00212
                   pclCurrent = pclCurrent->GetNext();
00213
00214
                   // Read the thread's event mask/mode
                   auto u16ThreadMask = pclPrev->GetEventFlagMask();
auto eThreadMode = pclPrev->GetEventFlagMode();
00215
00216
00217
                    // For the "any" mode - unblock the blocked threads if one or more bits
                   // in the thread's bitmask match the object's bitmask
00219
                    if ((EventFlagOperation::Any_Set == eThreadMode) || (
00220
      EventFlagOperation::Any_Clear == eThreadMode)) {
                     if ((u16ThreadMask & m_u16SetMask) != 0) {
00221
00222
                            pclPrev->SetEventFlagMode(
      EventFlagOperation::Pending_Unblock);
00223
                            pclPrev->SetEventFlagMask(m_u16SetMask & u16ThreadMask);
00224
                            bReschedule = true;
00225
                            // If the "clear" variant is set, then clear the bits in the mask
00226
00227
                            // that caused the thread to unblock.
                            if (EventFlagOperation::Any_Clear == eThreadMode) {
00228
00229
                                 u16NewMask &= ~(u16ThreadMask & u16Mask_);
00230
00231
00232
                   // For the "all" mode, every set bit in the thread's requested bitmask must // match the object's flag mask.
00233
00234
      else if ((EventFlagOperation::All_Set == eThreadMode) || (
EventFlagOperation::All_Clear == eThreadMode)) {
00235
00236
                      if ((u16ThreadMask & m_u16SetMask) == u16ThreadMask) {
00237
                            pclPrev->SetEventFlagMode(
      EventFlagOperation::Pending_Unblock);
                            pclPrev->SetEventFlagMask(u16ThreadMask);
00238
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 (EventFlagOperation::All_Clear == eThreadMode) {
00242
00243
00244
                                 u16NewMask &= ~(u16ThreadMask & u16Mask );
00245
00246
                        }
00247
                   }
00248
               // To keep looping, ensure that there's something in the list, and // that the next item isn't the head of the list.
00249
00250
00251
               while (pclPrev != m_clBlockList.GetTail());
00252
00253
               // Second loop - go through and unblock all of the threads that
00254
               // were tagged for unblocking.
               pclCurrent
00255
                            = m_clBlockList.GetHead();
               auto bIsTail = false;
00256
00257
               do {
00258
                   pclPrev
                              = pclCurrent;
00259
                   pclCurrent = pclCurrent->GetNext();
00260
00261
                    // Check to see if this is the condition to terminate the loop
00262
                    if (pclPrev == m_clBlockList.GetTail()) {
                        bIsTail = true;
00263
00264
00265
00266
                    \ensuremath{//} If the first pass indicated that this thread should be
00267
                    // unblocked, then unblock the thread
00268
                   if (pclPrev->GetEventFlagMode() ==
     EventFlagOperation::Pending_Unblock) {
00269
                       UnBlock (pclPrev);
00270
00271
               } while (!bIsTail);
00272
          }
00273
00274
           \ensuremath{//} If we awoke any threads, re-run the scheduler
00275
          if (bReschedule)
               Thread::Yield();
00277
00278
00279
           // Update the bitmask based on any "clear" operations performed along
00280
           // the way
00281
           m_u16SetMask = u16NewMask;
```

```
00282 }
00283
00284 //-
00285 void EventFlag::Clear(uint16_t u16Mask_)
00286 {
00287
            KERNEL_ASSERT(IsInitialized());
00289
            \ensuremath{//}\xspace Just clear the bitfields in the local object.
00290
            const auto cs = CriticalGuard{};
00291
           m_u16SetMask &= ~u16Mask_;
00292 }
00293
00294 //--
00295 uint16_t EventFlag::GetMask()
00296 {
00297
            KERNEL_ASSERT(IsInitialized());
00298
00299
           // Return the presently held event flag values in this object. Ensure // we get this within a critical section to guarantee atomicity.
00300
00301
            const auto cs = CriticalGuard{};
00302
            auto u16Return = m_u16SetMask;
00303
            return u16Return;
00304 }
00305 } // namespace Mark3
00306 #endif // #if KERNEL_EVENT_FLAGS
```

# 20.41 /home/moslevin/projects/m3-repo/kernel/src/kernel.cpp File Reference

Kernel initialization and startup code.

```
#include "mark3.h"
```

#### **Namespaces**

· Mark3

### 20.41.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

### 20.42 kernel.cpp

```
00001 /*===
00002
00003
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00021 #include "mark3.h"
00022 namespace Mark3
00023 {
00024 bool
                Kernel::m bIsStarted:
00025 bool
                Kernel::m_bIsPanic;
00026 PanicFunc Kernel::m_pfPanic;
```

```
00027
00028 #if KERNEL_THREAD_CREATE_CALLOUT
00029 ThreadCreateCallout Kernel::m_pfThreadCreateCallout;
                                                               // #if KERNEL_THREAD_CREATE_CALLOUT
00030 #endif
00031 #if KERNEL THREAD EXIT CALLOUT
00032 ThreadExitCallout Kernel::m_pfThreadExitCallout;
00033 #endif
                                                          // #if KERNEL_THREAD_EXIT_CALLOUT
00034 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00035 ThreadContextCallout Kernel::m_pfThreadContextCallout;
                                                                 // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00036 #endif
00037 DebugPrintFunction Kernel::m_pfDebugPrintFunction;
00038 #if KERNEL STACK CHECK
00039 uint16_t Kernel::m_u16GuardThreshold;
00040 #endif // #if KERNEL_STACK_CHECK
00041 uint32_t Kernel::m_u32Ticks;
00042
00043 //----
00044 void Kernel::Init()
ThreadPort::Init();
00048 AutoAlloc::Init();
00049 // Initialize the global kernel data - thread-scheduler, and timer-scheduler.
00050 Scheduler::Init();
00051 TimerScheduler::Init()
00052 #if KERNEL_STACK_CHECK
          m_u16GuardThreshold = KERNEL_STACK_GUARD_DEFAULT;
00053
00054 #endif // #if KERNEL_STACK_CHECK
00055 }
00056
00057 //-
00058 void Kernel::Start()
00059 {
00060
          ThreadPort::StartThreads();
00061 }
00062
00063 //-
00064 void Kernel::CompleteStart()
00065 {
00066
          m_bIsStarted = true;
00067 }
00068
00069 //--
00070 void Kernel::Panic(uint16_t u16Cause_)
00071 {
00072
          m_bIsPanic = true;
00073
          if (nullptr != m_pfPanic) {
00074
00075
              m_pfPanic(u16Cause_);
        } else {
        while (true) {}
}
00076
00077
00078 }
00079
00080 //----
00081 void Kernel::DebugPrint(const char* szString_)
00082 {
          KERNEL_ASSERT (nullptr != szString_);
        if (nullptr != m_pfDebugPrintFunction) {
00084
           m_pfDebugPrintFunction(szString_);
00085
00086
00087 }
00088
00089 //-
00090 uint32_t Kernel::GetTicks()
00091 {
00092
          auto cs = CriticalGuard{};
00093
          return m_u32Ticks;
00094 }
00095
00096 } // namespace Mark3
```

# 20.43 /home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp File Reference

Semaphore Blocking-Object Implemenation.

```
#include "mark3.h"
```

#### **Namespaces**

• Mark3

#### 20.43.1 Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

### 20.44 ksemaphore.cpp

```
00001
00002
00003
00004
                  1.11
00005
                   1.11
00006
00007
00008
00009 -- [Mark3 Realtime Platform] --
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00022 #include "mark3.h"
00023
00024 namespace Mark3
00025 {
00026 namespace
00027 {
00028
00038
          void TimedSemaphore_Callback(Thread* pclOwner_, void* pvData_)
00039
               KERNEL_ASSERT(nullptr != pclOwner_);
00040
               KERNEL_ASSERT (nullptr != pvData_);
00041
00042
00043
               auto* pclSemaphore = static_cast<Semaphore*>(pvData_);
00044
00045
               \ensuremath{//} Indicate that the semaphore has expired on the thread
00046
               pclOwner_->SetExpired(true);
00047
00048
               // Wake up the thread that was blocked on this semaphore.
00049
               pclSemaphore->WakeMe(pclOwner_);
00050
00051
               if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority()) {
00052
                   Thread::Yield();
00053
               }
00054
00055 } // anonymous namespace
00056
00057 //-----
00058 Semaphore::~Semaphore()
00059 {
00060
           // If there are any threads waiting on this object when it goes out
          /// of scope, set a kernel panic.
if (nullptr != m_clBlockList.GetHead()) {
00061
00062
00063
               Kernel::Panic(PANIC_ACTIVE_SEMAPHORE_DESCOPED);
00064
00065 }
00066
00067 //--
00068 void Semaphore::WakeMe(Thread* pclChosenOne_)
00069 {
          KERNEL_ASSERT(pclChosenOne_);
KERNEL_ASSERT(IsInitialized());
00070
00071
00072
00073
           // Remove from the semaphore waitlist and back to its ready list.
00074
          UnBlock (pclChosenOne_);
00075 }
00076
00077 //-
00078 uint8_t Semaphore::WakeNext()
00079 {
```

```
auto* pclChosenOne = m_clBlockList.HighestWaiter();
00081
          KERNEL_ASSERT (pclChosenOne);
00082
00083
           // Remove from the semaphore waitlist and back to its ready list.
00084
          UnBlock (pclChosenOne);
00085
           // Call a task switch if higher or equal priority thread
00087
           if (pclChosenOne->GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority()) {
             return 1;
00088
00089
00090
          return 0:
00091 }
00092
00093 //--
00094 void Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)
00095 {
00096
          KERNEL ASSERT(!m clBlockList.GetHead());
00098
          // Copy the paramters into the object - set the maximum value for this
00099
          // semaphore to implement either binary or counting semaphores, and set
          // the initial count. Clear the wait list for this object. m_u16Value = u16InitVal_;
00100
00101
          m u16Value
          m_u16MaxValue = u16MaxVal_;
00102
00103
00104
          SetInitialized();
00105 }
00106
00107 //--
00108 bool Semaphore::Post()
00109 {
00110
          KERNEL_ASSERT(IsInitialized());
00111
00112
          auto bThreadWake = false;
          auto 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
00117
00118
          { // Begin critical section
00119
               const auto cs = CriticalGuard{};
00120
               \ensuremath{//} If nothing is waiting for the semaphore
00121
               if (nullptr == m clBlockList.GetHead()) {
00122
                   // Check so see if we've reached the maximum value in the semaphore
00123
                   if (m_u16Value < m_u16MaxValue) {</pre>
00124
                       // Increment the count value
00125
                       m_u16Value++;
                   } else {
   // Maximum value has been reached, bail out.
00126
00127
00128
                       bBail = true;
00129
                   }
00130
               } else {
00131
                   // Otherwise, there are threads waiting for the semaphore to be
                   // posted, so wake the next one (highest priority goes first). bThreadWake = (WakeNext() != 0u);
00132
00133
00134
00135
          } // end critical section
00136
00137
          // If we weren't able to increment the semaphore count, fail out.
00138
          if (bBail) {
              return false;
00139
00140
00141
00142
          // if bThreadWake was set, it means that a higher-priority thread was
00143
          // woken. Trigger a context switch to ensure that this thread gets
00144
           // to execute next.
00145
          if (bThreadWake)
00146
               Thread::Yield();
00147
00148
          return true;
00149 }
00150
00151 //--
00152 bool Semaphore::Pend_i(uint32_t u32WaitTimeMS_)
00153 {
00154
          KERNEL ASSERT(IsInitialized());
00155
00156
          auto clSemTimer = Timer {};
00157
          auto bUseTimer = false;
00158
           // Once again, messing with thread data - ensure
00159
00160
          // we're doing all of these operations from within a thread-safe context.
00161
00162
          { // Begin critical section
00163
               const auto cs = CriticalGuard{};
00164
               // Check to see if we need to take any action based on the semaphore count
               if (0 != m_u16Value) {
00165
```

```
// The semaphore count is non-zero, we can just decrement the count
                  // and go along our merry way.
00168
                  m_u16Value--;
              } else {
    // The semaphore count is zero - we need to block the current thread
00169
00170
                  // and wait until the semaphore is posted from elsewhere.
00171
00172
                 if (Ou != u32WaitTimeMS_) {
00173
                      g_pclCurrent->SetExpired(false);
00174
                      clSemTimer.Init();
00175
                      clSemTimer.Start(false, u32WaitTimeMS_, TimedSemaphore_Callback, this);
00176
                      bUseTimer = true;
00177
00178
                  BlockPriority(g_pclCurrent);
00179
00180
                  // Switch Threads immediately
00181
                  Thread::Yield();
00182
         } // End critical section
00183
00184
00185
          if (bUseTimer) {
00186
             clSemTimer.Stop();
00187
              return (g_pclCurrent->GetExpired() == false);
00188
00189
          return true;
00190 }
00191
00192 //--
00193 // Redirect the untimed pend API to the timed pend, with a null timeout.
00194 void Semaphore::Pend()
00195 {
00196
          Pend i(0);
00197 }
00198
00199 //--
00200 bool Semaphore::Pend(uint32_t u32WaitTimeMS_)
00201 {
00202
          return Pend i(u32WaitTimeMS);
00204
00205 //---
00206 uint16_t Semaphore::GetCount()
00207 {
          KERNEL ASSERT(IsInitialized()):
00208
00209
00210
          auto cs = CriticalGuard{};
00211
          return m_u16Value;
00212 }
00213 } // namespace Mark3
```

### 20.45 /home/moslevin/projects/m3-repo/kernel/src/II.cpp File Reference

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

```
#include "mark3.h"
```

#### **Namespaces**

Mark3

#### 20.45.1 Detailed Description

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

Definition in file II.cpp.

20.46 II.cpp 473

### 20.46 II.cpp

```
00001 /*=========
00002
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00022 #include "mark3.h"
00023
00024 namespace Mark3
00025 {
00026 //---
00027 void LinkListNode::ClearNode()
00028 {
00029
          next = nullptr;
00030
         prev = nullptr;
00031 }
00032
00033 //--
00034 void DoubleLinkList::Add(LinkListNode* node)
00035 {
00036
          KERNEL_ASSERT(nullptr != node_);
00037
          node_->prev = m_pclTail;
node_->next = nullptr;
00038
00039
00040
00041
          // If the list is empty, initilize the head
00042
          if (nullptr == m_pclHead) {
00043
              m_pclHead = node_;
00044
00045
           // Otherwise, adjust the tail's next pointer
00046
          else (
00047
              m_pclTail->next = node_;
00048
00049
00050
          // Move the tail node, and assign it to the new node just passed in
00051
          m_pclTail = node_;
00052 }
00053
00054 //-
00055 void DoubleLinkList::Remove(LinkListNode* node_)
00056 {
00057
          KERNEL_ASSERT(nullptr != node_);
00058
00059
          if (nullptr != node ->prev) {
              if (node_->prev->next != node_) {
00060
00061
                   Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00062
00063
              node_->prev->next = node_->next;
00064
          if (nullptr != node_->next) {
00065
              if (node_->next->prev != node_) {
   Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00066
00067
00068
00069
              node_->next->prev = node_->prev;
00070
          if (node_ == m_pclHead) {
00071
              m_pclHead = node_->next;
00072
00073
00074
          if (node_ == m_pclTail) {
00075
              m_pclTail = node_->prev;
00076
00077
          node_->ClearNode();
00078 }
00079
00080 //-
00081 void CircularLinkList::Add(LinkListNode* node_)
00082 {
00083
          KERNEL ASSERT (nullptr != node );
00084
00085
          if (nullptr == m_pclHead) {
00086
              // If the list is empty, initilize the nodes
00087
              m_pclHead = node_;
              m_pclTail = node_;
00088
00089
          } else {
    // Move the tail node, and assign it to the new node just passed in
00090
00091
              m_pclTail->next = node_;
00092
```

```
00094
          // Add a node to the end of the linked list.
00095
          node_->prev = m_pclTail;
          node_->next = m_pclHead;
00096
00097
00098
          m_pclTail
                           = node :
00099
          m_pclHead->prev = node_;
00100 }
00101
00102 //---
00103 void CircularLinkList::Remove(LinkListNode* node_)
00104 {
00105
          KERNEL_ASSERT(nullptr != node_);
00106
00107
          // Check to see if this is the head of the list...
          if ((node_ == m_pclHead) && (m_pclHead == m_pclTail)) {
    // Clear the head and tail pointers - nothing else left.
00108
00109
              m_pclHead = nullptr;
m_pclTail = nullptr;
00110
00111
00112
              return;
00113
00114
          // Verify that all nodes are properly connected
00115
          if ((node_->prev->next != node_) || (node_->next->prev != node_)) {
   Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00116
00117
00118
00119
          // This is a circularly linked list - no need to check for connection, // just remove the node.
00120
00121
          node_->next->prev = node_->prev;
00122
00123
          node_->prev->next = node_->next;
00124
00125
          if (node_ == m_pclHead) {
00126
              m_pclHead = m_pclHead->next;
00127
          if (node_ == m_pclTail) {
00128
          m_pclTail = m_pclTail->prev;
}
00129
00130
00131
          node_->ClearNode();
00132 }
00133
00134 //---
00135 void CircularLinkList::PivotForward()
00136 {
00137
          if (nullptr != m_pclHead) {
00138
              m_pclHead = m_pclHead->next;
00139
              m_pclTail = m_pclTail->next;
00140
          }
00141 }
00142
00143 //--
00144 void CircularLinkList::PivotBackward()
00145 {
00146
          if (nullptr != m_pclHead) {
              m_pclHead = m_pclHead->prev;
00147
              m_pclTail = m_pclTail->prev;
00148
00150 }
00151
00152 //---
00153 void CircularLinkList::InsertNodeBefore(
      LinkListNode* node_, LinkListNode* insert_)
00154 {
00155
          KERNEL_ASSERT(nullptr != node_);
00156
         KERNEL_ASSERT (nullptr != insert_);
00157
00158
          node_->next = insert_;
         node_->prev = insert_->prev;
00159
00160
00161
          if (nullptr != insert_->prev) {
00162
              insert_->prev->next = node_;
00163
00164
          insert_->prev = node_;
00165 }
00166 } // namespace Mark3
```

### 20.47 /home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp File Reference

Mutex RAII helper class.

#include "mark3.h"

20.48 lockguard.cpp 475

#### **Namespaces**

Mark3

### 20.47.1 Detailed Description

Mutex RAII helper class.

Definition in file lockguard.cpp.

### 20.48 lockguard.cpp

```
00001
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2018 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00020 #include "mark3.h"
00021
00022 namespace Mark3
00023 {
00024 //----
00025 LockGuard::LockGuard(Mutex* pclMutex_)
00026 : m_bIsAcquired { true }
00027
         , m_pclMutex { pclMutex_ }
00028 {
00029
         KERNEL_ASSERT(nullptr != m_pclMutex);
00030
          m_pclMutex->Claim();
00031 }
00032
00033 //-----
00034 LockGuard::LockGuard(Mutex* pclMutex_, uint32_t u32TimeoutMs_)
00035
         : m_pclMutex { pclMutex_ }
00036 {
00037
         KERNEL_ASSERT(nullptr != pclMutex_);
00038
         m_bIsAcquired = m_pclMutex->Claim(u32TimeoutMs_);
00039 }
00040
00041 //---
00042 LockGuard::~LockGuard()
00043 {
00044
          if (m_bIsAcquired) {
00045
             m_pclMutex->Release();
         }
00046
00047 }
00048
00049 } // namespace Mark3
```

### 20.49 /home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp File Reference

Mailbox + Envelope IPC mechanism.

```
#include "mark3.h"
```

### **Namespaces**

• Mark3

#### 20.49.1 Detailed Description

Mailbox + Envelope IPC mechanism.

Definition in file mailbox.cpp.

### 20.50 mailbox.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00021 #include "mark3.h"
00022 namespace Mark3
00023 {
00024 //--
00025 Mailbox::~Mailbox()
00026 {
00027
          // If the mailbox isn't empty on destruction, kernel panic.
00028
          if (m_u16Free != m_u16Count)
00029
              Kernel::Panic(PANIC_ACTIVE_MAILBOX_DESCOPED);
00030
00031 }
00032
00033 //-
00034 void Mailbox::Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)
00035 {
00036
          KERNEL_ASSERT (u16BufferSize_);
00037
          KERNEL_ASSERT (u16ElementSize_);
          KERNEL_ASSERT(nullptr != pvBuffer_);
00038
00039
00040
                            = pvBuffer_;
00041
          m_u16ElementSize = u16ElementSize_;
00042
          m_u16Count = (u16BufferSize_ / u16ElementSize_);
m_u16Free = m_u16Count;
00043
00044
00045
00046
          m_u16Head = 0;
00047
          m_u16Tail = 0;
00048
00049
          // We use the counting semaphore to implement blocking - with one element
00050
          // in the mailbox corresponding to a post/pend operation in the semaphore.
00051
          m_clRecvSem.Init(0, m_u16Free);
00052
00053
          // Binary semaphore is used to track any threads that are blocked on a // "send" due to lack of free slots.
00054
          m_clSendSem.Init(0, 1);
00055
00056 }
00057
00058 //
00059 Mailbox* Mailbox::Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_)
00060 {
00061
          KERNEL_ASSERT(u16BufferSize_);
00062
          KERNEL_ASSERT (u16ElementSize_);
00063
00064
                         = AutoAlloc::NewObject<Mailbox, AutoAllocType::MailBox>();
          auto* pclNew
          auto* pvBuffer = AutoAlloc::NewRawData(u16BufferSize_);
00065
00066
00067
          KERNEL_ASSERT(nullptr != pclNew);
          KERNEL_ASSERT(nullptr != pvBuffer);
00068
00069
00070
          if (!pclNew) {
00071
              return nullptr;
00072
00073
00074
              AutoAlloc::DestroyObject<Mailbox, AutoAllocType::MailBox>(pclNew);
00075
              return nullptr;
00076
00077
          pclNew->Init(pvBuffer, u16BufferSize_, u16ElementSize_);
```

20.50 mailbox.cpp 477

```
00079
         return pclNew;
00080 }
00081
00082 //----
00083 void Mailbox::Receive(void* pvData )
00084 {
          KERNEL_ASSERT(nullptr != pvData_);
00086
          Receive_i (pvData_, false, 0);
00087 }
00088
00089 //----
00090 bool Mailbox::Receive(void* pvData_, uint32_t u32TimeoutMS_)
00091 {
00092
         KERNEL_ASSERT(nullptr != pvData_);
00093
         return Receive_i (pvData_, false, u32TimeoutMS_);
00094 }
00095
00096 //--
00097 void Mailbox::ReceiveTail(void* pvData_)
00098 {
00099
          KERNEL_ASSERT(nullptr != pvData_);
00100
         Receive_i (pvData_, true, 0);
00101 }
00102
00103 //-
00104 bool Mailbox::ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_)
00105 {
00106
         KERNEL_ASSERT (nullptr != pvData_);
00107
         return Receive_i (pvData_, true, u32TimeoutMS_);
00108 }
00109
00110 //-
00111 bool Mailbox::Send(void* pvData_)
00112 {
00113
         KERNEL_ASSERT(nullptr != pvData_);
00114
         return Send_i(pvData_, false, 0);
00115 }
00116
00117 //-
00118 bool Mailbox::SendTail(void* pvData_)
00119 {
00120
         KERNEL_ASSERT (nullptr != pvData_);
00121
         return Send_i (pvData_, true, 0);
00122 }
00123
00124 //-
00125 bool Mailbox::Send(void* pvData_, uint32_t u32TimeoutMS_)
00126 {
00127
         KERNEL_ASSERT (nullptr != pvData_);
         return Send_i (pvData_, false, u32TimeoutMS_);
00128
00129 }
00130
00131 //--
00132 bool Mailbox::SendTail(void* pvData_, uint32_t u32TimeoutMS_)
00133 {
00134
         KERNEL ASSERT (nullptr != pvData );
00135
          return Send_i(pvData_, true, u32TimeoutMS_);
00136 }
00137
00138 //--
00139 bool Mailbox::Send_i(const void* pvData_, bool bTail_, uint32_t u32TimeoutMS_)
00140 {
00141
          KERNEL_ASSERT(nullptr != pvData_);
00142
00143
          void* pvDst = nullptr;
00144
                          = false;
00145
         auto bRet
         auto bSchedState = Scheduler::SetScheduler(false);
00146
                        = false;
= false;
00147
         auto bBlock
00148
         auto bDone
00149
00150
         while (!bDone) {
00151
             // Try to claim a slot first before resorting to blocking.
00152
              if (bBlock) {
00153
                  bDone = true;
00154
                  Scheduler::SetScheduler(bSchedState);
                  m_clSendSem.Pend(u32TimeoutMS_);
00155
00156
                  Scheduler::SetScheduler(false);
00157
00158
              \{\ //\ {\tt Begin\ critical\ section}
00159
00160
                 const auto cs = CriticalGuard{};
00161
                  // Ensure we have a free slot before we attempt to write data
00162
                  if (0u != m_u16Free) {
00163
                      m_u16Free--;
00164
00165
                      if (bTail ) {
```

```
pvDst = GetTailPointer();
                          MoveTailBackward();
00168
                      } else {
00169
                          MoveHeadForward();
00170
                          pvDst = GetHeadPointer();
00171
00172
                      bRet = true;
00173
                      bDone = true;
00174
                  } else if (Ou != u32TimeoutMS_) {
00175
                      bBlock = true;
00176
                  } else {
00177
                      bDone = true;
00178
00179
              } // End critical section
00180
00181
          // Copy data to the claimed slot, and post the counting semaphore
00182
00183
          if (bRet) {
00184
              CopyData(pvData_, pvDst, m_u16ElementSize);
00185
00186
00187
          Scheduler::SetScheduler(bSchedState);
00188
00189
          if (bRet) {
00190
             m_clRecvSem.Post();
00191
00192
00193
          return bRet;
00194 }
00195
00196 //-
00197 bool Mailbox::Receive_i(void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_)
00198 {
00199
          KERNEL_ASSERT (nullptr != pvData_);
00200
          auto* pvSrc = (const void*){};
00201
00202
          if (!m clRecvSem.Pend(u32WaitTimeMS)) {
              // Failed to get the notification from the counting semaphore in the
00204
              // time allotted. Bail.
00205
              return false;
00206
00207
         // Disable the scheduler while we do this -- this ensures we don't have
00208
          // multiple concurrent readers off the same queue, which could be problematic
00209
00210
          // if multiple writes occur during reads, etc.
00211
          auto bSchedState = Scheduler::SetScheduler(false);
00212
00213
          // Update the head/tail indexes, and get the associated data pointer for
         // the read operation.
00214
00215
00216
          { // Begin critical section
00217
              const auto cs = CriticalGuard{};
00218
              m_u16Free++;
              if (bTail_) {
    MoveTailForward();
00219
00220
00221
                  pvSrc = GetTailPointer();
              } else {
00223
                  pvSrc = GetHeadPointer();
00224
                  MoveHeadBackward();
00225
         } // end critical section
00226
00227
00228
          KERNEL_ASSERT (pvSrc);
00229
          CopyData(pvSrc, pvData_, m_u16ElementSize);
00230
00231
          Scheduler::SetScheduler(bSchedState);
00232
00233
          // Unblock a thread waiting for a free slot to send to
00234
          m_clSendSem.Post();
00235
00236
          return true;
00237
00238 } // namespace Mark3
```

# 20.51 /home/moslevin/projects/m3-repo/kernel/src/message.cpp File Reference

Inter-thread communications via message passing.

```
#include "mark3.h"
```

20.52 message.cpp 479

#### **Namespaces**

Mark3

#### 20.51.1 Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

### 20.52 message.cpp

```
00001 /
00002
00003
00004 |
                  1 - 11
00005 1
                  1.11
00006
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00022 #include "mark3.h"
00023 namespace Mark3
00024 {
00025 //--
00026 void MessagePool::Init()
00027 {
00028
          m_clList.Init();
00029 }
00030
00031 //--
00032 void MessagePool::Push(Message* pclMessage_)
00033 {
00034
          KERNEL_ASSERT (pclMessage_);
00035
00036
          const auto cs = CriticalGuard{};
00037
          m_clList.Add(pclMessage_);
00038 }
00039
00040 //--
00041 Message* MessagePool::Pop()
00042 {
          const auto cs = CriticalGuard{};
auto* pclRet = m_clList.GetHead();
if (nullptr != pclRet) {
00043
00044
00045
00046
              m_clList.Remove(pclRet);
00047
00048
          return pclRet;
00049 }
00050
00051 //-
00052 Message* MessagePool::GetHead()
00053 {
00054
          return m_clList.GetHead();
00055 }
00056
00057 //-
00058 void MessageQueue::Init()
00059 {
00060
          m_clSemaphore.Init(0, 255);
00061 }
00062
00063 //--
00064 Message* MessageQueue::Receive()
00065 {
00066
          return Receive_i(0);
00067 }
00068
00069 //--
00070 Message* MessageQueue::Receive(uint32_t u32TimeWaitMS_)
00071 {
```

```
return Receive_i(u32TimeWaitMS_);
00073 }
00074
00075 //-----
00076 Message* MessageQueue::Receive_i(uint32_t u32TimeWaitMS_)
00077 {
          // Block the current thread on the counting semaphore
00079
         if (!m_clSemaphore.Pend(u32TimeWaitMS_)) {
            return nullptr;
08000
00081
00082
00083
        const auto cs = CriticalGuard{};
         // Pop the head of the message queue and return it
auto* pclRet = m_clLinkList.GetHead();
00084
00085
00086
         m_clLinkList.Remove(pclRet);
00087
00088
         return pclRet;
00089 }
00090
00091 //-
00092 void MessageQueue::Send(Message* pclSrc_)
00093 {
00094
          KERNEL_ASSERT (pclSrc_);
00095
00096
         CriticalSection::Enter();
00097
00098
         // Add the message to the head of the linked list
00099
         m_clLinkList.Add(pclSrc_);
00100
00101
          CriticalSection::Exit();
00102
00103
         // Post the semaphore, waking the blocking thread for the queue.
00104
          m_clSemaphore.Post();
00105 }
00106
00107 //---
00108 uint16_t MessageQueue::GetCount()
00110
          return m_clSemaphore.GetCount();
00111 }
00112 } // namespace Mark3
```

# 20.53 /home/moslevin/projects/m3-repo/kernel/src/mutex.cpp File Reference

Mutual-exclusion object.

```
#include "mark3.h"
```

### Namespaces

• Mark3

### 20.53.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

20.54 mutex.cpp 481

### 20.54 mutex.cpp

```
00001 /*=======
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00020 #include "mark3.h"
00021 namespace Mark3
00022 {
00023 namespace
00024 {
00025
00035
           void TimedMutex_Callback(Thread* pclOwner_, void* pvData_)
00036
               KERNEL_ASSERT(nullptr != pclOwner_);
KERNEL_ASSERT(nullptr != pvData_);
00037
00038
00039
00040
               auto* pclMutex = static_cast<Mutex*>(pvData_);
00041
00042
               // Indicate that the semaphore has expired on the thread
00043
               pclOwner_->SetExpired(true);
00044
00045
               \ensuremath{//} Wake up the thread that was blocked on this semaphore.
00046
               pclMutex->WakeMe(pclOwner_);
00047
               if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
00048
      GetCurPriority()) {
00049
                  Thread::Yield();
00050
               }
00051
00052 } // anonymous namespace
00053
00055 Mutex::~Mutex()
00056 {
          // If there are any threads waiting on this object when it goes out // of scope, set a kernel panic. if (nullptr != m_clBlockList.GetHead()) {
00057
00058
00059
00060
               Kernel::Panic(PANIC_ACTIVE_MUTEX_DESCOPED);
00061
00062 }
00063
00064 //---
00065 void Mutex::WakeMe(Thread* pclOwner_)
00066 {
00067
           KERNEL_ASSERT (nullptr != pclOwner_);
00068
           // Remove from the semaphore waitlist and back to its ready list.
00069
          UnBlock(pclOwner_);
00070 }
00071
00072 //-
00073 uint8_t Mutex::WakeNext()
00074 {
00075
           // Get the highest priority waiter thread
          auto* pclChosenOne = m_clBlockList.HighestWaiter();
KERNEL_ASSERT(pclChosenOne);
00076
00077
00078
00079
            // Unblock the thread
00080
           UnBlock (pclChosenOne);
00081
00082
           // The chosen one now owns the mutex
00083
           m_pclOwner = pclChosenOne;
00084
           // Signal a context switch if it's a greater than or equal to the current priority
00085
00086
           if (pclChosenOne->GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority()) {
00087
               return 1:
00088
00089
           return 0;
00090 }
00091
00092 //--
00093 void Mutex::Init(bool bRecursive_)
00094 {
00095
            // Cannot re-init a mutex which has threads blocked on it
00096
           KERNEL_ASSERT(!m_clBlockList.GetHead());
00097
```

```
// Reset the data in the mutex
          00099
00100
00101
                                  // Reset recurse count
          m_u8Recurse = 0;
00102
          m_bRecursive = bRecursive_;
00103
00104
          SetInitialized();
00105 }
00106
00107 //---
00108 bool Mutex::Claim_i (uint32_t u32WaitTimeMS_)
00109 {
00110
          KERNEL_ASSERT(IsInitialized());
00111
00112
          auto clTimer = Timer {};
00113
          auto bUseTimer = false;
00114
00115
          // Disable the scheduler while claiming the mutex - we're dealing with all
          // sorts of private thread data, can't have a thread switch while messing
00116
00117
          // with internal data structures.
00118
          Scheduler::SetScheduler(false);
00119
         // Check to see if the mutex is claimed or not
if (false != m_bReady) {
00120
00121
00122
              // Mutex isn't claimed, claim it.
00123
              m_bReady = false;
00124
              m_u8Recurse = 0;
              m_uMaxPri = g_pclCurrent->GetPriority();
m_pclOwner = g_pclCurrent;
00125
00126
00127
00128
              Scheduler::SetScheduler(true);
00129
              return true;
00130
00131
00132
          // If the mutex is already claimed, check to see if this is the owner thread,
         // since we allow the mutex to be claimed recursively.
if (m_bRecursive && (g_pclCurrent == m_pclOwner)) {
00133
00134
              // Ensure that we haven't exceeded the maximum recursive-lock count
00136
              KERNEL_ASSERT((m_u8Recurse < 255));</pre>
00137
              m_u8Recurse++;
00138
00139
              // Increment the lock count and bail
              Scheduler::SetScheduler(true);
00140
00141
              return true;
00142
         }
00143
00144
          // The mutex is claimed already - we have to block now. Move the
          // current thread to the list of threads waiting on the mutex.
00145
          if (Ou != u32WaitTimeMS_) {
00146
00147
              q_pclCurrent->SetExpired(false);
00148
              clTimer.Init();
00149
              clTimer.Start(false, u32WaitTimeMS_, TimedMutex_Callback, this);
00150
              bUseTimer = true;
00151
00152
          BlockPriority(g_pclCurrent);
00153
          // Check if priority inheritence is necessary. We do this in order
00155
          // to ensure that we don't end up with priority inversions in case
00156
          // multiple threads are waiting on the same resource.
          if (m_uMaxPri <= g_pclCurrent->GetPriority()) {
    m_uMaxPri = g_pclCurrent->GetPriority();
00157
00158
00159
00160
              auto* pclTemp = m_clBlockList.GetHead();
00161
              while (nullptr != pclTemp) {
00162
                  pclTemp->InheritPriority(m_uMaxPri);
00163
                  if (m_clBlockList.GetTail() == pclTemp) {
00164
                       break;
00165
00166
                  pclTemp = pclTemp->GetNext();
00167
00168
              m_pclOwner->InheritPriority(m_uMaxPri);
00169
          }
00170
00171
          // Done with thread data -reenable the scheduler
00172
          Scheduler::SetScheduler(true);
00173
00174
          // Switch threads if this thread acquired the mutex
00175
          Thread::Yield();
00176
00177
          if (bUseTimer) {
00178
              clTimer.Stop();
00179
              return (false == g_pclCurrent->GetExpired());
00180
00181
          return true;
00182 }
00183
00184 //----
```

```
00185 void Mutex::Claim(void)
00187
          Claim_i(0);
00188 }
00189
00190 //-
00191 bool Mutex::Claim(uint32_t u32WaitTimeMS_)
00192 {
00193
          return Claim_i (u32WaitTimeMS_);
00194 }
00195
00196 //--
00197 void Mutex::Release()
00198 {
00199
          KERNEL_ASSERT(IsInitialized());
00200
00201
          auto bSchedule = false;
00202
         // Disable the scheduler while we deal with internal data structures.
00204
          Scheduler::SetScheduler(false);
00205
00206
          // This thread had better be the one that owns the mutex currently...
         KERNEL_ASSERT((g_pclCurrent == m_pclOwner));
00207
00208
00209
          // If the owner had claimed the lock multiple times, decrease the lock
00210
          // count and return immediately.
00211
          if (m_bRecursive && (0u != m_u8Recurse)) {
00212
             m_u8Recurse--;
00213
              Scheduler::SetScheduler(true);
00214
              return;
00215
         }
00216
        // Restore the thread's original priority
if (g_pclCurrent->GetCurPriority() != g_pclCurrent->
00217
00218
     GetPriority()) {
00219
              q_pclCurrent->SetPriority(q_pclCurrent->
     GetPriority());
00220
00221
               // In this case, we want to reschedule
00222
              bSchedule = true;
00223
          }
00224
        // No threads are waiting on this semaphore?
00225
00226
         if (nullptr == m_clBlockList.GetHead()) {
          // Re-initialize the mutex to its default values
00227
             m_bReady = true;
m_uMaxPri = 0;
00228
00229
              m_pclOwner = nullptr;
00230
        } else {
   // Wake the highest priority Thread pending on the mutex
   if (Ou != WakeNext()) {
00231
00232
              if (Ou != WakeNext()) {
00234
                   // Switch threads if it's higher or equal priority than the current thread
00235
                  bSchedule = true;
00236
            }
00237
         }
00238
        // Must enable the scheduler again in order to switch threads.
Scheduler::SetScheduler(true);
00240
00241
         if (bSchedule) {
00242
               // Switch threads if a higher-priority thread was woken
00243
              Thread::Yield();
00244
00245 }
00246 } // namespace Mark3
```

# 20.55 /home/moslevin/projects/m3-repo/kernel/src/notify.cpp File Reference

Lightweight thread notification - blocking object.

```
#include "mark3.h"
```

### **Namespaces**

• Mark3

### 20.55.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.cpp.

### 20.56 notify.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009
      --[Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00021 #include "mark3.h"
00022 namespace Mark3
00023 {
00024 namespace
00025 {
00026
00027
           void TimedNotify_Callback(Thread* pclOwner_, void* pvData_)
00028
00029
                KERNEL_ASSERT (nullptr != pclOwner_);
00030
               KERNEL_ASSERT(nullptr != pvData_);
00031
00032
               auto* pclNotify = static cast<Notify*>(pvData );
00033
00034
               \ensuremath{//} Indicate that the semaphore has expired on the thread
00035
               pclOwner_->SetExpired(true);
00036
               \ensuremath{//} Wake up the thread that was blocked on this semaphore.
00037
00038
               pclNotify->WakeMe(pclOwner_);
00039
00040
               if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority()) {
00041
                   Thread::Yield();
00042
00043
00044 } // anonymous namespace
00045
00046 //---
00047 Notify::~Notify()
00048 {
           // If there are any threads waiting on this object when it goes out
// of scope, set a kernel panic.
if (nullptr != m_clBlockList.GetHead()) {
00049
00050
00051
00052
               Kernel::Panic(PANIC_ACTIVE_NOTIFY_DESCOPED);
00053
00054 }
00055
00056 //---
00057 void Notify::Init(void)
00058 {
00059
           KERNEL_ASSERT(!m_clBlockList.GetHead());
00060
           SetInitialized();
00061
00062
           m_bPending = false;
00063 }
00064
00065 //--
00066 void Notify::Signal(void)
00067 {
00068
           KERNEL ASSERT(IsInitialized());
00069
00070
           auto bReschedule = false;
00071
00072
           { // Begin critical section
               const auto cs = CriticalGuard{};
auto* pclCurrent = m_clBlockList.GetHead();
00073
00074
               if (nullptr == pclCurrent) {
    m_bPending = true;
00075
00076
00077
               } else {
```

20.56 notify.cpp 485

```
00078
                  while (nullptr != pclCurrent) {
00079
                      UnBlock(pclCurrent);
00080
                       if (!bReschedule && (pclCurrent->GetCurPriority() >=
     Scheduler::GetCurrentThread() ->GetCurPriority())) {
00081
                          bReschedule = true;
00082
00083
                      pclCurrent = m_clBlockList.GetHead();
00084
00085
                  m_bPending = false;
00086
          } // end critical section
00087
00088
00089
          if (bReschedule) {
              Thread::Yield();
00090
00091
00092 }
00093
00094 //--
00095 void Notify::Wait(bool* pbFlag_)
00096 {
00097
          KERNEL_ASSERT(nullptr != pbFlag_);
00098
          KERNEL_ASSERT(IsInitialized());
00099
00100
          auto bEarlyExit = false;
00101
          { // Begin critical section
00102
              const auto cs = CriticalGuard{};
00103
              if (!m_bPending) {
00104
                  Block(g_pclCurrent);
                  if (nullptr != pbFlag_) {
   *pbFlag_ = false;
00105
00106
00107
00108
              } else {
00109
                 m_bPending = false;
00110
                  bEarlyExit = true;
00111
          } // End critical section
00112
00113
00114
          if (bEarlyExit) {
00115
              return;
00116
00117
00118
          Thread::Yield();
          if (nullptr != pbFlag_) {
   *pbFlag_ = true;
00119
00120
00121
00122 }
00123
00124 //---
00125 bool Notify::Wait(uint32_t u32WaitTimeMS_, bool* pbFlag_)
00126 {
          KERNEL_ASSERT(nullptr != pbFlag_);
00127
00128
          KERNEL_ASSERT(IsInitialized());
00129
00130
          auto bUseTimer
                              = false;
                            = false;
00131
          auto bEarlvExit
          auto clNotifyTimer = Timer {};
00132
00134
          { // Begin critical section
00135
              const auto cs = CriticalGuard{};
00136
              if (!m_bPending) {
                   if (Ou != u32WaitTimeMS_) {
00137
                      bUseTimer = true;
00138
00139
                      g_pclCurrent->SetExpired(false);
00140
00141
                      clNotifyTimer.Init();
00142
                       clNotifyTimer.Start(false, u32WaitTimeMS_, TimedNotify_Callback, this);
00143
                  }
00144
00145
                  Block(g_pclCurrent);
00146
00147
                   if (nullptr != pbFlag_) {
00148
                       *pbFlag_ = false;
00149
00150
              } else {
                 m_bPending = false;
bEarlyExit = true;
00151
00152
00153
00154
          } // end critical section
00155
          if (bEarlyExit) {
00156
00157
              return true;
00158
00159
00160
          Thread::Yield();
00161
          if (bUseTimer) {
00162
00163
              clNotifyTimer.Stop();
```

```
return (g_pclCurrent->GetExpired() == false);
00165
00166
          if (nullptr != pbFlag_) {
   *pbFlag_ = true;
00167
00168
00169
00170
00171
00172 }
00173
00174 //----
00175 void Notify::WakeMe(Thread* pclChosenOne_)
00177
          KERNEL_ASSERT (nullptr != pclChosenOne_);
00178
          KERNEL_ASSERT(IsInitialized());
00179
00180
          UnBlock (pclChosenOne_);
00181 }
00182 } // namespace Mark3
```

# 20.57 /home/moslevin/projects/m3-repo/kernel/src/profile.cpp File Reference

Code profiling utilities.

```
#include "mark3.h"
```

#### **Namespaces**

· Mark3

### 20.57.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

# 20.58 profile.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00021 #include "mark3.h"
00022 namespace Mark3
00023 {
00024 //--
00025 void ProfileTimer::Init()
00026 {
00027
          m_u32StartTicks = 0;
          m_u32Cumulative = 0;
m_u16Iterations = 0;
00028
00029
                           = false;
00030
          m bActive
00031 }
00032
```

```
00034 void ProfileTimer::Start()
   00035 {
   00036
                            if (!m_bActive) {
                           { // Begin critical section
00037
00038
00039
00040
00041
                                        const auto cs = CriticalGuard{};
                                                    m_u32StartTicks = Kernel::GetTicks();
                                 } // End Critical Section
                                   m_bActive = true;
   00044
   00045 //---
   00046 void ProfileTimer::Stop()
   00047 {
   00048
                             if (m_bActive) {
                          uint32_t u32Final; { // Begin critical section
   00049
   00050
                                                 const auto cs = CriticalGuard{};
  00052
00053
                                                u32Final = Kernel::GetTicks();
// Compute total for current iteration...
   } // End critical section
  00057
00058
00059 }
                                     m_bActive = false;
   00060 }
   00061
   00062 //-
   00063 uint32_t ProfileTimer::GetAverage()
   00064 {
   comparison of the control of th
   00065
                              if (Ou != m_u16Iterations) {
                             return 0;
   00069 }
   00070
   00071 //----
   00072 uint32_t ProfileTimer::GetCurrent()
   00073 {
                             if (m_bActive) {
                          uint32_t u32Current;
{ // Begin critical section
   00076
                                        const auto cs = CriticalGuard{};
   00077
                            u32Current = Kernel::(
} // End critical section
                                                   u32Current = Kernel::GetTicks() - m_u32StartTicks;
   00078
   00079
   08000
                                       return u32Current;
                         return m_u32CurrentIteration;
   00082
   00083 }
  00084 } // namespace Mark3
```

## 20.59 /home/moslevin/projects/m3-repo/kernel/src/public/atomic.h File Reference

#### Basic Atomic Operations.

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

#### **Namespaces**

- Mark3
- · Mark3::Atomic

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

#### **Functions**

```
    template<typename T >
        T Mark3::Atomic::Set (T *pSource_, T val_)
```

Set Set a variable to a given value in an uninterruptable operation.

• template<typename T >

```
T Mark3::Atomic::Add (T *pSource_, T val_)
```

Add Add a value to a variable in an uninterruptable operation.

• template<typename T >

```
T Mark3::Atomic::Sub (T *pSource , T val )
```

Sub Subtract a value from a variable in an uninterruptable operation.

bool Mark3::Atomic::TestAndSet (bool \*pbLock)

TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

#### 20.59.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

### 20.60 atomic.h

```
00001 /*----
00002
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform] ---
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00021 #pragma once
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025 #include "ithreadport.h"
00026 #include "kerneldebug.h"
00027
00028 namespace Mark3
00029 {
00038 namespace Atomic
00039 {
00047
          template <typename T> T Set(T* pSource_, T val_)
00048
00049
             KERNEL_ASSERT (nullptr != pSource_);
00050
00051
             const auto cs = CriticalGuard{};
             00052
                           = *pSource_;
00053
00054
             return ret;
00055
         }
00056
00064
          template <typename T> T Add(T* pSource_, T val_)
00065
             KERNEL_ASSERT (nullptr != pSource_);
00066
00067
00068
             const auto cs = CriticalGuard{};
00069
             auto ret = *pSource_;
00070
              *pSource_ += val_;
00071
00072
             return ret;
          }
00073
00081
          template <typename T> T Sub(T* pSource_, T val_)
00082
```

# 20.61 /home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h File Reference

Automatic memory allocation for kernel objects.

```
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "mark3cfg.h"
```

#### Classes

· class Mark3::AutoAlloc

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

### **Namespaces**

Mark3

# **Typedefs**

- using Mark3::AutoAllocAllocator\_t = void \*(\*)(AutoAllocType eType\_, size\_t sSize\_)
- using Mark3::AutoAllocFree\_t = void(\*)(AutoAllocType eType\_, void \*pvObj\_)

## **Enumerations**

```
    enum Mark3::AutoAllocType:: uint8_t {
        Mark3::AutoAllocType::EventFlag, Mark3::AutoAllocType::MailBox, Mark3::AutoAllocType::Message,
        Mark3::AutoAllocType::MessagePool,
        Mark3::AutoAllocType::MessageQueue, Mark3::AutoAllocType::Mutex, Mark3::AutoAllocType::Notify,
        Mark3::AutoAllocType::Semaphore,
        Mark3::AutoAllocType::Thread, Mark3::AutoAllocType::Timer, Mark3::AutoAllocType::ConditionVariable,
        Mark3::AutoAllocType::ReaderWriterLock,
        Mark3::AutoAllocType::User, Mark3::AutoAllocType::Raw = 0xFF }
```

# 20.61.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.h.

# 20.62 autoalloc.h

```
00001 /*=
00002
00003
00004
00005
00006
00007
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =========
00020 #pragma once
00021
00022 #include <stddef.h>
00023 #include <stdint.h>
00024 #include <stdbool.h>
00025 #include "mark3cfg.h"
00026
00027 namespace Mark3
00028 {
00029 //----
00030 // Define function pointer types used for interfacing with an external heap.
00031 //----
00032 enum class AutoAllocType : uint8_t {
00033
         //-- Kernel object types
00034
         EventFlag,
00035
         MailBox,
00036
         Message,
00037
         MessagePool,
00038
         MessageOueue.
00039
          Mutex,
00040
         Notify,
00041
          Semaphore,
00042
          Thread,
00043
         Timer,
ConditionVariable,
00044
00045
         ReaderWriterLock,
00046
          //-- Allow for users to define their own object types beginning with AutoAllocType_t::User
00047
         User,
00048
00049
         Raw = 0xFF
00050 };
00051
00053 using AutoAllocatlocator_t = void* (*) (AutoAllocType eType_, size_t sSize_
00054 using AutoAllocFree_t
                                = void (*) (AutoAllocType eType_, void* pvObj_);
00055
00056 /
00057 // Forward declaration of kernel objects that can be auotomatically allocated.
00058 class EventFlag;
00059 class Mailbox;
00060 class Message;
00061 class MessagePool;
00062 class MessageQueue;
00063 class Mutex;
00064 class Notify;
00065 class Semaphore;
00066 class Thread;
00067 class Timer;
00068 class ReaderWriterLock;
00069 class ConditionVariable;
00070
00082 class AutoAlloc
00083 {
00084 public:
00090
         static void Init (void);
00091
00098
         static void SetAllocatorFunctions (AutoAllocator_t pfAllocator_,
```

```
AutoAllocFree_t pfFree_);
00099
00103
          template <typename T, AutoAllocType e> static T* NewObject()
00104
00105
              auto* pvObj = Allocate(e, sizeof(T));
00106
             <u>if</u> (pv0bj) {
00107
                 return new (pvObj) T();
00108
00109
             return 0;
00110
         }
00111
00116
         template <typename T, AutoAllocType e> static void DestroyObject(T* pObj_)
00117
00118
             pObj_->~T();
00119
             Free(e, pObj_);
00120
00121
00128
         static void* NewUserTypeAllocation(uint8_t eUserType_);
00129
00136
         static void DestroyUserTypeAllocation(uint8_t eUserType_, void* pvObj_);
00137
00144
         static void* NewRawData(size_t sSize_);
00145
00151
         static void DestroyRawData(void* pvData );
00152
00153 private:
00154
         static void* Allocate(AutoAllocType eType_, size_t sSize_);
00155
         static void Free(AutoAllocType eType_, void* pvObj_);
00156
         static AutoAllocAllocator_t m_pfAllocator;
00157
00158
         static AutoAllocFree t
                                     m_pfFree;
00159 };
00160 } // namespace Mark3
```

# 20.63 /home/moslevin/projects/m3-repo/kernel/src/public/blocking.h File Reference

Blocking object base class declarations.

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

#### **Classes**

class Mark3::BlockingObject

The BlockingObject class. 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.

#### **Namespaces**

• Mark3

## 20.63.1 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 manipu32ate 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.

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 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.

# 20.64 blocking.h

```
00001 /
00003
00004
00005
00006
00007
00008
00009 -
       -[Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00046 #pragma once
00047
00048 #include "kerneltypes.h"
00049 #include "mark3cfg.h"
00050
00051 #include "11.h"
00052 #include "threadlist.h"
00053
00054 namespace Mark3
00055
00056 class Thread;
00057
00058 //-
00065 class BlockingObject
00066 {
00067 public:
00068
         BlockingObject() { m_u8Initialized =
     m_uBlockingInvalidCookie; }
00069
          ~BlockingObject() { m u8Initialized =
     m_uBlockingInvalidCookie; }
00070
00071 protected:
00091
          void Block(Thread* pclThread_);
00092
00100
          void BlockPriority(Thread* pclThread);
00101
00112
          void UnBlock(Thread* pclThread_);
```

```
00113
          void SetInitialized(void) { m_u8Initialized =
00117
      m_uBlockingInitCookie; }
00118
00123
          bool IsInitialized(void) { return (m_u8Initialized ==
     m_uBlockingInitCookie); }
00124
00125
          // Cookies used to determine whether or not an object has been initialized
00126
          static constexpr auto m_uBlockingInvalidCookie = uint8_t { 0x3C };
                                                        = uint8_t { 0xC3 };
00127
          static constexpr auto m_uBlockingInitCookie
00128
          ThreadList m clBlockList:
00133
00134
00139
          uint8_t m_u8Initialized;
00140 };
00141 } // namespace Mark3
```

# 20.65 /home/moslevin/projects/m3-repo/kernel/src/public/colist.h File Reference

CoRoutine List structure implementation.

```
#include "mark3cfg.h"
#include "coroutine.h"
```

### Classes

· class Mark3::CoList

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

### **Namespaces**

• Mark3

### 20.65.1 Detailed Description

CoRoutine List structure implementation.

Definition in file colist.h.

## 20.66 colist.h

```
00021 #include "mark3cfg.h"
00022 #include "coroutine.h"
00023
00024 namespace Mark3 {
00025 class Coroutine;
00035 class CoList : public TypedCircularLinkList<Coroutine>
00036 {
00037 public:
00038
00045
          void SetPrioMap(CoPrioMap* pclPrioMap_);
00046
00054
          void SetPriority(PORT_PRIO_TYPE uPriority_);
00055
00062
          void Add(Coroutine* pclCoroutine_);
00063
00070
          void Remove(Coroutine* pclCoroutine_);
00072 private:
00073
          CoPrioMap* m_pclPrioMap;
00074
          uint8_t m_uPriority;
00075 };
00076 } // namespace Mark3
```

# 20.67 /home/moslevin/projects/m3-repo/kernel/src/public/condvar.h File Reference

Condition Variable implementation.

```
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "mutex.h"
#include <stddef.h>
```

## Classes

class Mark3::ConditionVariable

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

### **Namespaces**

· Mark3

#### 20.67.1 Detailed Description

Condition Variable implementation.

Definition in file condvar.h.

20.68 condvar.h 495

# 20.68 condvar.h

```
00001 /*======
00002
00003
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ======
00019 #pragma once
00020
00021 #include "mark3cfg.h"
00022 #include "ksemaphore.h"
00023 #include "mutex.h"
00024
00025 #include <stddef.h>
00026
00027 namespace Mark3
00028 {
00039 class ConditionVariable
00040 {
00041 public:
         void* operator new(size_t sz, void* pv) { return reinterpret_cast
00042
     ConditionVariable*>(pv); }
00043
00049
          void Init();
00050
00057
          void Wait (Mutex* pclMutex_);
00058
00067
         bool Wait (Mutex* pclMutex_, uint32_t u32WaitTimeMS_);
00068
00073
          void Signal();
00074
00079
          void Broadcast();
08000
00081 private:
00082
                   m clMutex;
          Mutex
00083
          Semaphore m_clSemaphore;
00084
          uint8_t m_u8Waiters;
00085 };
00086 } // namespace Mark3
```

# 20.69 /home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h File Reference

#### CoRoutine implementation.

```
#include "mark3cfg.h"
#include "ll.h"
#include "priomapl1.h"
#include "priomapl2.h"
```

### **Classes**

· class Mark3::Coroutine

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

## **Namespaces**

• Mark3

# **Typedefs**

- using Mark3::CoPrioMap = PriorityMapL1< PORT PRIO TYPE, PORT COROUTINE PRIORITIES >
- using Mark3::CoroutineHandler = void(\*)(Coroutine \*pclCaller , void \*pvContext )

## 20.69.1 Detailed Description

CoRoutine implementation.

Definition in file coroutine.h.

## 20.70 coroutine.h

```
00001 /*-----
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00019 #pragma once
00020
00021 #include "mark3cfg.h"
00022 #include "ll.h"
00022 #include "priomapl1.h"
00024 #include "priomapl2.h"
00025
00026 namespace Mark3 {
00027
00028 \!\!\!// Priority map type declaration, based on port configuration
00029 #if PORT_COROUTINE_PRIORITIES <= (PORT_PRIO_MAP_WORD_SIZE \star 8u)
00030 using CoPrioMap =
      PriorityMapL1<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
00031 #else
00032 using CoPrioMap =
      PriorityMapL2<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
00033 #endif
00034
00035 // Forward declarations
00036 class CoList;
00037 class Coroutine;
00038
00039 // CoRoutine functino handler type definition
00040 using CoroutineHandler = void (*) (Coroutine* pclCaller_, void* pvContext_);
00053 class Coroutine : public TypedLinkListNode<Coroutine>
00054 {
00055 public:
00056
00057
          ~Coroutine();
00058
00070
          void Init(PORT_PRIO_TYPE uPriority_, CoroutineHandler pfHandler_,
     void* pvContext_);
00071
00077
          void Run();
00078
00084
          void Activate();
00085
00093
          void SetPriority(PORT_PRIO_TYPE uPriority_);
00094
00101
          PORT PRIO TYPE GetPriority();
00102
00103 private:
00104
          CoList* m_pclOwner;
          CoroutineHandler m_pfHandler;
00105
00106
          void* m_pvContext;
          PORT_PRIO_TYPE m_uPriority;
00108
          bool m bOueued:
00109 };
00110 } // namespace Mark3
```

# 20.71 /home/moslevin/projects/m3-repo/kernel/src/public/cosched.h File Reference

CoRoutine Scheduler implementation.

```
#include "mark3cfg.h"
#include "coroutine.h"
#include "colist.h"
```

#### Classes

class Mark3::CoScheduler

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

## **Namespaces**

· Mark3

# 20.71.1 Detailed Description

CoRoutine Scheduler implementation.

Definition in file cosched.h.

## 20.72 cosched.h

```
00001 /*===
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00019 #pragma once
00020
00021 #include "mark3cfg.h"
00022 #include "coroutine.h"
00023 #include "colist.h"
00024
00025 namespace Mark3 {
00033 class CoScheduler
00034 {
00035 public:
00036
00042
          static void Init();
00043
00050
          static CoPrioMap* GetPrioMap();
00051
00059
          static CoList* GetStopList();
00060
00068
          static CoList* GetCoList(PORT PRIO TYPE uPriority );
00069
00077
          static Coroutine* Schedule();
00078
00079 private:
00080
          static CoList m_aclPriorities[PORT_COROUTINE_PRIORITIES];
00081
           static CoList m_clStopList;
00082
           static CoPrioMap m_clPrioMap;
00083 };
00084 } // namespace Mark3
```

# 20.73 /home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h File Reference

RAII Critical Section Implementation.

```
#include "mark3cfg.h"
#include "criticalsection.h"
```

## **Classes**

· class Mark3::CriticalGuard

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

# **Namespaces**

• Mark3

# 20.73.1 Detailed Description

RAII Critical Section Implementation.

Definition in file criticalguard.h.

# 20.74 criticalguard.h

```
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00020 #pragma once
00021
00022 #include "mark3cfg.h"
00023 #include "criticalsection.h"
00024
00025 namespace Mark3 {
00026
00027 //----
00038 class CriticalGuard {
00039 public:
         CriticalGuard() {
00040
00041
              CriticalSection::Enter();
00042
00043
00044
          ~CriticalGuard() {
00045
              CriticalSection::Exit();
          }
00046
00047 };
00048 } // namespace Mark3
```

# 20.75 /home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h File Reference

## Critical Section Support.

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

### Classes

· class Mark3::CriticalSection

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

## **Namespaces**

• Mark3

## 20.75.1 Detailed Description

Critical Section Support.

Definition in file criticalsection.h.

# 20.76 criticalsection.h

```
00001 /*
00002
00003
00004 |
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00020 #pragma once
00021
00022 #include "mark3cfg.h"
00023 #include "threadport.h"
00024
00025 namespace Mark3 {
00026
00027 //---
00048 class CriticalSection {
00049 public:
00050
         static inline void Enter() {
00057
             PORT_CS_ENTER();
00058
00059
         static inline void Exit() {
00065
00066
            PORT_CS_EXIT();
00067
00068
00073
         static inline K_WORD NestingCount() {
00074
             return PORT_CS_NESTING();
00075
00076 };
00077
00078 } // namespace Mark3
```

# 20.77 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::EventFlag

The EventFlag class. This class implements 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.

## **Namespaces**

· Mark3

## 20.77.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

# 20.78 eventflag.h

```
00001
00002
00003
00004
00005
00006
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00018 #pragma once
00019
00020 #include "mark3cfg.h"
00021 #include "kernel.h"
00022 #include "kerneltypes.h"
00023 #include "blocking.h"
00025
00026 #if KERNEL_EVENT_FLAGS
00027 namespace Mark3
00028 {
00029 //--
00045 class EventFlag : public BlockingObject
```

```
00046 {
00047 public:
00048
         void* operator new(size_t sz, void* pv) { return reinterpret_cast<EventFlag*>(pv); };
00049
         ~EventFlag();
00050
00054
         void Init();
00055
00064
         uint16_t Wait(uint16_t u16Mask_, EventFlagOperation eMode_);
00065
         uint16_t Wait(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t u32TimeMS_);
00075
00076
         void WakeMe(Thread* pclChosenOne_);
00083
00084
00091
         void Set(uint16_t u16Mask_);
00092
00097
00098
         void Clear(uint16_t u16Mask_);
00103
         uint16_t GetMask();
00104
00105 private:
00117
         uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t
     u32TimeMS_);
00118
         uint16_t m_u16SetMask;
00119
00120 };
00121 } // namespace Mark3
00122 #endif // #if KERNEL_EVENT_FLAGS
```

# 20.79 /home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h File Reference

Thread porting interface.

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

#### **Classes**

· class Mark3::ThreadPort

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

## **Namespaces**

· Mark3

## 20.79.1 Detailed Description

Thread porting interface.

Definition in file ithreadport.h.

# 20.80 ithreadport.h

```
00001 /
00002
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00020 #pragma once
00021
00022 #include <stdint.h>
00023
00024 namespace Mark3
00025 {
00026 //----
00027 class Thread;
00035 class ThreadPort
00036 {
00037 public:
         static void Init() {}
00043
00044
00049
         static void StartThreads();
00050
        friend class Thread;
00051
00052 private:
00059
         static void InitStack(Thread* pstThread_);
00060 };
00061 } // namespace Mark3
```

# 20.81 /home/moslevin/projects/m3-repo/kernel/src/public/kernel.h File Reference

Kernel initialization and startup class.

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

## Classes

· class Mark3::Kernel

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

## **Namespaces**

• Mark3

# **Typedefs**

using DebugPrintFunction = void(\*)(const char \*szString\_)

20.82 kernel.h 503

## 20.81.1 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.

## 20.81.2 Typedef Documentation

#### 20.81.2.1 DebugPrintFunction

```
using DebugPrintFunction = void (*)(const char* szString_)
```

Definition at line 39 of file kernel.h.

# 20.82 kernel.h

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00032 #pragma once
00033
00034 #include "mark3cfg.h"
00035 #include "kerneltypes.h"
00036 #include "paniccodes.h"
00037 #include "thread.h"
00038
00039 using DebugPrintFunction = void (*)(const char* szString_);
00040
00041 namespace Mark3
00042 {
00043 //--
00048 class Kernel
00049 {
00050 public:
00058
          static void Init();
00059
00071
          static void Start();
00072
00078
          static void CompleteStart();
00079
00086
          static bool IsStarted() { return m_bIsStarted; }
          static void SetPanic(PanicFunc pfPanic_) { m_pfPanic = pfPanic_; }
static bool IsPanic() { return m_bIsPanic; }
00094
00099
00104
          static void Panic(uint16_t u16Cause_);
00105
```

```
00106 #if KERNEL_THREAD_CREATE_CALLOUT
         static void SetThreadCreateCallout(ThreadCreateCallout
     pfCreate_) { m_pfThreadCreateCallout = pfCreate_; }
00117 #endif // #if KERNEL_THREAD_CREATE_HOOK
00118
00119 #if KERNEL_THREAD_EXIT_CALLOUT
         static void SetThreadExitCallout(ThreadExitCallout pfExit_) {
     m_pfThreadExitCallout = pfExit_; ]
00131 #endif // #if KERNEL_THREAD_EXIT_CALLOUT
00132
00133 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00143
          static void SetThreadContextSwitchCallout(
     ThreadContextCallout pfContext_)
00144
00145
              m pfThreadContextCallout = pfContext;
00147 #endif // KERNEL_CONTEXT_SWITCH_CALLOUT
00155
         \verb|static| void SetDebugPrintFunction| (DebugPrintFunction|) \\
     pfPrintFunction_)
00156
00157
             m_pfDebugPrintFunction = pfPrintFunction_;
00159
00168
         static void DebugPrint(const char* szString_);
00169
00170 #if KERNEL THREAD CREATE CALLOUT
00171
          static ThreadCreateCallout GetThreadCreateCallout() { return
00178
     m_pfThreadCreateCallout; }
00179 #endif // #if KERNEL_THREAD_CREATE_HOOK
00180 #if KERNEL_THREAD_EXIT_CALLOUT
00181
         static ThreadExitCallout GetThreadExitCallout() { return
00188
     m_pfThreadExitCallout; }
00189 #endif // #if KERNEL_THREAD_EXIT_HOOK
00190 #if KERNEL_CONTEXT_SWITCH_CALLOUT
00191
00198
         static ThreadContextCallout
     GetThreadContextSwitchCallout() { return
     m_pfThreadContextCallout; }
00199 #endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00200 #if KERNEL_STACK_CHECK
00201
         static void
                         SetStackGuardThreshold(uint16_t u16Threshold_) {
     m_ul6GuardThreshold = ul6Threshold_; }
00202
         static uint16_t GetStackGuardThreshold() { return
     m_u16GuardThreshold; }
00203 #endif // #if KERNEL_STACK_CHECK
00204
00205
         static void
                         Tick() { m_u32Ticks++; }
00206
       static uint32_t GetTicks();
00207
00208 private:
                        m_bIsStarted;
m_bIsPanic;
00209 static bool
00210
         static bool
00211
        static PanicFunc m_pfPanic;
00212
00213 #if KERNEL THREAD CREATE CALLOUT
00214
         static ThreadCreateCallout m_pfThreadCreateCallout;
00215 #endif
                                                              // #if KERNEL_THREAD_CREATE_HOOK
00216 #if KERNEL_THREAD_EXIT_CALLOUT
         static ThreadExitCallout m_pfThreadExitCallout;
                                                          // #if KERNEL_THREAD_EXIT_HOOK
00218 #endif
00219 #if KERNEL_CONTEXT_SWITCH_CALLOUT
         static ThreadContextCallout m_pfThreadContextCallout;
00220
                                                                // #if KERNEL_CONTEXT_SWITCH_CALLOUT
00221 #endif
         static DebugPrintFunction m_pfDebugPrintFunction;
00223 #if KERNEL_STACK_CHECK
00224
         static uint16_t m_u16GuardThreshold;
00225 #endif // #if KERNEL_STACK_CHECK
        static uint32_t m_u32Ticks;
00226
00227 };
00229 } // namespace Mark3
```

# 20.83 /home/moslevin/projects/m3-repo/kernel/src/public/kerneldebug.h File Reference

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

20.84 kerneldebug.h 505

```
#include "mark3cfg.h"
#include "paniccodes.h"
#include "kernel.h"
```

## **Namespaces**

• Mark3

## **Macros**

• #define KERNEL\_ASSERT(x)

## 20.83.1 Detailed Description

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

Definition in file kerneldebug.h.

### 20.83.2 Macro Definition Documentation

### 20.83.2.1 KERNEL ASSERT

```
#define KERNEL_ASSERT(
```

Definition at line 36 of file kerneldebug.h.

# 20.84 kerneldebug.h

```
00001 /*
00002
00003
00004
00005
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00019 #pragma once
00020
00021 #include "mark3cfg.h"
00022 #include "paniccodes.h"
00023 #include "kernel.h"
00024
00025 //----
00026 namespace Mark3
00027 {
00028 #if KERNEL_DEBUG
```

# 20.85 /home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h File Reference

Kernel Software interrupt declarations.

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

#### **Classes**

· class Mark3::KernelSWI

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

# **Namespaces**

• Mark3

# 20.85.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

# 20.86 kernelswi.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00021 #pragma once
00022 #include "kerneltypes.h"
00023
00024 //---
```

# 20.87 /home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h File Reference

#### Kernel Timer Class declaration.

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

#### Classes

· class Mark3::KernelTimer

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

### **Namespaces**

· Mark3

## 20.87.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

# 20.88 kerneltimer.h

```
00001
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00021 #pragma once
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 namespace Mark3
00027 {
00028 //---
00034 class KernelTimer
00035 {
00036 public:
00041
        static void Config(void);
00042
00047
          static void Start (void);
00048
00053
          static void Stop(void);
00054 };
00055 } // namespace Mark3
```

# 20.89 /home/moslevin/projects/m3-repo/kernel/src/public/kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

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

## **Namespaces**

Mark3

# **Typedefs**

```
    using Mark3::PanicFunc = void(*)(uint16_t u16PanicCode_)
    using Mark3::IdleFunc = void(*)()
    using Mark3::ThreadEntryFunc = void(*)(void *pvArg_)
```

#### **Enumerations**

```
    enum Mark3::EventFlagOperation : uint8_t {
        Mark3::EventFlagOperation::All_Set = 0, Mark3::EventFlagOperation::Any_Set, Mark3::EventFlagOperation::Any_Clear,
        Operation::All_Clear, Mark3::EventFlagOperation::Any_Clear,
        Mark3::EventFlagOperation::Pending_Unblock }

    enum Mark3::ThreadState : uint8_t {
        Mark3::ThreadState::Exit = 0, Mark3::ThreadState::Blocked, Mark3::ThreadState::Blocked, Mark3::ThreadState::Stop,
        Mark3::ThreadState::Invalid }
```

## 20.89.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

# 20.90 kerneltypes.h

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00019 #include <stdint.h>
00020 #include <stdbool.h>
00021 #include <stddef.h>
```

```
00022
00023 #pragma once
00024 namespace Mark3
00025 {
00026 //----
00030 using PanicFunc = void (*) (uint16 t u16PanicCode );
00032 //----
00037 using IdleFunc = void (*)();
00038
00039 //----
00043 using ThreadEntryFunc = void (*)(void* pvArg_);
00050 enum class EventFlagOperation : uint8_t {
00050 enum class Eventria
00051 All_Set = 0,
00052 Any_Set,
00053 All_Clear,
00054 Any_Clear,
00055 Pending_Unblock
00056 );
00057
00058 //----
00062 enum class ThreadState : uint8_t { Exit = 0, Ready, Blocked,
      Stop, Invalid };
00063
00064 } // namespace Mark3
```

# 20.91 /home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h File Reference

Semaphore Blocking Object class declarations.

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

### Classes

· class Mark3::Semaphore

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

### **Namespaces**

Mark3

## 20.91.1 Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

# 20.92 ksemaphore.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013
00021 #pragma once
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 #include "blocking.h"
00027 #include "threadlist.h"
00028
00029 namespace Mark3
00030 {
00031 //-
00036 class Semaphore : public BlockingObject
00037 {
00038 public:
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Semaphore*>(pv); };
00039
00040
          ~Semaphore();
00041
00062
          void Init(uint16_t u16InitVal_, uint16_t u16MaxVal_);
00063
00077
          bool Post();
00078
00085
          void Pend();
00086
00097
          uint16_t GetCount();
00098
00109
          bool Pend(uint32_t u32WaitTimeMS_);
00110
00120
          void WakeMe(Thread* pclChosenOne_);
00121
00122 private:
00127
          uint8_t WakeNext();
00128
00137
          bool Pend_i(uint32_t u32WaitTimeMS_);
00138
          uint16_t m_u16Value;
00139
00140
          uint16_t m_u16MaxValue;
00141 };
00142 } // namespace Mark3
```

# 20.93 /home/moslevin/projects/m3-repo/kernel/src/public/II.h File Reference

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 u16 to keep our RTOS code efficient and logically-separated.

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

#### Classes

class Mark3::LinkListNode

The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

class Mark3::TypedLinkListNode< T >

20.94 II.h 511

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

· class Mark3::LinkList

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

· class Mark3::DoubleLinkList

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

· class Mark3::CircularLinkList

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

class Mark3::TypedDoubleLinkList< T >

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

class Mark3::TypedCircularLinkList< T >

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

## **Namespaces**

Mark3

# 20.93.1 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 u16 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.

## 20.94 II.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -
       -[Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00041 #pragma once
00042 #include "kerneltypes.h"
00043
00044 namespace Mark3
```

```
00045 {
00046 //--
00052 class LinkList;
00053 class DoubleLinkList;
00054 class CircularLinkList;
00055
00056 //---
00062 class LinkListNode
00063 {
00064 protected:
          LinkListNode* next:
00065
00066
          LinkListNode* prev;
00067
00068
         LinkListNode() {}
00069
00075
         void ClearNode();
00076
00077 public:
        LinkListNode* GetNext(void) { return next; }
00085
00093
          LinkListNode* GetPrev(void) { return prev; }
00094
          friend class LinkList;
00095
         friend class DoubleLinkList;
         friend class CircularLinkList;
00096
00097 };
00098
00099 //---
00107 template <typename T>
00108 class TypedLinkListNode : public LinkListNode
00109 {
00110 public:
        T* GetNext() { return static_cast<T*>(LinkListNode::GetNext()); }
00111
          T* GetPrev() { return static_cast<T*>(LinkListNode::GetPrev()); }
00113 };
00114 //---
00119 class LinkList
00120 {
00121 protected:
          LinkListNode* m_pclHead;
00123
          LinkListNode* m_pclTail;
00124
00125 public:
         void Init()
00131
00132
          {
00133
              m_pclHead = nullptr;
00134
              m_pclTail = nullptr;
00135
00136
00144
          LinkListNode* GetHead() { return m_pclHead; }
00145
00153
          void SetHead(LinkListNode* pclNode_) { m_pclHead = pclNode_; }
00154
00162
          LinkListNode* GetTail() { return m_pclTail; }
00163
00171
00172 };
          void SetTail(LinkListNode* pclNode_) { m_pclTail = pclNode_; }
00173
00179 class DoubleLinkList : public LinkList
00180 {
00181 public:
00182
         void* operator new(size t sz, void* pv) { return reinterpret cast<
     DoubleLinkList*>(pv); };
00188
         DoubleLinkList()
00189
          {
00190
              m_pclHead = nullptr;
00191
              m_pclTail = nullptr;
00192
          }
00193
00201
          void Add(LinkListNode* node_);
00202
00210
          void Remove(LinkListNode* node_);
00211 };
00212
00213 //---
00218 class CircularLinkList : public LinkList
00219 {
00220 public:
00221
       void* operator new(size_t sz, void* pv) { return (CircularLinkList*)pv; };
00222
          CircularLinkList()
00223
          {
00224
              m pclHead = nullptr;
             m_pclTail = nullptr;
00225
00226
00227
00234
          void Add(LinkListNode* node_);
00235
00242
         void Remove(LinkListNode* node_);
```

```
00243
00249
          void PivotForward();
00250
00256
         void PivotBackward();
00257
         void InsertNodeBefore(LinkListNode* node_, LinkListNode* insert_);
00266
00267 };
00268
00269 //--
00275 template <typename T>
00276 class TypedDoubleLinkList : public DoubleLinkList
00277 {
00278 public:
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<</pre>
     TypedDoubleLinkList<T>*>(pv); }
00280
00281
          TypedDoubleLinkList<T>() {
00282
              DoubleLinkList::Init();
00283
00284
00291
          T* GetHead() { return static_cast<T*>(DoubleLinkList::GetHead()); }
00292
00299
         void SetHead(T* pclNode_) { DoubleLinkList::SetHead(pclNode_); }
00300
          T* GetTail() { return static_cast<T*>(DoubleLinkList::GetTail()); }
00307
00308
00315
          void SetTail(T* pclNode_) { DoubleLinkList::SetTail(pclNode_); }
00316
00323
          void Add(T* pNode_)
00324
00325
              DoubleLinkList::Add(pNode);
00326
         }
00327
00334
          void Remove(T* pNode_)
00335
              DoubleLinkList::Remove(pNode_);
00336
00337
00338 };
00339
00340 //----
00346 template <typename T>
00347 class TypedCircularLinkList : public CircularLinkList
00348 {
00349 public:
         void* operator new(size_t sz, void* pv) { return reinterpret_cast<</pre>
00350
     TypedCircularLinkList<T>*>(pv); }
00351
00352
          TypedCircularLinkList<T>() {
00353
              CircularLinkList::Init();
00354
00355
00362
         T* GetHead() { return static_cast<T*>(CircularLinkList::GetHead()); }
00363
00370
         void SetHead(T* pclNode_) { CircularLinkList::SetHead(pclNode_); }
00371
00378
          T* GetTail() { return static cast<T*>(CircularLinkList::GetTail()); }
00379
00386
          void SetTail(T* pclNode_) { CircularLinkList::SetTail(pclNode_); }
00387
00394
          void Add(T* pNode_)
00395
00396
              CircularLinkList::Add(pNode);
00397
         }
00398
00405
          void Remove (T* pNode_)
00406
00407
              CircularLinkList::Remove(pNode_);
00408
00409
00418
          void InsertNodeBefore(T* pNode_, T* pInsert_)
00419
00420
              CircularLinkList::InsertNodeBefore(pNode_, pInsert_);
00421
          }
00422 };
00423 } // namespace Mark3
```

# 20.95 /home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h File Reference

Mutex RAII helper class.

```
#include "mark3.h"
```

#### Classes

· class Mark3::LockGuard

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

# **Namespaces**

· Mark3

## 20.95.1 Detailed Description

Mutex RAII helper class.

Definition in file lockguard.h.

# 20.96 lockguard.h

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2018 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00020 #pragma once
00021
00022 #include "mark3.h"
00023
00024 namespace Mark3
00025 {
00032 class LockGuard
00033 {
00034 public:
00038
          LockGuard(Mutex* pclMutex);
00039
          LockGuard(Mutex* pclMutex, uint32_t u32TimeoutMs_);
00044
00046
          ~LockGuard();
00047
          bool isAcquired() { return m_bIsAcquired; }
00054
00055
00056 private:
00057
                 m bIsAcquired;
          bool
00058
          Mutex* m_pclMutex;
00059 };
00060 } // namespace Mark3
```

# 20.97 /home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h File Reference

## Mailbox + Envelope IPC Mechanism.

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

20.98 mailbox.h 515

#### **Classes**

· class Mark3::Mailbox

The Mailbox class. This class implements an IPC mechnism based on sending/receiving envelopes containing data of a fixed size, configured at initialization) that reside within a buffer of memory provided by the user.

### **Namespaces**

• Mark3

## 20.97.1 Detailed Description

Mailbox + Envelope IPC Mechanism.

Definition in file mailbox.h.

## 20.98 mailbox.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009
       -[Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00020 #pragma once
00021
00022 #include "mark3cfg.h"
00023 #include "kerneltypes.h"
00024 #include "ithreadport.h"
00025 #include "ksemaphore.h"
00026
00027 namespace Mark3
00028 {
00035 class Mailbox
00036 {
00037 public:
00038
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Mailbox*>(pv); }
00039
          ~Mailbox();
00040
00050
          void Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_);
00051
00064
          static Mailbox* Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_);
00065
00078
          bool Send(void* pvData_);
00079
00092
          bool SendTail(void* pvData_);
00093
00107
          bool Send(void* pvData_, uint32_t u32TimeoutMS_);
00108
00122
          bool SendTail(void* pvData_, uint32_t u32TimeoutMS_);
00123
00132
          void Receive(void* pvData_);
00133
00142
          void ReceiveTail(void* pvData_);
00143
00155
          bool Receive(void* pvData_, uint32_t u32TimeoutMS_);
00156
00168
          bool ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_);
00169
          uint16_t GetFreeSlots(void)
00171
00172
              const auto cs = CriticalGuard{};
00173
              return m_u16Free;
```

```
00175
00176
         bool IsFull(void) { return (GetFreeSlots() == 0); }
00177
         bool IsEmpty(void) { return (GetFreeSlots() == m_u16Count); }
00178
00179 private:
         void* GetHeadPointer(void)
00188
00189
              auto uAddr = reinterpret_cast<K_ADDR>(m_pvBuffer);
00190
             uAddr += static_cast<K_ADDR>(m_u16ElementSize) * static_cast<K_ADDR>(
     m_u16Head);
00191
             return reinterpret cast<void*>(uAddr);
00192
00193
00201
          void* GetTailPointer(void)
00202
              auto uAddr = reinterpret_cast<K_ADDR>(m_pvBuffer);
00203
              uAddr += static_cast<K_ADDR>(m_u16ElementSize) * static_cast<K_ADDR>(
00204
     m_u16Tail);
00205
             return reinterpret_cast<void*>(uAddr);
00206
00207
00216
         void CopyData(const void* src_, void* dst_, uint16_t len_)
00217
00218
             auto* u8Src = reinterpret_cast<const uint8_t*>(src_);
             auto* u8Dst = reinterpret_cast<uint8_t*>(dst_);
00220
              while (len_--) { *u8Dst++ = *u8Src++; }
00221
         }
00222
00227
         void MoveTailForward(void)
00228
00229
              m_u16Tail++;
00230
              if (m_u16Tail == m_u16Count) {
00231
                 m_u16Tail = 0;
00232
         }
00233
00234
          void MoveHeadForward(void)
00240
00241
              m_u16Head++;
00242
              if (m_u16Head == m_u16Count) {
00243
                  m_u16Head = 0;
00244
00245
         }
00246
00251
          void MoveTailBackward(void)
00252
              if (m_u16Tail == 0) {
00253
                  m_u16Tail = m_u16Count;
00254
00255
             m_u16Tail--;
00257
00258
00263
         void MoveHeadBackward(void)
00264
00265
              if (m u16Head == 0) {
                 m_u16Head = m_u16Count;
00267
00268
             m_u16Head--;
00269
         }
00270
00280
          bool Send_i(const void* pvData_, bool bTail_, uint32_t u32TimeoutMS_);
00281
00291
         bool Receive_i(void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_);
00292
00293
         uint16_t m_u16Head;
00294
         uint16_t m_u16Tail;
00295
00296
         uint16_t
                           m_u16Count;
         volatile uint16_t m_u16Free;
00298
         uint16_t     m_u16ElementSize;
00299
         const void* m_pvBuffer;
00300
00301
00302
          Semaphore m clRecvSem;
00303
         Semaphore m_clSendSem;
00304 };
00305 } // namespace Mark3
```

# 20.99 /home/moslevin/projects/m3-repo/kernel/src/public/manual.h File Reference

20.100 manual.h 517

## 20.99.1 Detailed Description

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

Definition in file manual.h.

## 20.100 manual.h

# 20.101 /home/moslevin/projects/m3-repo/kernel/src/public/mark3.h File Reference

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

```
#include "mark3cfg.h"
#include "threadport.h"
#include "criticalsection.h"
#include "criticalguard.h"
#include "kerneltypes.h"
#include "kerneldebug.h"
#include "ithreadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "kernel.h"
#include "thread.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "mutex.h"
#include "lockguard.h"
#include "eventflag.h"
#include "message.h"
#include "notify.h"
#include "mailbox.h"
#include "readerwriter.h"
#include "condvar.h"
#include "atomic.h"
#include "profile.h"
#include "autoalloc.h"
#include "priomap.h"
#include "threadlist.h"
#include "threadlistlist.h"
#include "schedulerguard.h"
#include "coroutine.h"
#include "colist.h"
#include "cosched.h"
```

# 20.101.1 Detailed Description

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

Definition in file mark3.h.

## 20.102 mark3.h

```
00003
00004
00005
00006 1
00007
00008
00009
        -[Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00020 #pragma once
00021
00022 #include "mark3cfg.h"
00023 #include "threadport.h"
00024 #include "criticalsection.h"
00025 #include "criticalguard.h"
00026
00027 #include "kerneltypes.h"
00028 #include "kerneldebug.h"
00029
00030 #include "ithreadport.h"
00031 #include "kernelswi.h"
00032 #include "kerneltimer.h"
00033
00034 #include "kernel.h"
00035 #include "thread.h"
00036 #include "timerlist.h"
00037
00038 #include "ksemaphore.h"
00039 #include "mutex.h"
00040 #include "lockguard.h"
00041 #include "eventflag.h"
00042 #include "message.h"
00042 "Include "notify.h"
00044 #include "mailbox.h"
00045 #include "readerwriter.h"
00046 #include "condvar.h"
00047
00048 #include "atomic.h"
00049
00050 #include "profile.h"
00051 #include "autoalloc.h'
00052 #include "priomap.h"
00053
00054 #include "threadlist.h"
00055 #include "threadlistlist.h"
00056
00057 #include "schedulerguard.h"
00059 #include "coroutine.h"
00060 #include "colist.h"
00061 #include "cosched.h"
00062
```

# 20.103 /home/moslevin/projects/m3-repo/kernel/src/public/mark3cfg.h File Reference

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.

```
#include "portcfg.h"
```

## **Macros**

- #define KERNEL\_DEBUG (0)
- #define KERNEL\_STACK\_CHECK (1)
- #define KERNEL NAMED THREADS (1)
- #define KERNEL\_EVENT\_FLAGS (1)
- #define KERNEL\_CONTEXT\_SWITCH\_CALLOUT (1)
- #define KERNEL\_THREAD\_CREATE\_CALLOUT (1)
- #define KERNEL\_THREAD\_EXIT\_CALLOUT (1)
- #define KERNEL ROUND ROBIN (1)
- #define KERNEL EXTENDED CONTEXT (1)

include CPU/Port specific configuration options

## 20.103.1 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.

Note: in the R7 and beyond version of the kernel, all options are enabled by default. As a result, the only configuration options presented are now located within the architecture-specific "portcfg.h".

Definition in file mark3cfg.h.

#### 20.103.2 Macro Definition Documentation

## 20.103.2.1 KERNEL\_CONTEXT\_SWITCH\_CALLOUT

```
#define KERNEL_CONTEXT_SWITCH_CALLOUT (1)
```

When enabled, this feature allows a user to define a callback to be executed whenever a context switch occurs. Enabling this provides a means for a user to track thread statistics, but it does result in additional overhead during a context switch.

Definition at line 70 of file mark3cfg.h.

### 20.103.2.2 KERNEL\_DEBUG

#define KERNEL\_DEBUG (0)

Enable kernel asserts at runtime.

Definition at line 30 of file mark3cfg.h.

#### 20.103.2.3 KERNEL\_EVENT\_FLAGS

```
#define KERNEL_EVENT_FLAGS (1)
```

This flag enables the event-flags synchronization object. This feature allows threads to be blocked, waiting on specific condition bits to be set or cleared on an EventFlag object.

While other synchronization objects are enabled by default, this one is configurable because it impacts the Thread object's member data.

Definition at line 62 of file mark3cfg.h.

#### 20.103.2.4 KERNEL\_EXTENDED\_CONTEXT

```
#define KERNEL_EXTENDED_CONTEXT (1)
```

include CPU/Port specific configuration options

Provide a special data pointer in the thread object, which may be used to add additional context to a thread. Typically this would be used to implement thread-local-storage.

Definition at line 97 of file mark3cfg.h.

#### 20.103.2.5 KERNEL NAMED THREADS

```
#define KERNEL_NAMED_THREADS (1)
```

Enabling this provides the Thread::SetName() and Thread::GetName() methods, allowing for each thread to be named with a null-terminated const char\* string.

Note: the string passed to Thread::SetName() must persist for the lifetime of the thread

Definition at line 52 of file mark3cfg.h.

#### 20.103.2.6 KERNEL\_ROUND\_ROBIN

```
#define KERNEL_ROUND_ROBIN (1)
```

Enable round-robin scheduling within each priority level. When selected, this results in a small performance hit during context switching and in the system tick handler, as a special software timer is used to manage the running thread's quantum. Can be disabled to optimize performance if not required.

Definition at line 90 of file mark3cfg.h.

20.104 mark3cfg.h 521

#### 20.103.2.7 KERNEL\_STACK\_CHECK

```
#define KERNEL_STACK_CHECK (1)
```

Perform stack-depth checks on threads at each context switch, which is useful in detecting stack overflows / near overflows. Near-overflow detection uses thresholds defined in the target's portcfg.h. Enabling this also adds the Thread::GetStackSlack() method, which allows a thread's stack to be profiled on-demand.

Note: When enabled, the additional stack checks result in a performance hit to context switches and thread initialization.

Definition at line 43 of file mark3cfg.h.

### 20.103.2.8 KERNEL\_THREAD\_CREATE\_CALLOUT

```
#define KERNEL_THREAD_CREATE_CALLOUT (1)
```

This feature provides a user-defined kernel callback that is executed whenever a thread is started.

Definition at line 76 of file mark3cfg.h.

#### 20.103.2.9 KERNEL\_THREAD\_EXIT\_CALLOUT

```
#define KERNEL_THREAD_EXIT_CALLOUT (1)
```

This feature provides a user-defined kernel callback that is executed whenever a thread is terminated.

Definition at line 82 of file mark3cfg.h.

# 20.104 mark3cfg.h

```
00001 /
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00025 #pragma once
00026
00030 #define KERNEL_DEBUG (0)
00031
00043 #define KERNEL_STACK_CHECK (1)
00044
00052 #define KERNEL_NAMED_THREADS (1)
00053
00062 #define KERNEL_EVENT_FLAGS (1)
00063
00070 #define KERNEL_CONTEXT_SWITCH_CALLOUT (1)
00071
00076 #define KERNEL_THREAD_CREATE_CALLOUT (1)
00077
00082 #define KERNEL_THREAD_EXIT_CALLOUT (1)
00083
00090 #define KERNEL_ROUND_ROBIN (1)
00091
00097 #define KERNEL_EXTENDED_CONTEXT (1)
00098
00099 #include "portcfg.h"
```

# 20.105 /home/moslevin/projects/m3-repo/kernel/src/public/message.h File Reference

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.

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

#### **Classes**

· class Mark3::Message

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

• class Mark3::MessagePool

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

· class Mark3::MessageQueue

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

# Namespaces

Mark3

## 20.105.1 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.

20.106 message.h 523

# 20.105.2 using Messages, Queues, and the Global Message Pool

```
// Declare a message queue shared between two threads
MessageQueue my_queue;
int main()
    // Initialize the message queue
    my_queue.init();
void Thread1()
    // Example TX thread - sends a message every 10ms
    while(1)
        // Grab a message from the global message pool
        Message *tx_message = GlobalMessagePool::Pop();
        // Set the message data/parameters
        tx_message->SetCode(1234);
        tx_message->SetData( nullptr );
        // Send the message on the queue.
        my_queue.Send( tx_message );
        Thread::Sleep(10);
}
void Thread2()
        // 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.

# 20.106 message.h

```
00001 /*===
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform] ---
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00078 #pragma once
00079
00080 #include "kerneltypes.h"
00081 #include "mark3cfg.h"
00082
00083 #include "11.h"
00084 #include "ksemaphore.h"
00085 #include "timerlist.h"
00086
00087 namespace Mark3
00088 {
00089 //----
00097 class Message : public TypedLinkListNode<Message>
00098 {
00099 public:
00100
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Message*>(pv); }
```

```
00105
          void Init()
00106
00107
              ClearNode();
              m_pvData = nullptr;
m_u16Code = 0;
00108
00109
00110
          }
00111
00118
          void SetData(void* pvData_) { m_pvData = pvData_; }
00125
          void* GetData() { return m_pvData; }
          void SetCode(uint16_t u16Code_) { m_u16Code = u16Code_; }
00132
          uint16_t GetCode() { return m_u16Code; }
00139
00140
00141 private:
          void* m_pvData;
00144
00146
          uint16_t m_u16Code;
00147 };
00148
00157 class MessagePool
00158 {
00159 public:
00160
          void* operator new(size_t sz, void* pv) { return (MessagePool*)pv; }
00161
          ~MessagePool() {}
00167
          void Init();
00168
00178
          void Push (Message* pclMessage_);
00179
00188
          Message* Pop();
00189
00197
          Message* GetHead():
00198
00199 private:
00201
          TypedDoubleLinkList<Message> m_clList;
00202 };
00203
00204 //--
00210 class MessageQueue
00211 {
00212 public:
00213
          void* operator new(size_t sz, void* pv) { return (MessageQueue*)pv; }
00214
          ~MessageQueue() {}
00215
00221
          void Init();
00222
00231
          Message* Receive();
00232
00247
          Message* Receive(uint32_t u32TimeWaitMS_);
00248
00257
          void Send(Message* pclSrc );
00258
00266
          uint16_t GetCount();
00267
00268 private:
          Message* Receive_i(uint32_t u32TimeWaitMS_);
00278
00279
          Semaphore m_clSemaphore;
00282
00284
          TypedDoubleLinkList<Message> m_clLinkList;
00285 };
00286 } // namespace Mark3
```

# 20.107 /home/moslevin/projects/m3-repo/kernel/src/public/mutex.h File Reference

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.

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

20.108 mutex.h 525

# **Classes**

· class Mark3::Mutex

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

#### **Namespaces**

• Mark3

#### 20.107.1 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.

#### 20.107.2 Initializing

Initializing a mutex object by calling:

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

## 20.107.3 Resource protection example

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

Definition in file mutex.h.

### 20.108 mutex.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] --
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00049 #pragma once
00050
00051 #include "kerneltypes.h"
00052 #include "mark3cfg.h"
00053
```

```
00054 #include "blocking.h"
00056 namespace Mark3
00057 {
00058 //---
00063 class Mutex : public BlockingObject
00065 public:
00066 void* operator new(size_t sz, void* pv) { return reinterpret_cast<Mutex*>(pv); };
00067
         ~Mutex();
00068
00076
         void Init(bool bRecursive_ = true);
00077
00094
         void Claim();
00095
00105
         bool Claim(uint32_t u32WaitTimeMS_);
00106
00118
         void WakeMe(Thread* pclOwner_);
00119
00139
         void Release();
00140
00141 private:
         uint8_t WakeNext();
00147
00148
00156
         bool Claim_i(uint32_t u32WaitTimeMS_);
00157
00158
         uint8_t
                       m_u8Recurse;
         bool m_bReady;
00159
00160
         bool
                       m_bRecursive;
         PORT_PRIO_TYPE m_uMaxPri;
00161
00162
         Thread* m_pclOwner;
00163 };
00164 } // namespace Mark3
```

# 20.109 /home/moslevin/projects/m3-repo/kernel/src/public/notify.h File Reference

Lightweight thread notification - blocking object.

```
#include "mark3cfg.h"
#include "blocking.h"
```

#### Classes

· class Mark3::Notify

The Notify class. This class provides a blocking object type that allows one or more threads to wait for an event to occur before resuming operation.

## **Namespaces**

• Mark3

## 20.109.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.h.

20.110 notify.h 527

# 20.110 notify.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00021 #pragma once
00022
00023 #include "mark3cfg.h"
00024 #include "blocking.h"
00025
00026 namespace Mark3
00027
00033 class Notify : public BlockingObject
00034 {
00035 public:
00036
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Notify*>(pv); };
00037
          ~Notify();
00038
00043
          void Init (void);
00044
00053
          void Signal(void);
00054
00063
          void Wait (bool* pbFlag_);
00064
00076
          bool Wait (uint32_t u32WaitTimeMS_, bool* pbFlaq_);
00077
00086
          void WakeMe(Thread* pclChosenOne_);
00087
00088 private:
00089
          bool m_bPending;
00090 };
00091 } // namespace Mark3
```

# 20.111 /home/moslevin/projects/m3-repo/kernel/src/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

## **Macros**

- #define PANIC\_ASSERT\_FAILED (1)
- #define PANIC\_LIST\_UNLINK\_FAILED (2)
- #define PANIC\_STACK\_SLACK\_VIOLATED (3)
- #define PANIC\_AUTO\_HEAP\_EXHAUSTED (4)
- #define PANIC\_POWERMAN\_EXHAUSTED (5)
- #define PANIC\_NO\_READY\_THREADS (6)
- #define PANIC\_RUNNING\_THREAD\_DESCOPED (7)
- #define PANIC\_ACTIVE\_SEMAPHORE\_DESCOPED (8)
- #define PANIC\_ACTIVE\_MUTEX\_DESCOPED (9)
- #define PANIC\_ACTIVE\_EVENTFLAG\_DESCOPED (10)
- #define PANIC\_ACTIVE\_NOTIFY\_DESCOPED (11)
- #define PANIC ACTIVE MAILBOX DESCOPED (12)
- #define PANIC\_ACTIVE\_TIMER\_DESCOPED (13)
- #define PANIC\_ACTIVE\_COROUTINE\_DESCOPED (14)

# 20.111.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file paniccodes.h.

# 20.111.2 Macro Definition Documentation

## 20.111.2.1 PANIC\_ACTIVE\_COROUTINE\_DESCOPED

```
#define PANIC_ACTIVE_COROUTINE_DESCOPED (14)
```

Definition at line 35 of file paniccodes.h.

## 20.111.2.2 PANIC\_ACTIVE\_EVENTFLAG\_DESCOPED

```
#define PANIC_ACTIVE_EVENTFLAG_DESCOPED (10)
```

Definition at line 31 of file paniccodes.h.

# 20.111.2.3 PANIC\_ACTIVE\_MAILBOX\_DESCOPED

```
#define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
```

Definition at line 33 of file paniccodes.h.

## 20.111.2.4 PANIC\_ACTIVE\_MUTEX\_DESCOPED

```
#define PANIC_ACTIVE_MUTEX_DESCOPED (9)
```

Definition at line 30 of file paniccodes.h.

# 20.111.2.5 PANIC\_ACTIVE\_NOTIFY\_DESCOPED

```
#define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
```

Definition at line 32 of file paniccodes.h.

#### 20.111.2.6 PANIC\_ACTIVE\_SEMAPHORE\_DESCOPED

```
#define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
```

Definition at line 29 of file paniccodes.h.

#### 20.111.2.7 PANIC\_ACTIVE\_TIMER\_DESCOPED

```
#define PANIC_ACTIVE_TIMER_DESCOPED (13)
```

Definition at line 34 of file paniccodes.h.

## 20.111.2.8 PANIC\_ASSERT\_FAILED

```
#define PANIC_ASSERT_FAILED (1)
```

Definition at line 22 of file paniccodes.h.

# 20.111.2.9 PANIC\_AUTO\_HEAP\_EXHAUSTED

```
#define PANIC_AUTO_HEAP_EXHAUSTED (4)
```

Definition at line 25 of file paniccodes.h.

# 20.111.2.10 PANIC\_LIST\_UNLINK\_FAILED

```
#define PANIC_LIST_UNLINK_FAILED (2)
```

Definition at line 23 of file paniccodes.h.

# 20.111.2.11 PANIC\_NO\_READY\_THREADS

```
#define PANIC_NO_READY_THREADS (6)
```

Definition at line 27 of file paniccodes.h.

#### 20.111.2.12 PANIC\_POWERMAN\_EXHAUSTED

```
#define PANIC_POWERMAN_EXHAUSTED (5)
```

Definition at line 26 of file paniccodes.h.

#### 20.111.2.13 PANIC\_RUNNING\_THREAD\_DESCOPED

```
#define PANIC_RUNNING_THREAD_DESCOPED (7)
```

Definition at line 28 of file paniccodes.h.

#### 20.111.2.14 PANIC\_STACK\_SLACK\_VIOLATED

```
#define PANIC_STACK_SLACK_VIOLATED (3)
```

Definition at line 24 of file paniccodes.h.

# 20.112 paniccodes.h

```
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00020 #pragma once
00021
00022 #define PANIC_ASSERT_FAILED (1)
00023 #define PANIC_LIST_UNLINK_FAILED (2)
00024 #define PANIC_STACK_SLACK_VIOLATED (3)
00025 #define PANIC_AUTO_HEAP_EXHAUSTED (4)
00026 #define PANIC_POWERMAN_EXHAUSTED (5)
00027 #define PANIC_NO_READY_THREADS (6)
00028 #define PANIC_RUNNING_THREAD_DESCOPED (7)
00029 #define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
00030 #define PANIC_ACTIVE_MUTEX_DESCOPED (9)
00031 #define PANIC_ACTIVE_EVENTFLAG_DESCOPED
00032 #define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
00033 #define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
00034 #define PANIC_ACTIVE_TIMER_DESCOPED (13)
00035 #define PANIC_ACTIVE_COROUTINE_DESCOPED (14)
```

# 20.113 /home/moslevin/projects/m3-repo/kernel/src/public/priomap.h File Reference

## Priority map data structure.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "priomapl1.h"
#include "priomapl2.h"
```

20.114 priomap.h 531

## **Namespaces**

· Mark3

# **Typedefs**

using Mark3::PriorityMap = PriorityMapL1 < PORT PRIO TYPE, KERNEL NUM PRIORITIES >

# 20.113.1 Detailed Description

Priority map data structure.

Definition in file priomap.h.

# 20.114 priomap.h

```
00002
00003
00004
00005 1
00006
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00019 #pragma once
00020
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023
00024 #include "priomapl1.h"
00025 #include "priomapl2.h"
00026
00027 namespace Mark3 {
00028 #if KERNEL_NUM_PRIORITIES <= (PORT_PRIO_MAP_WORD_SIZE \star 8u)
00029 using PriorityMap =
      PriorityMapL1<PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES>;
00030 #else
00031 using PriorityMap =
      PriorityMapL2<PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES>;
00032 #endif
00033 } // namespace Mark3
```

# 20.115 /home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h File Reference

1-Level bitmap allocator template-class used for scheduler implementation

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

#### Classes

class Mark3::PriorityMapL1
 T, C >

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

## **Namespaces**

• Mark3

## 20.115.1 Detailed Description

1-Level bitmap allocator template-class used for scheduler implementation

Definition in file priomapl1.h.

# 20.116 priomapl1.h

```
00001
00002
00003
00004
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform] --
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =
00020 #pragma once
00021
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024 #include "threadport.h"
00025
00026 namespace Mark3
00027 (
00028 //---
00044 template <typename T, size_t C>
00045 class PriorityMapL1
00046 {
00047 public:
00052
          PriorityMapL1()
00053
00054
              m_uXPriorityMap = 0;
00055
          }
00056
00062
          void Set(T uXPrio_)
00063
00064
              auto uXPrioBit = PrioBit(uXPrio);
              m_uXPriorityMap |= (1 << uXPrioBit);</pre>
00065
00066
          }
00067
00073
00074
          void Clear(T uXPrio_)
00075
              auto uXPrioBit = PrioBit(uXPrio);
00076
              m_uXPriorityMap &= ~(1 << uXPrioBit);</pre>
00077
          }
00078
00086
          T HighestPriority(void)
00087
00088
              auto uXPrio = PriorityFromBitmap(m_uXPriorityMap);
00089
              return uXPrio:
00090
          }
00091
```

```
00092 private:
           static inline T PrioBit(T prio) { return prio & m_uXPrioMapBitMask; }
00094
            static inline T PrioMapWordIndex(T prio) { return prio >>
       m_uXPrioMapWordShift; }
00096
            static inline T PriorityFromBitmap(T uXPrio_)
00098
00099 #if PORT_USE_HW_CLZ
00100
                // Support hardware-accelerated Count-leading-zeros instruction
00101
                return m_uXPrioMapBits - PORT_CLZ(uXPrio_);
00102 #else
             // Default un-optimized count-leading zeros operation
T uXMask = 1 << (m_uXPrioMapBits - 1);</pre>
00103
00104
00105
                auto
                                   u8Zeros = T { 0 };
00106
00107
                while (uXMask) {
                  if (uXMask & uXPrio_) {
    return (m_uXPrioMapBits - u8Zeros);
00108
00109
00110
00111
00112
                    uXMask >>= 1;
00113
                    u8Zeros++;
00114
00115
                 return 0;
00116 #endif
00117
00118
         static constexpr size_t m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0, 0, 6};
static constexpr auto m_uXPrioMapWordShift = T { m_uXPrioMapShiftLUT[sizeof(T)] };
static constexpr auto m_uXPrioMapBits = T { 8 * sizeof(T) };
00119
00120
00121
00122
            static constexpr auto m_uXPrioMapBitMask = T { (1 <<</pre>
      m_uXPrioMapWordShift) - 1 };
00123
00124
00125 };
            T m_uXPriorityMap;
00126 } // namespace Mark3
```

# 20.117 /home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h File Reference

2-Level priority allocator template-class used for scheduler implementation

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

# Classes

class Mark3::PriorityMapL2< T, C >

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

#### **Namespaces**

Mark3

## 20.117.1 Detailed Description

2-Level priority allocator template-class used for scheduler implementation

Definition in file priomapl2.h.

# 20.118 priomapl2.h

```
00001 /*=======
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ======
00020 #pragma once
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024 #include "threadport.h"
00025
00026 namespace Mark3
00028 //---
00049 template <typename T, size_t C>
00050 class PriorityMapL2
00051 {
00052 public:
          PriorityMapL2()
00058
00059
               m_uXPriorityMapL2 = 0;
     for (auto i = PORT_PRIO_TYPE{0}; i < m_uXPrioMapNumWords; i++) {
   m_auXPriorityMap[i] = 0; }</pre>
00060
00061
          }
00062
00068
           void Set(T uXPrio_)
00069
00070
               auto uXPrioBit = PrioBit(uXPrio_);
00071
               auto uXWordIdx = PrioMapWordIndex(uXPrio_);
00072
00073
               m_auXPriorityMap[uXWordIdx] |= (1 << uXPrioBit);</pre>
00074
               m_uXPriorityMapL2 |= (1 << uXWordIdx);</pre>
00075
          }
00076
00082
           void Clear(T uXPrio_)
00083
00084
               auto uXPrioBit = PrioBit(uXPrio_);
00085
               auto uXWordIdx = PrioMapWordIndex(uXPrio_);
00086
00087
               m_auXPriorityMap[uXWordIdx] &= ~(1 << uXPrioBit);</pre>
               if (!m_auXPriorityMap[uXWordIdx]) {
   m_uXPriorityMapL2 &= ~(1 << uXWordIdx);</pre>
00088
00089
00090
00091
          }
00092
00100
           T HighestPriority(void)
00101
00102
               auto uXMapIdx = PriorityFromBitmap(m_uXPriorityMapL2);
00103
               if (!uXMapIdx) {
00104
                   return 0;
00105
               uXMapIdx--;
00106
00107
               auto uXPrio = PriorityFromBitmap(m_auXPriorityMap[uXMapIdx]);
00108
               uXPrio += (uXMapIdx * m_uXPrioMapBits);
               return uXPrio;
00109
00110
00111
00112 private:
00113
          static inline T PrioBit(T prio) { return prio & m_uXPrioMapBitMask; }
00114
           static inline T PrioMapWordIndex(T prio) { return prio >>
00115
      m_uXPrioMapWordShift; }
00116
00117
           static inline T PriorityFromBitmap(T uXPrio_)
00118
00119 #if PORT USE HW CLZ
               // Support hardware-accelerated Count-leading-zeros instruction
00120
               return m_uXPrioMapBits - PORT_CLZ(uXPrio_);
00121
00122 #else
00123
               // Default un-optimized count-leading zeros operation
00124
               T uXMask = 1 << (m_uXPrioMapBits - 1);</pre>
                               u8Zeros = T { 0 };
00125
               auto
00126
00127
               while (uXMask) {
00128
                  if (uXMask & uXPrio_) {
00129
                        return (m_uXPrioMapBits - u8Zeros);
```

```
00130
               }
00131
00132
               uXMask >>= 1;
00133
              u8Zeros++;
00134
00135
            return 0:
00136 #endif
00137
00138
      00139
00140
00141
00142
        static constexpr auto m_uXPrioMapBitMask = T { (1 <<</pre>
    m_uXPrioMapWordShift) - 1 };
00143
00144
        // Required size of the bitmap array in words
00145
        static constexpr auto m_uXPrioMapNumWords
           = T { (C + (m_uXPrioMapBits - 1)) / m_uXPrioMapBits };
00146
        T m_auXPriorityMap[m_uXPrioMapNumWords];
       T m_uXPriorityMapL2;
00149
00150 };
00151 } // namespace Mark3
```

# 20.119 /home/moslevin/projects/m3-repo/kernel/src/public/profile.h File Reference

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.

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

# Classes

· class Mark3::ProfileTimer

Profiling timer. This class is used to perform high-performance profiling of code to see how int32\_t certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

# Namespaces

Mark3

# 20.119.1 Detailed Description

High-precision profiling timers Enables the profiling and instrumentation of performance-critical code. Multiple timers can be used simultaneously to enable system-wide performance metrics to be computed in a lightweight manner.

Usage:

```
ProfileTimer clMyTimer;
int i;

clMyTimer.Init();

// Profile the same block of code ten times
for (i = 0; i < 10; i++)
{
    clMyTimer.Start();
    ...
    //Block of code to profile
    ...
    clMyTimer.Stop();
}

// Get the average execution time of all iterations
u32AverageTimer = clMyTimer.GetAverage();

// Get the execution time from the last iteration
u32LastTimer = clMyTimer.GetCurrent();</pre>
```

Definition in file profile.h.

# 20.120 profile.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] --
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ====
00051 #pragma once
00052
00053 #include "kerneltypes.h"
00055 #include "mark3cfg.h"
00055 #include "ll.h"
00056
00057 namespace Mark3
00058 {
00067 class ProfileTimer
00068 {
00069 public:
00075
          void Init();
00076
00082
          void Start();
00083
00089
          void Stop();
00090
00097
          uint32_t GetAverage();
00098
00106
          uint32_t GetCurrent();
00107
00108 private:
00109
          uint32_t m_u32StartTicks;
00110
          uint32_t m_u32CurrentIteration;
00111
          uint32_t m_u32Cumulative;
00112
          uint16_t m_u16Iterations;
00113
          bool
                    m bActive:
00114 };
00115 } // namespace Mark3
```

- 20.121 /home/moslevin/projects/m3-repo/kernel/src/public/profiling\_results.h File Reference
- 20.122 profiling\_results.h

# 20.123 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::Quantum

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

## **Namespaces**

Mark3

## 20.123.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

# 20.124 quantum.h

```
00001 /*=
00002
00003
00004
00005
00006 |
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ===
00022 #pragma once
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026
00027 #include "thread.h"
00028 #include "timer.h"
00029 #include "timerlist.h"
00030 #include "timerscheduler.h"
00032 #if KERNEL_ROUND_ROBIN
00033 namespace Mark3
00034
00035 class Timer;
00036
00042 class Quantum
00043 {
```

```
00044 public:
           static void Init();
00046
00054
           static void SetInTimer();
00055
00060
           static void ClearInTimer();
00070
           static void Update(Thread* pclTargetThread_);
00071
           static void SetTimerThread(Thread* pclTimerThread_) {
00079
      m_pclTimerThread = pclTimerThread_; }
08000
00084
           static void Cancel();
00085
00086 private:
        static Thread* m_pclActiveThread;
static Thread* m_pclTimerThread;
static uint16_t m_u16TicksRemain;
static bool m_bInTimer;
00087
00088
00089
00091 };
00092 } // namespace Mark3
00093 #endif // #if KERNEL_ROUND_ROBIN
```

# 20.125 /home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h File Reference

Reader-Writer lock implementation.

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

# Classes

class Mark3::ReaderWriterLock

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

## **Namespaces**

• Mark3

## 20.125.1 Detailed Description

Reader-Writer lock implementation.

Definition in file readerwriter.h.

20.126 readerwriter.h 539

# 20.126 readerwriter.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00021 #pragma once
00022
00023 #include "mark3cfg.h"
00024 #include "blocking.h'
00025 #include "mutex.h"
00026
00027 namespace Mark3
00028 {
00040 class ReaderWriterLock
00041 {
00042 public:
          void* operator new(size_t sz, void* pv) { return reinterpret_cast
      ReaderWriterLock*>(pv); }
00044
00050
          void Init();
00051
00058
          void AcquireReader();
00059
00068
          bool AcquireReader(uint32_t u32TimeoutMs_);
00069
00074
          void ReleaseReader();
00075
00082
          void AcquireWriter();
00083
00092
          bool AcquireWriter(uint32_t u32TimeoutMs_);
00093
00098
          void ReleaseWriter();
00099
00100 private:
00107
          bool AcquireReader_i(uint32_t u32TimeoutMs_);
00114
          bool AcquireWriter_i(uint32_t u32TimeoutMs_);
00115
00116
          Mutex
                  m_clGlobalMutex;
00117
          Mutex
                 m clReaderMutex;
          uint8_t m_u8ReadCount;
00119 };
00120
00121 } // namespace Mark3
```

# 20.127 /home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h File Reference

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.

```
#include "kerneltypes.h"
#include "thread.h"
#include "ithreadport.h"
#include "priomap.h"
```

#### Classes

· class Mark3::Scheduler

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

## **Namespaces**

· Mark3

#### **Variables**

- Mark3::Thread \* g\_pclNextMark3::Thread \* g\_pclCurrent
- 20.127.1 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.

#### 20.127.2 Variable Documentation

20.127.2.1 g\_pclCurrent

Mark3::Thread\* g\_pclCurrent

Definition at line 25 of file scheduler.cpp.

20.127.2.2 g\_pclNext

Mark3::Thread\* g\_pclNext

Definition at line 24 of file scheduler.cpp.

20.128 scheduler.h 541

# 20.128 scheduler.h

```
00001 /*==
00002
00003
00004 |
                 1 11
00005 |
00006 |
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00045 #pragma once
00046
00047 #include "kerneltypes.h"
00048 #include "thread.h'
00049 #include "ithreadport.h"
00050 #include "priomap.h"
00051
00052 extern Mark3::Thread* g_pclNext;
00053 extern Mark3::Thread* g_pclCurrent;
00054
00055 namespace Mark3
00056 {
00057 //----
00063 class Scheduler
00064 (
00065 public:
00070
         static void Init();
00071
00078
          static void Schedule();
00079
00086
         static void Add(Thread* pclThread_);
00087
00095
         static void Remove(Thread* pclThread);
00096
00108
          static bool SetScheduler(bool bEnable_);
00109
g_pclCurrent; }
00124 static
         static Thread* GetCurrentThread() { return
          static volatile Thread* GetNextThread() { return
     g_pclNext; }
         static ThreadList* GetThreadList(PORT_PRIO_TYPE uXPriority_) {
     return &m_aclPriorities[uXPriority_]; }
00142
         static ThreadList* GetStopList() { return &m_clStopList; }
          static bool IsEnabled() { return m_bEnabled; }
00150
00156
          static void QueueScheduler() { m_bQueuedSchedule = true; }
00157
00159
         static constexpr auto m_uNumPriorities = size_t {
     KERNEL_NUM_PRIORITIES };
00160
00162
          static bool m bEnabled;
00163
00165
          static bool m_bQueuedSchedule;
00166
00168
          static ThreadList m_clStopList;
00169
00171
          static ThreadList m_aclPriorities[m_uNumPriorities];
00172
00174
          static PriorityMap m_clPrioMap;
00175 };
00176 } // namespace Mark3
```

# 20.129 /home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h File Reference

#### RAII Scheduler Locking.

```
#include "mark3cfg.h"
#include "scheduler.h"
```

#### Classes

· class Mark3::SchedulerGuard

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

## **Namespaces**

· Mark3

# 20.129.1 Detailed Description

RAII Scheduler Locking.

Definition in file schedulerguard.h.

# 20.130 schedulerguard.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00021 #pragma once
00022
00023 #include "mark3cfg.h"
00024 #include "scheduler.h"
00025
00026 namespace Mark3 {
00037 class SchedulerGuard {
00038 public:
00039
         SchedulerGuard()
00040
00041
             m_bSchedState = Scheduler::SetScheduler(false);
00042
         }
00043
00044
          ~SchedulerGuard()
00045
         {
00046
             Scheduler::SetScheduler(m_bSchedState);
00047
00048
00049 private:
00050
         bool m_bSchedState;
00051 };
00052
00053 \} // namespace Mark3
```

# 20.131 /home/moslevin/projects/m3-repo/kernel/src/public/sizeprofile.h File Reference

# 20.132 sizeprofile.h

```
00001 #pragma once
00002
```

# 20.133 /home/moslevin/projects/m3-repo/kernel/src/public/thread.h File Reference

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.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "scheduler.h"
#include "ithreadport.h"
#include "quantum.h"
#include "autoalloc.h"
#include "priomap.h"
```

#### Classes

class Mark3::Thread

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

## **Namespaces**

· Mark3

## **Typedefs**

- using Mark3::ThreadCreateCallout = void(\*)(Thread \*pclThread )
- using Mark3::ThreadExitCallout = void(\*)(Thread \*pclThread
- using Mark3::ThreadContextCallout = void(\*)(Thread \*pclThread )

## 20.133.1 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.

## 20.134 thread.h

```
00001 /*=======
00002
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ======
00034 #pragma once
00035
00036 #include "kerneltypes.h"
00037 #include "mark3cfg.h"
00038
00039 #include "11.h"
00040 #include "threadlist.h"
00041 #include "scheduler.h"
00042 #include "ithreadport.h"
00043 #include "quantum.h"
00044 #include "autoalloc.h"
00044 "include "priomap.h"
00046
00047 namespace Mark3
00048 {
00049 class Thread;
00050
00051 //----
00052 using ThreadCreateCallout = void (*)(Thread* pclThread_);
00053 using ThreadExitCallout = void (*)(Thread* pclThread_);
00054 using ThreadContextCallout = void (*) (Thread* pclThread_);
00055
00056 //---
00064 class Thread : public TypedLinkListNode<Thread>
00065 {
00066 public:
00067
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Thread*>(pv); };
00068
           ~Thread();
00069
00070
          Thread() { m_eState = ThreadState::Invalid; }
00071
          bool IsInitialized() { return (m_eState !=
00077
      ThreadState::Invalid); }
00078
00093
           void Init(K_WORD*
                                       pwStack_,
00094
                     uint16 t
                                       u16StackSize
                     PORT_PRIO_TYPE uXPriority_
00095
00096
                     ThreadEntryFunc pfEntryPoint_,
00097
                                      pvArg_);
                     void*
00098
00116
          static Thread*
00117
          Init(uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
      ThreadEntryFunc pfEntryPoint_, void* pvArg_);
00118
00125
          void Start();
00126
00132
          void Stop();
00133
00134 #if KERNEL_NAMED_THREADS
00135
00143
           void SetName(const char* szName_) { m_szName = szName_; }
          const char* GetName() { return m_szName; }
00149
00150 #endif // #if KERNEL_NAMED_THREADS
00151
00159
           ThreadList* GetOwner(void) { return m_pclOwner; }
           inline ThreadList* GetCurrent(void) { return
00166
      m pclCurrent; }
00174
          PORT_PRIO_TYPE GetPriority(void) { return
      m_uXPriority; }
00181
          PORT_PRIO_TYPE GetCurPriority(void) { return
      m_uXCurPriority; }
00182
00183 #if KERNEL_ROUND_ROBIN
00184
00190
          void SetQuantum(uint16_t u16Quantum_) { m_u16Quantum = u16Quantum_; }
00197
          uint16_t GetQuantum(void) { return m_u16Quantum; }
00198 #endif // #if KERNEL_ROUND_ROBIN
00199
           void SetCurrent(ThreadList* pclNewList ) { m pclCurrent = pclNewList ;
00206
00213
          void SetOwner(ThreadList* pclNewList_) { m_pclOwner = pclNewList_; }
```

20.134 thread.h 545

```
00225
          void SetPriority(PORT_PRIO_TYPE uXPriority_);
00226
00235
          void InheritPriority(PORT_PRIO_TYPE uXPriority_);
00236
00247
          void Exit();
00248
          static void Sleep(uint32_t u32TimeMs_);
00256
00257
00264
          static void Yield(void);
00265
00274
          static void CoopYield(void);
00275
00282
          void SetID(uint8_t u8ID_) { m_u8ThreadID = u8ID_; }
00292
          uint8_t GetID() { return m_u8ThreadID; }
00293
00294 #if KERNEL_STACK_CHECK
00295
00306 uint16_t GetStackSlack();
00307 #endif // #if KERNEL_STACK_CHECK
00308
00309 #if KERNEL_EVENT_FLAGS
00310
00317
          uint16_t GetEventFlagMask() { return m_u16FlagMask; }
00318
          void SetEventFlagMask(uint16_t u16Mask_) { m_u16FlagMask = u16Mask_; }
00323
00324
00330
          void SetEventFlagMode(EventFlagOperation eMode_) {
      m_eFlagMode = eMode_; }
00331
00336
          EventFlagOperation GetEventFlagMode() { return
      m eFlagMode: }
00337 #endif // #if KERNEL_EVENT_FLAGS
00338
00342
          Timer* GetTimer();
00343
          void SetExpired(bool bExpired_);
00350
00351
00357
          bool GetExpired();
00358
00359 #if KERNEL_EXTENDED_CONTEXT
00360
00368
          void* GetExtendedContext() { return m_pvExtendedContext; }
00369
00380
          void SetExtendedContext(void* pvData_) {
      m_pvExtendedContext = pvData_; }
00381 #endif // #if KERNEL_EXTENDED_CONTEXT
00382
00389
           ThreadState GetState() { return m_eState; }
00397
          void SetState(ThreadState eState_) { m_eState = eState_; }
00398
00403
          K_WORD* GetStack() { return m_pwStack; }
00404
00409
          uint16_t GetStackSize() { return m_u16StackSize; }
00410
00411
          friend class ThreadPort:
00412
00413 private:
00420
          static void ContextSwitchSWI (void);
00421
00427
          void SetPriorityBase(PORT_PRIO_TYPE uXPriority_);
00428
00430
          K_WORD* m_pwStackTop;
00431
00433
          K_WORD* m_pwStack;
00434
00436
          uint8_t m_u8ThreadID;
00437
          PORT PRIO TYPE m uXPriority:
00439
00440
          PORT_PRIO_TYPE m_uXCurPriority;
00443
00445
          ThreadState m_eState;
00446
00447 #if KERNEL EXTENDED CONTEXT
00448
          void* m_pvExtendedContext;
00450 #endif // #if KERNEL_EXTENDED_CONTEXT
00451
00452 #if KERNEL_NAMED_THREADS
00453 const Char* m_szName;
00455 #endif // #if KERNEL_NAMED_THREADS
00456
00458
          uint16_t m_u16StackSize;
00459
00461
          ThreadList* m_pclCurrent;
00462
00464
          ThreadList* m_pclOwner;
00465
```

```
ThreadEntryFunc m_pfEntryPoint;
00468
00470
00471
            void* m_pvArg;
00472 #if KERNEL_ROUND_ROBIN
00473 uint16_t m_u16Quantum;
00475 #endif // #if KERNEL_ROUND_ROBIN
00476
00477 #if KERNEL_EVENT_FLAGS
00478
         uint16_t m_u16FlagMask;
00480
00482 EventFlagOperation m_eFlagMode; 00483 #endif // #if KERNEL_EVENT_FLAGS
00484
00486
            Timer m_clTimer;
00487
00489
            bool m_bExpired;
00490 };
00492 } // namespace Mark3
```

# 20.135 /home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h File Reference

Thread linked-list declarations.

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

#### **Classes**

· class Mark3::ThreadList

The ThreadList Class. This class is used for building thread-management facilities, such as schedulers, and blocking objects.

# Namespaces

• Mark3

# 20.135.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

20.136 threadlist.h 547

# 20.136 threadlist.h

```
00001 /*
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00022 #pragma once
00023
00024 #include "kerneltypes.h"
00025 #include "priomap.h"
00026 #include "11.h"
00027
00028 //-
00029 namespace Mark3
00030 {
00031 class Thread;
00032
00038 class ThreadList : public LinkListNode, public TypedCircularLinkList<Thread>
00039 {
00040 public:
00041
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<ThreadList*>(pv); };
00046
          ThreadList();
00047
00054
          void SetPriority(PORT PRIO TYPE uXPriority );
00055
00064
          void SetMapPointer(PriorityMap* pclMap_);
00065
00072
          void Add(Thread* node_);
00073
          void Add(Thread* node_, PriorityMap* pclMap_,
00084
      PORT_PRIO_TYPE uXPriority_);
00085
00093
          void AddPriority(Thread* node_);
00094
00101
          void Remove(Thread* node_);
00102
00109
          Thread* HighestWaiter();
00110
00111 private:
00113
          PORT_PRIO_TYPE m_uXPriority;
00114
00116
          PriorityMap* m_pclMap;
00117 };
00118 } // namespace Mark3
```

# 20.137 /home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h File Reference

Class implementing a doubly-linked list of thread lists.

```
#include "mark3.h"
```

## Classes

· class Mark3::ThreadListList

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

## **Namespaces**

• Mark3

# 20.137.1 Detailed Description

Class implementing a doubly-linked list of thread lists.

Definition in file threadlistlist.h.

# 20.138 threadlistlist.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009
       -[Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00021 #pragma once
00022
00023 #include "mark3.h"
00024
00025 //---
00026 namespace Mark3 {
00027 class ThreadList;
00029 //--
00036 class ThreadListList {
00037 public:
          static void Add(ThreadList* pclThreadList_)
00043
00044
00045
              m_clThreadListList.Add(pclThreadList_);
00046
00047
          static void Remove(ThreadList* pclThreadList_)
00053
00054
00055
              m_clThreadListList.Remove(pclThreadList_);
00057
00062
          static ThreadList* GetHead()
00063
00064
              return m clThreadListList.GetHead();
00065
00066
00067 private:
00068
          static TypedDoubleLinkList<ThreadList>
      m_clThreadListList;
00069 };
00070 } // namespace Mark3
```

# 20.139 /home/moslevin/projects/m3-repo/kernel/src/public/timer.h File Reference

Timer object declarations.

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

## **Classes**

class Mark3::Timer

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

20.140 timer.h 549

## **Namespaces**

• Mark3

# **Typedefs**

using Mark3::TimerCallback = void(\*)(Thread \*pclOwner\_, void \*pvData\_)

#### **Variables**

- static constexpr auto Mark3::uMaxTimerTicks = uint32\_t { 0x7FFFFFF }
   Maximum value to set.
- static constexpr auto Mark3::uTimerTicksInvalid = uint32\_t { 0 }
- static constexpr auto Mark3::uTimerFlagOneShot = uint8\_t { 0x01 }

Timer is one-shot.

static constexpr auto Mark3::uTimerFlagActive = uint8\_t { 0x02 }

Timer is currently active.

static constexpr auto Mark3::uTimerFlagCallback = uint8\_t { 0x04 }

Timer is pending a callback.

static constexpr auto Mark3::uTimerFlagExpired = uint8\_t { 0x08 }

Timer is actually expired.

## 20.139.1 Detailed Description

Timer object declarations.

Definition in file timer.h.

# 20.140 timer.h

```
00001
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00021 #pragma once
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "11.h"
00026
00027 namespace Mark3
00028 4
00029 class Thread;
00030
00031 //---
00032 static constexpr auto uMaxTimerTicks
00033 static constexpr auto uTimerTicksInvalid = uint32_t { 0 };
00034 static constexpr auto uTimerFlagOneShot = uint8_t { 0x01 };
00035 static constexpr auto uTimerFlagActive = uint8_t { 0x02 };
00035 static constexpr auto uTimerFlagActive
00036 static constexpr auto uTimerFlagCallback = uint8_t { 0x04 };
00037 static constexpr auto uTimerFlagExpired = uint8_t { 0x08 };
```

```
00050 using TimerCallback = void (*)(Thread* pclOwner_, void* pvData_);
00051
00052 //----
00053 class TimerList;
00054 class TimerScheduler;
00055 class Quantum;
00056
00057 //----
00068 class Timer : public TypedLinkListNode<Timer>
00069 {
00070 public:
00071
          void* operator new(size_t sz, void* pv) { return reinterpret_cast<Timer*>(pv); }
00072
00073
          Timer();
00079
00080
00085
          void Init();
00086
00097
          void Start(bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback pfCallback_, void*
void pvData_);
00106
          void Start();
00107
00113
          void Stop();
00114
00115 private:
00116
          friend class TimerList;
00117
          void SetInitialized() { m u8Initialized =
00121
      m_uTimerInitCookie; }
00122
00127
         bool IsInitialized(void) { return (m_u8Initialized ==
      m_uTimerInitCookie); }
00128
          static inline uint32_t SecondsToTicks(uint32_t x) { return (x) * 1000; }
static inline uint32_t MSecondsToTicks(uint32_t x) { return (x); }
00129
00130
00131
          static inline uint32_t USecondsToTicks(uint32_t x) { return ((x + 999) / 1000); }
00132
00133
          static constexpr auto m_uTimerInvalidCookie = uint8_t { 0x3C };
00134
          static constexpr auto m_uTimerInitCookie
                                                        = uint8_t { 0xC3 };
00135
00137
          uint8_t m_u8Initialized;
00138
00140
          uint8_t m_u8Flags;
00141
          TimerCallback m_pfCallback;
00143
00144
00146
          uint32_t m_u32Interval;
00147
00149
          uint32_t m_u32TimeLeft;
00150
00152
          Thread* m_pclOwner;
00153
00155
          void* m pvData;
00156 };
00157 } // namespace Mark3
```

# 20.141 /home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h File Reference

Timer list declarations.

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

## Classes

· class Mark3::TimerList

the TimerList class. This class implements a doubly-linked-list of timer objects.

20.142 timerlist.h 551

## **Namespaces**

· Mark3

## 20.141.1 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.

## 20.142 timerlist.h

```
00001 /*======
00002
00003
00004
00005
00007
80000
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00024 #pragma once
00025
00026 #include "kerneltypes.h"
00027 #include "mark3cfg.h"
00029 #include "mutex.h"
00030
00031 namespace Mark3
00032 {
00033 class Timer;
00034
00040 class TimerList : public TypedDoubleLinkList<Timer>
00041 {
00042 public:
00048
          void Init();
00049
00056
          void Add(Timer* pclListNode_);
00057
00064
         void Remove(Timer* pclLinkListNode_);
00065
00071
         void Process();
00072
00073 private:
         uint32_t m_u32NextWakeup;
00076
00078
          bool m_bTimerActive;
00079
00081
          Mutex m_clMutex;
00082 };
00083 } // namespace Mark3
```

# 20.143 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::TimerScheduler

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

## **Namespaces**

Mark3

## 20.143.1 Detailed Description

Timer scheduler declarations.

Definition in file timerscheduler.h.

# 20.144 timerscheduler.h

```
00001 /*==========
00002
00003
00004 1
00005
00006
00007
80000
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ==
00021 #pragma once
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 #include "11.h"
00027 #include "timer.h"
00028 #include "timerlist.h"
00029
00030 namespace Mark3
00031 {
00032 //---
00038 class TimerScheduler
00039 {
00040 public:
00046 static void Init() { m_clTimerList.Init(); }
00054 static void Add(Timer* pclListNode_) { m_clTimerList.
      Add(pclListNode_); }
00062
          static void Remove(Timer* pclListNode_) { m_clTimerList.
      Remove(pclListNode_); }
00070
          static void Process() { m_clTimerList.Process(); }
00071
00072 private:
00074
          static TimerList m_clTimerList;
00075 };
00076 } // namespace Mark3
```

# 20.145 /home/moslevin/projects/m3-repo/kernel/src/quantum.cpp File Reference

Thread Quantum Implementation for Round-Robin Scheduling.

```
#include "mark3.h"
```

20.146 quantum.cpp 553

## 20.145.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

# 20.146 quantum.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
00008
00009
      --[Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =======
00022 #include "mark3.h"
00023
00024 #if KERNEL_ROUND_ROBIN
00025 namespace Mark3
00026 {
00027 //--
00028 uint16_t Quantum::m_u16TicksRemain;
00029 Thread*
               Quantum::m_pclActiveThread;
00030 Thread* Quantum::m_pclTimerThread;
00031 bool
                Quantum::m_bInTimer;
00032
00033 //--
00034 void Quantum::SetInTimer()
00035 {
00036
          const auto cs = CriticalGuard{};
00037
          m_bInTimer = true;
00038
00039
           // Timer is active
00040
          if (m_u16TicksRemain) {
00041
               m_u16TicksRemain--;
00042
00043 }
00044
00045 //---
00046 void Quantum::ClearInTimer()
00047 {
00048
           const auto cs = CriticalGuard{};
00049
          m_bInTimer = false;
00050
00051
          \ensuremath{//} Timer expired - Pivot the thread list.
          if (m_pclActiveThread && (!m_ul6TicksRemain)) {
    auto* pclThreadList = m_pclActiveThread->GetCurrent();
00052
00053
00054
               if (pclThreadList->GetHead() != pclThreadList->GetTail()) {
00055
                   pclThreadList->PivotForward();
00056
00057
               m_pclActiveThread = nullptr;
00058
          }
00059 }
00060
00061 //--
00062 void Quantum::Update(Thread* pclTargetThread_)
00063 {
           // Don't cancel the current RR interval if we're being interrupted by
00064
00065
          // the timer thread, or are in the middle of running the timer thread.
          // OR if the thread list only has one thread
00066
00067
          auto* pclThreadList = pclTargetThread_->GetCurrent();
00068
           if ((pclThreadList->GetHead() == pclThreadList->GetTail()) || (pclTargetThread_ ==
      m_pclTimerThread)
              || (pclTargetThread_ == m_pclActiveThread) ||
00069
      m_bInTimer) {
00070
              return;
00071
00072
00073
          \ensuremath{//} Update with a new thread and timeout.
00074
          m_pclActiveThread = pclTargetThread_;
m_ul6TicksRemain = pclTargetThread_->GetQuantum();
00075
00076 }
00077
```

# 20.147 /home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp File Reference

Reader-writer lock implementation.

```
#include "mark3.h"
```

#### **Namespaces**

Mark3

# 20.147.1 Detailed Description

Reader-writer lock implementation.

Definition in file readerwriter.cpp.

# 20.148 readerwriter.cpp

```
00001 /*==
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform]-
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =========
00020 #include "mark3.h"
00021 namespace Mark3
00022 {
00023 //---
00024 void ReaderWriterLock::Init()
00025 {
00026
         m_u8ReadCount = 0;
         m_clGlobalMutex.Init();
00027
00028
         m_clReaderMutex.Init();
00029 }
00030
00031 //--
00032 void ReaderWriterLock::AcquireReader()
00033 {
00034
          AcquireReader_i(0);
00035 }
00036
00037 //--
00038 bool ReaderWriterLock::AcquireReader(uint32_t u32TimeoutMs_)
00039 {
00040
          return AcquireReader_i(u32TimeoutMs_);
00041 }
00042
```

```
00044 void ReaderWriterLock::ReleaseReader()
00045 {
00046
         m_clReaderMutex.Claim();
00047
         m_u8ReadCount--;
00048 if (0 == m_u8ReadCount) {
00049 m_clGlobalMutex.Relea
       m_clGlobalMutex.Release();
}
m_clReaderMutex.Release();
00050
00051
00052 }
00053
00054 //--
00055 void ReaderWriterLock::AcquireWriter()
00056 {
00057
         AcquireWriter_i(0);
00058 }
00059
00060 //--
00061 bool ReaderWriterLock::AcquireWriter(uint32_t u32TimeoutMs_)
00063
         return AcquireWriter_i(u32TimeoutMs_);
00064 }
00065
00066 //----
00067 void ReaderWriterLock::ReleaseWriter()
00069
         m_clGlobalMutex.Release();
00070 }
00071
00072 //----
00073 bool ReaderWriterLock::AcquireReader i(uint32 t u32TimeoutMs)
00074 {
00075
         auto rc = true;
        if (!m_clReaderMutex.Claim(u32TimeoutMs_)) {
00076
       return false;
00077
00078
00079
08000
       m_u8ReadCount++;
rc = m_clGlobalMutex.Claim(u32TimeoutMs_);
00083
00084
00085
       m_clReaderMutex.Release();
00086
         return rc;
00087 }
00088
00089 //---
00090 bool ReaderWriterLock::AcquireWriter_i(uint32_t u32TimeoutMs_)
00091 {
00092
         return m_clGlobalMutex.Claim(u32TimeoutMs_);
00093 }
00094 } // namespace Mark3
```

# 20.149 /home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp File Reference

Strict-Priority + Round-Robin thread scheduler implementation.

```
#include "mark3.h"
```

#### **Namespaces**

• Mark3

#### **Variables**

- Mark3::Thread \* g pclNext
- Mark3::Thread \* g\_pclCurrent

# 20.149.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

#### 20.149.2 Variable Documentation

#### 20.149.2.1 g\_pclCurrent

```
Mark3::Thread* g_pclCurrent
```

Definition at line 25 of file scheduler.cpp.

#### 20.149.2.2 g\_pclNext

```
Mark3::Thread* g_pclNext
```

Definition at line 24 of file scheduler.cpp.

# 20.150 scheduler.cpp

```
00001 /*=
00002
00003
00004
00005
00006
00007
00008
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00022 #include "mark3.h"
00023
00024 Mark3::Thread* q_pclNext;
00025 Mark3::Thread* g_pclCurrent;
00026
00027 namespace Mark3
00028 {
00029 bool
                    Scheduler::m_bEnabled;
00030 bool
                    Scheduler::m_bQueuedSchedule;
00031 ThreadList Scheduler::m_clStopList;
00032 ThreadList Scheduler::m_aclPriorities[
       KERNEL_NUM_PRIORITIES];
00033 PriorityMap Scheduler::m_clPrioMap;
00034
00035 //--
00036 void Scheduler::Init()
00037 {
           for (size_t i = 0; i < m_uNumPriorities; i++) {
    m_aclPriorities[i].SetPriority(i);</pre>
00038
00039
00040
               m_aclPriorities[i].SetMapPointer(&
     m_clPrioMap);
00041
           }
00042 }
```

```
00043
00044 //--
00045 void Scheduler::Schedule()
00046 {
00047
         auto uXPrio = m_clPrioMap.HighestPriority();
00048
        if (0 == uXPrio) {
            Kernel::Panic(PANIC_NO_READY_THREADS);
00050
00051
        // Priorities are one-indexed
00052
        uXPrio--;
00053
        // Get the thread node at this priority.
00054
        g_pclNext = m_aclPriorities[uXPrio].GetHead();
00055
00056 }
00057
00058 //--
00059 void Scheduler::Add(Thread* pclThread_)
00060 {
00061
         KERNEL_ASSERT(pclThread_ != nullptr);
00062
00063
         m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00064 }
00065
00066 //--
00067 void Scheduler::Remove(Thread* pclThread_)
00069
         KERNEL_ASSERT(pclThread_ != nullptr);
00070
00071
         m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00072 }
00073
00074 //-
00075 bool Scheduler::SetScheduler(bool bEnable_)
00076 {
00077
         const auto cs = CriticalGuard{};
00078
auto bRet
                        = m_bEnabled;
        m_bQueuedSchedule = false;
Thread::Yield();
00084
00085
       }
return bRet;
00086
00088 } // namespace Mark3
```

# 20.151 /home/moslevin/projects/m3-repo/kernel/src/thread.cpp File Reference

Platform-Independent thread class Definition.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "mark3.h"
```

# **Namespaces**

Mark3

# 20.151.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

# 20.152 thread.cpp

```
00001 /*======
00003
00004
00005
00006 1
00007
00008
00009 -- [Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 =====
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "mark3.h"
00026
00027 namespace Mark3
00028 {
00029 //---
00030 Thread::~Thread()
00031 {
00032
                    // On destruction of a thread located on a stack,
                   // on destruction of a tributal took of a state of a st
00033
00034
00035
00036
00037
                   if (ThreadState::Stop == m_eState) {
                           const auto cs = CriticalGuard{};
00038
                           m_pclCurrent->Remove(this);
00039
                           m_pclCurrent = nullptr;
m_pclOwner = nullptr;
m_eState = ThreadState::Exit;
00040
00041
00042
00043
                   } else if (ThreadState::Exit != m_eState) {
00044
                           Kernel::Panic(PANIC_RUNNING_THREAD_DESCOPED);
00045
00046 }
00047
00048 //--
00049 void Thread::Init(
00050
                   K_WORD* pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
           ThreadEntryFunc pfEntryPoint_, void* pvArg_)
00051 {
00052
                    static auto u8ThreadID = uint8_t { 0 };
00053
00054
                   KERNEL_ASSERT (pwStack_);
00055
                   KERNEL_ASSERT (pfEntryPoint_);
00056
00057
                   ClearNode();
00058
00059
                   m_u8ThreadID = u8ThreadID++;
00060
00061
                    // Initialize the thread parameters to their initial values.
00062
                   m_pwStack = pwStack_;
                   m_pwStackTop = PORT_TOP_OF_STACK(pwStack_, u16StackSize_);
00063
00064
00065
                   m_u16StackSize = u16StackSize_;
00066
                   m_uXPriority
                                                  = uXPriority_;
00067
                   m_uXCurPriority = m_uXPriority;
                   m_pfEntryPoint = pfEntryPoint_;
00068
                                                   = pvArg_;
00069
                   m_pvArg
00070
00071 #if KERNEL_NAMED_THREADS
00072
                  m_szName = nullptr;
00073 #endif
00074 #if KERNEL_ROUND_ROBIN
00075
                  m_u16Quantum = THREAD_QUANTUM_DEFAULT;
00076 #endif
00077
00078
                   m_clTimer.Init();
00079
00080
                    // Call CPU-specific stack initialization
00081
                   ThreadPort::InitStack(this);
00082
                    // Add to the global "stop" list.
00083
                   { // Begin critical section
   const auto cs = CriticalGuard{};
00084
00085
00086
                           m_pclOwner = Scheduler::GetThreadList(
           m_uXPriority);
00087
                           m_pclCurrent = Scheduler::GetStopList();
00088
                           m_eState
                                                     = ThreadState::Stop;
00089
                           m_pclCurrent->Add(this);
00090
                   } // End critical section
```

20.152 thread.cpp 559

```
00091
00092 #if KERNEL_THREAD_CREATE_CALLOUT
00093
         ThreadCreateCallout pfCallout =
     Kernel::GetThreadCreateCallout();
       if (nullptr != pfCallout) {
00094
00095
             pfCallout(this);
00097 #endif
00098 }
00099
00100 //----
00101 Thread* Thread::Init(uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
      ThreadEntryFunc pfEntryPoint_, void* pvArg_)
00102 {
00103
          auto* pclNew = AutoAlloc::NewObject<Thread, AutoAllocType::Thread>();
          auto* pwStack = static_cast<K_WORD*>(AutoAlloc::NewRawData(u16StackSize_));
00104
00105
          pclNew->Init(pwStack, u16StackSize_, uXPriority_, pfEntryPoint_, pvArg_);
00106
          return pclNew;
00107 }
00108
00109 //--
00110 void Thread::Start(void)
00111 {
00112
          KERNEL ASSERT(IsInitialized()):
00113
          // Remove the thread from the scheduler's "stopped" list, and add it
00114
00115
         // to the scheduler's ready list at the proper priority.
00116
00117
         const auto cs = CriticalGuard{};
         Scheduler::GetStopList() ->Remove(this);
00118
00119
         Scheduler::Add(this);
00120
          m_pclOwner = Scheduler::GetThreadList(
     m_uXPriority);
        m_pclCurrent = m_pclOwner;
m_eState = ThreadState::Ready;
00121
         m_eState
00122
00123
00124 #if KERNEL_ROUND_ROBIN
       if (Kernel::IsStarted()) {
00126
              if (GetCurPriority() >= Scheduler::GetCurrentThread()->
     GetCurPriority()) {
00127
                 // Deal with the thread Quantum
00128
                 Quantum:: Update(this);
00129
             }
00130
00131 #endif
00132
00133
          if (Kernel::IsStarted()) {
00134
             if (GetCurPriority() >= Scheduler::GetCurrentThread()->
     GetCurPriority()) {
00135
                Thread::Yield();
00136
00137
00138 }
00139
00140 //--
00141 void Thread::Stop()
00142 {
00143
          KERNEL_ASSERT(IsInitialized());
00144
          auto bReschedule = false;
00145
00146
          if (ThreadState::Stop == m_eState) {
00147
             return;
00148
         }
00149
00150
          { // Begin critical section
00151
              const auto cs = CriticalGuard{};
00152
              // If a thread is attempting to stop itself, ensure we call the scheduler
00153
00154
              if (this == Scheduler::GetCurrentThread()) {
00155
                  bReschedule = true;
00156
         #if KERNEL_ROUND_ROBIN
00157
                  // Cancel RR scheduling
00158
                  Quantum::Cancel();
         #endif
00159
00160
             }
00161
00162
              // Add this thread to the stop-list (removing it from active scheduling)
00163
              // Remove the thread from scheduling
00164
              if (ThreadState::Ready == m_eState)
                  Scheduler::Remove(this);
00165
              } else if (ThreadState::Blocked == m eState) {
00166
00167
                 m_pclCurrent->Remove(this);
00168
00169
             m_pclOwner = Scheduler::GetStopList();
m_pclCurrent = m_pclOwner;
00170
00171
00172
              m_pclOwner->Add(this);
```

```
m_eState = ThreadState::Stop;
00174
00175
               // Just to be safe - attempt to remove the thread's timer
               // from the timer-scheduler (does no harm if it isn't
00176
               // in the timer-list)
00177
00178
               TimerScheduler::Remove(&m_clTimer);
00179
          } // End Critical Section
00180
00181
          if (bReschedule) {
00182
               Thread::Yield();
          }
00183
00184 }
00185
00186 //---
00187 void Thread::Exit()
00188 {
          KERNEL ASSERT(IsInitialized());
00189
00190
00191
          auto bReschedule = false;
00192
00193
          if (ThreadState::Exit == m_eState) {
00194
               return;
          }
00195
00196
00197
          { // Begin critical section
              const auto cs = CriticalGuard{};
00198
00199
00200
               // If this thread is the actively-running thread, make sure we run the
00201
               // scheduler again.
               if (this == Scheduler::GetCurrentThread()) {
00202
00203
                   bReschedule = true;
00204 #if KERNEL_ROUND_ROBIN
00205
                  // Cancel RR scheduling
00206
                   Quantum::Cancel();
00207 #endif
              }
00208
00209
00210
               // Remove the thread from scheduling
00211
              if (ThreadState::Ready == m_eState) {
00212
                   Scheduler::Remove(this);
00213
              } else if ((ThreadState::Blocked == m_eState) || (
     ThreadState::Stop == m_eState)) {
    m_pclCurrent->Remove(this);
00214
00215
00216
00217
              m_pclCurrent = nullptr;
              m_pclOwner = nullptr;
m_eState = ThreadState::Exit;
00218
00219
              m_eState
00220
00221
              // We've removed the thread from scheduling, but interrupts might
00222
              // trigger checks against this thread's currently priority before
00223
               // we get around to scheduling new threads. As a result, set the
00224
               \ensuremath{//} priority to idle to ensure that we always wind up scheduling
00225
              // new threads.
00226
               m_uXCurPriority = 0;
00227
              m_uXPriority
                               = 0;
00228
00229
               // Just to be safe - attempt to remove the thread's timer
00230
               // from the timer-scheduler (does no harm if it isn't
               // in the timer-list)
00231
               TimerScheduler::Remove(&m clTimer):
00232
          } // End Critical Section
00233
00234
00235 #if KERNEL_THREAD_EXIT_CALLOUT
00236
          ThreadExitCallout pfCallout = Kernel::GetThreadExitCallout
          if (nullptr != pfCallout) {
00237
00238
              pfCallout(this);
00239
00240 #endif
00241
00242
          if (bReschedule) {
              // Choose a new "next" thread if we must
Thread::Yield();
00243
00244
00245
          }
00246 }
00247
00248 //----
00249 void Thread::Sleep(uint32_t u32TimeMs_)
00250 {
                              = Semaphore {};
= g_pclCurrent->GetTimer();
00251
          auto clSemaphore
00252
          auto* pclTimer
          auto lTimerCallback = [](Thread* /*pcl0wner*/, void* pvData_) {
   auto* pclSemaphore = static_cast<Semaphore*>(pvData_);
00253
00254
00255
              pclSemaphore->Post();
00256
          };
00257
```

20.152 thread.cpp 561

```
00258
            / Create a semaphore that this thread will block on
00259
          clSemaphore.Init(0, 1);
00260
00261
          // Create a one-shot timer that will call a callback that posts the
00262
          // semaphore, waking our thread.
00263
          pclTimer->Init();
00264
          pclTimer->Start(false, u32TimeMs_, lTimerCallback, &clSemaphore);
00265
00266
          clSemaphore.Pend();
00267 }
00268
00269 #if KERNEL STACK CHECK
00270 //
00271 uint16_t Thread::GetStackSlack()
00272 {
00273
          KERNEL_ASSERT(IsInitialized());
00274
00275
          auto wBottom = uint16 t { 0 };
          auto wTop = static_cast<uint16_t>((m_u16StackSize - 1) / sizeof(
00276
      K_ADDR));
00277
         auto wMid
                      = static_cast<uint16_t>(((wTop + wBottom) + 1) / 2);
00278
00279
         { // Begin critical section
              const auto cs = CriticalGuard{};
00280
00281
00282
              \ensuremath{//} Logarithmic bisection - find the point where the contents of the
00283
              // stack go from 0xFF's to non 0xFF. Not Definitive, but accurate enough
00284
              while ((wTop - wBottom) > 1) {
00285 #if PORT_STACK_GROWS_DOWN
                  if (m_pwStack[wMid] != static_cast<K_WORD>(-1))
00286
00287 #else
00288
                  if (m_pwStack[wMid] == static_cast<K_WORD>(-1))
00289 #endif
00290
                  {
00292
                      wTop = wMid;
00293
                  } else {
00294
                      wBottom = wMid;
00295
00296
                  wMid = (wTop + wBottom + 1) / 2;
00297
00298
         } // End Critical Section
00299
          return wMid * sizeof(K ADDR):
00300
00301 }
00302 #endif
00303
00304 //----
00305 void Thread::Yield()
00306 {
00307
          const auto cs = CriticalGuard{};
00308
          // Run the scheduler
00309
         if (Scheduler::IsEnabled()) {
00310
              Scheduler::Schedule();
00311
              // Only switch contexts if the new task is different than the old task if (g_pclCurrent != g_pclNext) {
00312
00313
00314 #if KERNEL_ROUND_ROBIN
00315
                  Quantum:: Update(g_pclNext);
00316 #endif
00317
                  Thread::ContextSwitchSWI();
00318
              }
00319
          } else {
00320
             Scheduler::QueueScheduler();
00321
00322 }
00323
00324 //---
00325 void Thread::CoopYield(void)
00326 {
00327
          g_pclCurrent->GetCurrent()->PivotForward();
00328
          Yield();
00329 }
00330
00331 //--
00332 void Thread::SetPriorityBase(PORT_PRIO_TYPE /*uXPriority_*/)
00333 {
00334
          KERNEL_ASSERT(IsInitialized());
00335
00336
          GetCurrent()->Remove(this);
          SetCurrent(Scheduler::GetThreadList(
m_uXPriority));
00338 Ge+0
          GetCurrent()->Add(this);
00339 }
00340
00341 //---
00342 void Thread::SetPriority(PORT_PRIO_TYPE uXPriority_)
00343 {
```

562 File Documentation

```
00344
          KERNEL_ASSERT(IsInitialized());
00345
         auto bSchedule = false;
00346
00347
         { // Begin critical section
00348
             const auto cs = CriticalGuard{};
00349
              // If this is the currently running thread, it's a good idea to reschedule
00350
00351
             // Or, if the new priority is a higher priority than the current thread's.
00352
              if ((this == g_pclCurrent) || (uXPriority_ > g_pclCurrent->
     GetPriority())) {
00353
                 bSchedule = true;
00354
          #if KERNEL_ROUND_ROBIN
00355
                 Quantum::Cancel();
00356
         #endif
00357
00358
              Scheduler::Remove(this);
00359
             m_uXCurPriority = uXPriority_;
m_uXPriority = uXPriority_;
00360
             m_uXPriority
00361
00362
00363
             Scheduler::Add(this);
        } // End critical section
00364
00365
         if (bSchedule) {
00366
00367
             if (Scheduler::IsEnabled()) {
00368
                 { // Begin critical section
00369
                      const auto cs = CriticalGuard{};
00370
                     Scheduler::Schedule();
00371 #if KERNEL_ROUND_ROBIN
                    Quantum::Update(g_pclNext);
00372
00373 #endif
00374
                  } // End critical sectin
00375
                 Thread::ContextSwitchSWI();
00376
             } else {
00377
                Scheduler::QueueScheduler();
             }
00378
00379
         }
00380 }
00381
00382 //---
00383 void Thread::InheritPriority(PORT_PRIO_TYPE uXPriority_)
00384 {
00385
         KERNEL ASSERT (IsInitialized()):
00386
00387
          SetOwner(Scheduler::GetThreadList(uXPriority_));
00388
         m_uXCurPriority = uXPriority_;
00389 }
00390
00391 //---
00392 void Thread::ContextSwitchSWI()
00393 {
00394
          // Call the context switch interrupt if the scheduler is enabled.
00395
          if (Scheduler::IsEnabled()) {
00396 #if KERNEL_STACK_CHECK
             if (g_pclCurrent && (g_pclCurrent->GetStackSlack() <=</pre>
00397
     Kernel::GetStackGuardThreshold())) {
00398
                Kernel::Panic(PANIC_STACK_SLACK_VIOLATED);
00399
              }
00400 #endif
00401 #if KERNEL_CONTEXT_SWITCH_CALLOUT
             auto pfCallout = Kernel::GetThreadContextSwitchCallout();
00402
              if (nullptr != pfCallout) {
00403
00404
                 pfCallout(g_pclCurrent);
00405
00406 #endif
00407
             KernelSWI::Trigger();
00408
         }
00409 }
00410
00411 //---
00412 Timer* Thread::GetTimer()
00413 {
00414
         KERNEL_ASSERT(IsInitialized());
00415
         return &m_clTimer;
00416 }
00417 //--
00418 void Thread::SetExpired(bool bExpired_)
00419 {
         KERNEL ASSERT(IsInitialized());
00420
00421
         m bExpired = bExpired;
00422 }
00423
00424 //--
00425 bool Thread::GetExpired()
00426 {
         KERNEL ASSERT(IsInitialized());
00427
00428
         return m bExpired:
```

```
00429 }
00430 } // namespace Mark3
```

## 20.153 /home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp File Reference

Thread linked-list definitions.

```
#include "mark3.h"
#include "threadlistlist.h"
```

### **Namespaces**

· Mark3

### 20.153.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

### 20.154 threadlist.cpp

```
00001 /*===========
00002
00004
00005 |
00006 |_
00007
80000
00009 -- [Mark3 Realtime Platform] -
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00022 #include "mark3.h"
00023 #include "threadlistlist.h"
00024 namespace Mark3
00025 {
00026 //----
00027 ThreadList::ThreadList()
00028 : m_uXPriority(0)
00029 . m_pclMap(nullpt
00029
         , m_pclMap(nullptr)
00031 }
00032
00033 //---
00034 void ThreadList::SetPriority(PORT_PRIO_TYPE uXPriority_)
00035 {
00036
         m_uXPriority = uXPriority_;
00037 }
00038
00039 //---
00040 void ThreadList::SetMapPointer(PriorityMap* pclMap_)
00041 {
00042
         KERNEL_ASSERT (pclMap_ != nullptr);
00043
        m_pclMap = pclMap_;
00044 }
00045
00046 //--
00047 void ThreadList::Add(Thread* pclThread_)
00048 {
         KERNEL_ASSERT(pclThread_ != nullptr);
```

564 File Documentation

```
00051
          // If list was empty, add the object for global threadlist tracking
00052
          if (!GetHead()) {
              ThreadListList::Add(this);
00053
00054
00055
          TypedCircularLinkList<Thread>::Add(pclThread_);
          PivotForward();
00056
00057
          // We've specified a bitmap for this threadlist
00058
          if (nullptr != m_pclMap) {
    // Set the flag for this priority level
00059
00060
00061
              m_pclMap->Set (m_uXPriority);
00062
          }
00063 }
00064
00065 //---
00066 void ThreadList::AddPriority(Thread* pclThread_)
00067 {
00068
          KERNEL_ASSERT(node_ != nullptr);
          auto* pclCurr = GetHead();
if (nullptr == pclCurr) {
00069
00070
00071
              Add (pclThread_);
00072
              return;
00073
00074
          auto uXHeadPri = pclCurr->GetCurPriority();
00075
          auto* pclTail = GetTail();
00076
00077
          // Set the threadlist's priority level, flag pointer, and then add the
          // thread to the threadlist
00078
00079
          auto uXPriority = pclThread_->GetCurPriority();
08000
          do {
00081
              if (uXPriority > pclCurr->GetCurPriority()) {
00082
                   break;
00083
00084
              pclCurr = pclCurr->GetNext();
          } while (pclCurr != pclTail);
00085
00086
          // Insert pclNode before pclCurr in the linked list.
00088
          InsertNodeBefore(pclThread_, pclCurr);
00089
00090
          \ensuremath{//} If the priority is greater than current head, reset
00091
          // the head pointer.
00092
          if (uXPriority > uXHeadPri) {
00093
              SetHead(pclThread_);
00094
              SetTail(GetHead()->GetPrev());
00095
          } else if (pclThread_->GetNext() == GetHead()) {
00096
             SetTail(pclThread_);
00097
          }
00098 }
00099
00100 //--
00101 void ThreadList::Add(Thread* node_, PriorityMap* pclMap_,
      PORT_PRIO_TYPE uXPriority_)
00102 {
00103
          \ensuremath{//} Set the threadlist's priority level, flag pointer, and then add the
00104
          // thread to the threadlist
          SetPriority(uXPriority_);
00106
          SetMapPointer(pclMap_);
00107
          Add (node_);
00108 }
00109
00110 //-
00111 void ThreadList::Remove(Thread* node_)
00112 {
00113
           // Remove the thread from the list
00114
          TypedCircularLinkList<Thread>::Remove(node_);
00115
00116
          // If the list is empty...
          if (nullptr == GetHead()) {
00117
00118
               // No more threads - remove this object from global threadlist tracking
00119
              ThreadListList::Remove(this);
00120
              if (nullptr != m_pclMap) {
                   // Clear the bit in the bitmap at this priority level
00121
00122
                  m_pclMap->Clear(m_uXPriority);
00123
              }
00124
          }
00125 }
00126
00127 //--
00128 Thread* ThreadList::HighestWaiter()
00129 {
          return GetHead();
00131
00132 } // namespace Mark3
```

## 20.155 /home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp File Reference

Class implementing a doubly-linked list of thread lists.

```
#include "mark3.h"
#include "threadlistlist.h"
```

#### **Namespaces**

• Mark3

### 20.155.1 Detailed Description

Class implementing a doubly-linked list of thread lists.

Definition in file threadlistlist.cpp.

## 20.156 threadlistlist.cpp

```
00001 /
00002
00004
00005 |
00006 |
00007
80000
00009 -- [Mark3 Realtime Platform]
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00021 #include "mark3.h"
00022 #include "threadlistlist.h"
00023
00024 namespace Mark3 {
00025 TypedDoubleLinkList<ThreadList> ThreadListList::m_clThreadListList;
00026 } // namespace Mark3
```

## 20.157 /home/moslevin/projects/m3-repo/kernel/src/timer.cpp File Reference

Timer implementations.

```
#include "mark3.h"
```

#### **Namespaces**

• Mark3

566 File Documentation

### 20.157.1 Detailed Description

Timer implementations.

Definition in file timer.cpp.

### 20.158 timer.cpp

```
00001 /*=
00002
00003
00004
00005
00006
00007
00009 -- [Mark3 Realtime Platform] -
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 -----
00022 #include "mark3.h"
00023
00024 namespace Mark3
00025
00026 TimerList TimerScheduler::m_clTimerList;
00027
00028 //-
00029 Timer::Timer()
00030 {
00031
          m_u8Initialized = m_uTimerInvalidCookie;
                          = 0;
00032
          m_u8Flags
00033 }
00034
00035 //---
00036 void Timer::Init()
00037 {
00038
          if (IsInitialized()) {
               KERNEL_ASSERT((m_u8Flags & uTimerFlagActive) == 0);
00039
00040
00041
00042
          ClearNode();
00043
          m_u32Interval = 0;
00044
          m_u32TimeLeft = 0;
00045
          m_u8Flags
00046
00047
          SetInitialized();
00048 }
00049
00050 //--
00051 void Timer::Start(bool bRepeat_, uint32_t u32IntervalMs_, 
TimerCallback pfCallback_, void* pvData_)
00052 {
00053
          KERNEL_ASSERT(IsInitialized());
00054
00055
          if ((m_u8Flags & uTimerFlagActive) != 0) {
00056
              return;
00057
00058
00059
          m_u32Interval = u32IntervalMs_;
          m_pfCallback = pfCallback_;
m_pvData = pvData_;
00060
00061
00062
00063
          if (!bRepeat_) {
              m_u8Flags = uTimerFlagOneShot;
00064
00065
          } else {
00066
              m_u8Flags = 0;
00067
          }
00068
00069
          Start():
00070 }
00071
00072 //--
00073 void Timer::Start()
00074 {
00075
          KERNEL_ASSERT(IsInitialized());
00076
00077
           if ((m_u8Flags & uTimerFlagActive) != 0) {
00078
              return;
```

```
00079
08000
00081
          m_pclOwner = Scheduler::GetCurrentThread();
00082
          TimerScheduler::Add(this);
00083 }
00084
00085 //--
00086 void Timer::Stop()
00087 {
          KERNEL_ASSERT(IsInitialized());
00088
00089
          if ((m_u8Flags & uTimerFlagActive) == 0) {
00090
             return;
00091
00092
          TimerScheduler::Remove(this);
00093 }
00094 } // namespace Mark3
```

### 20.159 /home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp File Reference

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

```
#include "mark3.h"
```

### **Namespaces**

Mark3

### 20.159.1 Detailed Description

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

Definition in file timerlist.cpp.

### 20.160 timerlist.cpp

```
00001 /
00002
00003
00004
00005
00006
00007
80000
       -[Mark3 Realtime Platform]
00010
00011 Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
00012 See license.txt for more information
00013 ========
00023 #include "mark3.h"
00024 namespace Mark3
00025 {
00026 //---
00027 void TimerList::Init(void)
00028 {
00029
         m bTimerActive = false;
         m_u32NextWakeup = 0;
00030
00031
         m_clMutex.Init();
00032 }
00033
00034 //--
00035 void TimerList::Add(Timer* pclListNode_)
00036 {
00037
         KERNEL_ASSERT(nullptr != pclListNode_);
```

568 File Documentation

```
00038
          auto lock = LockGuard { &m_clMutex };
00039
00040
           pclListNode_->ClearNode();
           TypedDoubleLinkList<Timer>::Add(pclListNode_);
00041
00042
00043
           // Set the initial timer value
00044
          pclListNode_->m_u32TimeLeft = pclListNode_->m_u32Interval;
00045
           // Set the timer as active.
00046
           pclListNode_->m_u8Flags |= uTimerFlagActive;
00047
00048 }
00049
00050 //--
00051 void TimerList::Remove(Timer* pclLinkListNode_)
00052 {
00053
           KERNEL_ASSERT (nullptr != pclLinkListNode_);
          auto lock = LockGuard { &m_clMutex };
00054
00055
00056
          TypedDoubleLinkList<Timer>::Remove(pclLinkListNode_);
00057
          pclLinkListNode_->m_u8Flags &= ~uTimerFlagActive;
00058 }
00059
00060 //---
00061 void TimerList::Process(void)
00062 {
00063
           auto lock = LockGuard { &m_clMutex };
00064
00065
           auto* pclCurr = GetHead();
          // Subtract the elapsed time interval from each active timer.
while (nullptr != pclCurr) {
   auto* pclNext = pclCurr->GetNext();
00066
00067
00068
00069
00070
               // Active timers only...
00071
               if ((pclCurr->m_u8Flags & uTimerFlagActive) != 0) {
                   pclCurr->m_u32TimeLeft--;
if (0 == pclCurr->m_u32TimeLeft) {
00072
00073
00074
                        // Expired -- run the callback. these callbacks must be very fast...
00075
                        if (nullptr != pclCurr->m_pfCallback) {
00076
                            pclCurr->m_pfCallback(pclCurr->m_pclOwner, pclCurr->m_pvData);
00077
00078
                        if ((pclCurr->m_u8Flags & uTimerFlagOneShot) != 0) {
                            // If this was a one-shot timer, deactivate the timer + remove pclCurr->m_u8Flags |= uTimerFlagExpired;
00079
00080
                            pclCurr->m_u8Flags &= ~uTimerFlagActive;
00081
00082
                            Remove (pclCurr);
00083
                        } else {
00084
                            // Reset the interval timer.
                            pclCurr->m_u32TimeLeft = pclCurr->m_u32Interval;
00085
00086
00087
                   }
00088
00089
               pclCurr = pclNext;
00090
00091 }
00092
00093 } // namespace Mark3
```

## **Chapter 21**

# **Example Documentation**

## 21.1 lab10\_notifications/main.cpp

This examples demonstrates how to use notification objects as a thread synchronization mechanism.

```
--[Mark3 Realtime Platform]-
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
                #include "mark3.h"
Lab Example 10: Thread Notifications
Lessons covered in this example include:
- Create a notification object, and use it to synchronize execution of Threads.
- Notification objects are a lightweight mechanism to signal thread execution
  in situations where even a semaphore would be a heavier-weigth option.
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
Thread clapp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void ApplMain(void* unused_);
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void App2Main(void* unused_);
// idle thread -- do nothing
Thread clidleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
```

```
// Notification object used in the example.
Notify clNotify;
void ApplMain(void* unused_)
    while (1) {
        auto bNotified = false;
        // Block the thread until the notification object is signalled from
        // elsewhere.
        clNotify.Wait(&bNotified);
        Kernel::DebugPrint("T1: Notified\n");
void App2Main(void* unused_)
    while (1) {
        // Wait a while, then signal the notification object
        Kernel::DebugPrint("T2: Wait 1s\n");
        Thread::Sleep(1000);
        Kernel::DebugPrint("T2: Notify\n");
        clNotify.Signal();
} // anonymous namespace
using namespace Mark3;
int main (void)
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();
    // Initialize notifer and notify-ee threads
    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp1Thread.Start();
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);
    clApp2Thread.Start();
    // Initialize the Notify objects
    clNotify.Init();
    Kernel::Start();
```

### 21.2 lab11\_mailboxes/main.cpp

This examples shows how to use mailboxes to deliver data between threads in a synchronized way.

```
Lab Example 11: Mailboxes
Lessons covered in this example include:
- Initialize a mailbox for use as an IPC mechanism.
- Create and use mailboxes to pass data between threads.
- Mailboxes are a powerful IPC mechanism used to pass messages of a fixed-size
 between threads.
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
Thread clapp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void ApplMain(void* unused_);
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void App2Main(void* unused_);
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
Mailbox clMailbox;
uint8_t au8MBData[100];
typedef struct {
    uint8_t au8Buffer[10];
} MBType_t;
void ApplMain(void* unused_)
    while (1) {
        MBType_t stMsg;
        // Wait until there is an envelope available in the shared mailbox, and
        // then log a trace message.
        clMailbox.Receive(&stMsg);
        // KernelAware::Trace(0, __LINE__, stMsg.au8Buffer[0], stMsg.au8Buffer[9]);
void App2Main(void* unused_)
    while (1) {
        MBType_t stMsg;
        // Place a bunch of envelopes in the mailbox, and then wait for a
        // while. Note that this thread has a higher priority than the other // thread, so it will keep pushing envelopes to the other thread until
         // it gets to the sleep, at which point the other thread will be allowed
        // to execute.
        Kernel::DebugPrint("Messages Begin\n");
        for (uint8_t i = 0; i < 10; i++) {
    for (uint8_t j = 0; j < 10; j++) { stMsg.au8Buffer[j] = (i * 10) + j; }</pre>
             clMailbox.Send(&stMsg);
        Kernel::DebugPrint("Messages End\n");
        Thread::Sleep(2000);
} // anonymous namespace
using namespace Mark3;
```

```
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    // Initialize the threads used in this example
    clApplThread.Init(awApplStack, sizeof(awApplStack), 1, ApplMain, 0);
    clApplThread.Start();

    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 2, App2Main, 0);
    clApp2Thread.Start();

    // Initialize the mailbox used in this example
    clMailbox.Init(au8MBData, 100, sizeof(MBType_t));

    Kernel::Start();

    return 0;
}
```

### 21.3 lab1\_kernel\_setup/main.cpp

This example demonstrates basic kernel setup with two threads.

```
--[Mark3 Realtime Platform]-
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
#include "mark3.h"
Lab Example 1: Initializing the Mark3 RTOS kernel with two threads.
The following example code presents a working example of how to initialize
the Mark3 RTOS kernel, configure two application threads, and execute the
configured tasks. This example also uses the flAVR kernel-aware module to
print out messages when run through the flAVR AVR Simulator. This is a turnkey-ready example of how to use the Mark3 RTOS at its simplest level,
and should be well understood before moving on to other examples.
Lessons covered in this example include:
- usage of the Kernel class - configuring and starting the kernel - usage of the Thread class - initializing and starting static threads.
- Demonstrate the relationship between Thread objects, stacks, and entry
  functions.
 usage of Thread::Sleep() to block execution of a thread for a period of time
- When using an idle thread, the idle thread MUST not block.
Exercise:
- Add another application thread that prints a message, flashes an LED, etc.
 using the code below as an example.
Takeaway:
At the end of this example, the reader should be able to use the Mark3
Kernel and Thread APIs to initialize and start the kernel with any number
of static threads.
extern "C" {
void _
       _cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
```

```
using namespace Mark3;
// This block declares the thread data for the main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
Thread clAppThread;
K_WORD awAppStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void AppMain(void* unused_);
// This block declares the thread data for the idle thread. It defines a
// thread object, stack (in word-array form), and the entry-point function
// used by the idle thread.
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* unused);
void AppMain(void* unused_)
    // This function is run from within the application thread. Here, we
    // simply print a friendly greeting and allow the thread to sleep for a // while before repeating the message. Note that while the thread is
    // sleeping, CPU execution will transition to the Idle thread.
        Kernel::DebugPrint("Hello World!\n");
        Thread::Sleep(1000);
void IdleMain(void* unused_)
    while (1) {
        // Low priority task + power management routines go here.
         // The actions taken in this context must *not* cause the thread
         // to block, as the kernel requires that at least one thread is
         // schedulable at all times when not using an idle thread.
         // Note that if you have no special power-management code or idle
        // tasks, an empty while(1){} loop is sufficient to guarantee that
        // condition.
} // anonymous namespace
using namespace Mark3:
int main(void)
    // Before any Mark3 RTOS APIs can be called, the user must call Kernel::Init().
    // Note that if you have any hardware-specific init code, it can be called
    // before Kernel::Init, so long as it does not enable interrupts, or
    // rely on hardware peripherals (timer, software interrupt, etc.) used by the
    // kernel.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
    // Once the kernel initialization has been complete, the user can add their
    // application thread(s) and idle thread. Threads added before the kerel
    // is started are refered to as the "static threads" in the system, as they
    // are the default working-set of threads that make up the application on
    // kernel startup.
    \ensuremath{//} Initialize the application thread to use a specified word-array as its stack.
    // The thread will run at priority level "1", and start execution the // "AppMain" function when it's started.
    clAppThread.Init(awAppStack, sizeof(awAppStack), 1, AppMain, 0);
    \ensuremath{//} Initialize the idle thread to use a specific word-array as its stack.
    // The thread will run at priority level "0", which is reserved for the idle // priority thread. IdleMain will be run when the thread is started.
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    // Once the static threads have been added, the user must then ensure that the
    // threads are ready to execute. By default, creating a thread is created
    \ensuremath{//} in a STOPPED state. All threads must manually be started using the
    // Start() API before they will be scheduled by the system. Here, we are // starting the application and idle threads before starting the kernel - and
    // that's OK. When the kernel is started, it will choose which thread to run
    // first from the pool of ready threads.
    clAppThread.Start();
    clIdleThread.Start();
```

```
// All threads have been initialized and made ready. The kernel will now
// select the first thread to run, enable the hardware required to run the
// kernel (Timers, software interrupts, etc.), and then do whatever is
// necessary to maneuver control of thread execution to the kernel. At this
// point, execution will transition to the highest-priority ready thread.
// This function will not return.

Kernel::Start();

// As Kernel::Start() results in the operating system being executed, control
// will not be relinquished back to main(). The "return 0" is simply to
// avoid warnings.

return 0;
```

### 21.4 lab2\_coroutines/main.cpp

This example demonstrates usage of the coroutine scheduler run from the idle thread.

```
--[Mark3 Realtime Platform]---
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
#include "mark3.h"
/*----
The following example demonstrates how the Mark3 Coroutine Scheudler can be
used to implement cooperative scheduling in the system.
Cooperatively-scheduled tasks run to completion and are not pre-emptable,
unlike threads. They can, however be assigned a priority to ensure that
the highest-priority tasks are executed first. Cooperatively-scheduled
tasks should not block, but may use other Mark3 kernel APIs that cause other
threads to unblock (post semaphores, send messages, etc.). Note that the cooperatively-scheduled tasks are also intended to be run from a single
thread context - and so the notion of priority is relative only to the set
of coroutines in the system, and is unrelated to the priority of the threads
Lessons covered in this example include:
  Initialize the coroutine scheduler and run the scheduler loop from the
idle thread.
- Activate and run coroutines based on stimulus from outside the thread
running the coroutine scheduler.
- Add a coroutine that is activated by an interrupt
- Add a coroutine that is activated by a timer
Coroutines are an effective way of providing cooperative multitasking in a
system for tasks that do not require pre-emption, and can benefit from having
relative priorities.
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3:
// This block declares the thread data for the main application thread. It
```

```
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
Thread clappThread;
K_WORD awAppStack[PORT_KERNEL_DEFAULT_STACK_SIZE / sizeof(
     K_WORD) ];
      AppMain(void* unused_);
void
// This block declares the thread data for the idle thread. It defines a
// thread object, stack (in word-array form), and the entry-point function // used by the idle thread.
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE / sizeof(
     K_WORD)];
void IdleMain(void* unused_);
^{\prime\prime} // Declare 3 co-routines, each with a variable which will be passed into its
// handle function.
Coroutine clCoroutinel;
int iCounter1;
Coroutine clCoroutine2:
int iCounter2;
Coroutine clCoroutine3;
int iCounter3;
// Declare handler functions used by the three coroutines. In this example,
^{-} the coroutine tasks simply increments a counter and print their identity.
void CoroutineTask1(Coroutine* pclCaller_, void* pvArg_)
    Kernel::DebugPrint("Task1\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
}
void CoroutineTask2(Coroutine* pclCaller_, void* pvArg_)
    Kernel::DebugPrint("Task2\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
void CoroutineTask3(Coroutine* pclCaller_, void* pvArg_)
    Kernel::DebugPrint("Task3\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
void AppMain(void* unused_)
    // In this example, the application task is responsible for activating the
    // various coroutines in a round-robin fashion.
    while (1) {
        Thread::Sleep(100);
        clCoroutine1.Activate();
        Thread::Sleep(100);
        clCoroutine2.Activate();
        Thread::Sleep(100);
        clCoroutine3.Activate();
    }
}
void IdleMain(void* unused_)
    while (1) {
    // In this example, use the Idle context to run coroutines as they
        // are scheduled
        auto* pclCoRoutine = CoScheduler::Schedule();
        if (pclCoRoutine != nullptr) {
            pclCoRoutine->Run();
        1
} // anonymous namespace
using namespace Mark3;
int main (void)
```

```
// See Lab1 for detailed description of the kernel + initial thread
// bringup sequence.

Kernel::Init();
Kernel::SetDebugPrintFunction(DebugPrint);

CoScheduler::Init();
clCoroutinel.Init(1, CoroutineTask1, &iCounter1);
clCoroutine2.Init(2, CoroutineTask2, &iCounter2);
clCoroutine3.Init(3, CoroutineTask3, &iCounter3);

clAppThread.Init(awAppStack, sizeof(awAppStack), 1, AppMain, 0);
clAppThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
clAppThread.Start();
clIdleThread.Start();
Kernel::Start();
```

### 21.5 lab3\_round\_robin/main.cpp

This example demonstrates how to use round-robin thread scheduling with multiple threads of the same priority.

```
--[Mark3 Realtime Platform]-
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
#include "mark3.h"
Lab Example 3: using round-robin scheduling to time-slice the CPU.
Lessons covered in this example include:
 Threads at the same priority get timesliced automatically The Thread::SetQuantum() API can be used to set the maximum amount of CPU
  time a thread can take before being swapped for another task at that
 priority level.
Takeawav:
- CPU Scheduling can be achieved using not just strict Thread priority, but
 also with round-robin time-slicing between threads at the same priority.
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
Thread clApp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
      ApplMain(void* unused_);
void
// This block declares the thread data for one main application thread. It // defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
```

```
void App2Main(void* unused_);
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
void ApplMain(void* unused )
     // Simple loop that increments a volatile counter to 1000000 then resets
     // it while printing a message.
     volatile uint32_t u32Counter = 0;
     while (1) {
         u32Counter++;
          if (u32Counter == 1000000) {
              u32Counter = 0;
              Kernel::DebugPrint("Thread 1 - Did some work\n");
         }
    }
}
void App2Main(void* unused_)
    // Same as ApplMain. However, as this thread gets twice as much CPU time // as Thread 1, you should see its message printed twice as often as the
     // above function.
     volatile uint32_t u32Counter = 0;
     while (1) {
         u32Counter++;
         if (u32Counter == 1000000) {
               u32Counter = 0;
              Kernel::DebugPrint("Thread 2 - Did some work\n");
     }
} // anonymous namespace
using namespace Mark3;
int main(void)
     // See the annotations in lab1.
     Kernel::Init();
     Kernel::SetDebugPrintFunction(DebugPrint);
     clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
     clIdleThread.Start();
     // In this exercise, we create two threads at the same priority level. // As a result, the CPU will automatically swap between these threads // at runtime to ensure that each get a chance to execute.
     clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);
     // Set the threads up so that Thread 1 can get 4ms of CPU time uninterrupted, // but Thread 2 can get 8ms of CPU time uninterrupted. This means that
     // in an ideal situation, Thread 2 will get to do twice as much work as
     // Thread 1 - even though they share the same scheduling priority.
     // Note that if SetQuantum() isn't called on a thread, a default value
     // is set such that each thread gets equal timeslicing in the same // priority group by default. You can play around with these values and
     // observe how it affects the execution of both threads.
     clApp1Thread.SetQuantum(4);
     clApp2Thread.SetQuantum(8);
     clApplThread.Start();
     clApp2Thread.Start();
     Kernel::Start();
     return 0:
```

## 21.6 lab4\_semaphores/main.cpp

This example demonstrates how to use semaphores for Thread synchronization.

```
-- [Mark3 Realtime Platform]
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
#include "mark3.h"
Lab Example 4: using binary semaphores
In this example, we implement two threads, synchronized using a semaphore to
model the classic producer-consumer pattern. One thread does work, and then posts the semaphore indicating that the other thread can consume that work.
The blocking thread just waits idly until there is data for it to consume.
Lessons covered in this example include: 
 -Use of a binary semaphore to implement the producer-consumer pattern \,
-Synchronization of threads (within a single priority, or otherwise)
 using a semaphore
Semaphores can be used to control which threads execute at which time. This
allows threads to work cooperatively to achieve a goal in the system.
extern "C" {
       _cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApplThread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
void ApplMain(void* unused_);
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void App2Main(void* unused_);
// idle thread -- do nothing
Thread clidleThread:
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
     while (1) {}
// This is the semaphore that we'll use to synchronize two threads in this
// demo application
Semaphore clMySem;
void ApplMain(void* unused)
     while (1) {
```

```
// Wait until the semaphore is posted from the other thread
        Kernel::DebugPrint("Wait\n");
        clMySem.Pend();
        // Producer thread has finished doing its work -- do something to
        // consume its output. Once again - a contrived example, but we // can imagine that printing out the message is "consuming" the output
         // from the other thread.
        Kernel::DebugPrint("Triggered!\n");
void App2Main(void* unused_)
    volatile uint32_t u32Counter = 0;
    while (1) {
        // Do some work. Once the work is complete, post the semaphore. This
        // will cause the other thread to wake up and then take some action.
        // It's a bit contrived, but imagine that the results of this process
        // are necessary to drive the work done by that other thread.
        u32Counter++;
        if (u32Counter == 1000000) {
            u32Counter = 0;
            Kernel::DebugPrint("Posted\n");
            clMySem.Post();
} // anonymous namespace
using namespace Mark3;
int main(void)
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();
    // In this example we create two threads to illustrate the use of a
    \ensuremath{//} binary semaphore as a synchronization method between two threads.
    // Thread 1 is a "consumer" thread -- It waits, blocked on the semaphore
    \ensuremath{//} until thread 2 is done doing some work. Once the semaphore is posted,
    // the thread is unblocked, and does some work.
    // Thread 2 is thus the "producer" thread -- It does work, and once that
    // work is done, the semaphore is posted to indicate that the other thread
    // can use the producer's work product.
    clApp1Thread.Init(awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);
    clApp1Thread.Start();
    clApp2Thread.Start();
    // Initialize a binary semaphore (maximum value of one, initial value of
    // zero).
    clMySem.Init(0, 1);
    Kernel::Start();
    return 0;
```

### 21.7 lab5\_mutexes/main.cpp

This example demonstrates how to use mutexes to protect against concurrent access to resources.

```
--[Mark3 Realtime Platform]-----
Copyright (c) 2012 - 2019 mOslevin, all rights reserved. See license.txt for more information % \left( 1\right) =\left( 1\right) +\left( 1
 /*----
Lab Example 5: using Mutexes.
Lessons covered in this example include:
 -You can use mutexes to lock accesses to a shared resource
 extern "C" {
 void __cxa_pure_virtual(void) {}
 void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
 // This block declares the thread data for one main application thread. It
 // defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
 Thread clapp1Thread;
 K_WORD awApp1Stack[APP1_STACK_SIZE];
 void ApplMain(void* unused_);
 // This block declares the thread data for one main application thread. It
 // defines a thread object, stack (in word-array form), and the entry-point
 // function used by the application thread.
#define APP2_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
 Thread clApp2Thread;
 K_WORD awApp2Stack[APP2_STACK_SIZE];
 void App2Main(void* unused_);
 // idle thread -- do nothing
 Thread clIdleThread;
 K_WORD awidleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
 void IdleMain(void* /*unused_*/)
            while (1) {}
 // This is the mutex that we'll use to synchronize two threads in this
  // demo application.
 Mutex clMyMutex;
 \ensuremath{//} This counter variable is the "shared resource" in the example, protected
 // by the mutex. Only one thread should be given access to the counter at
 // anv time.
 volatile uint32 t u32Counter = 0;
 void ApplMain(void* unused_)
             while (1) {
                        // Claim the mutex. This will prevent any other thread from claiming
                         // this lock simulatenously. As a result, the other thread has to
                         // wait until we're done before it can do its work. You will notice
                        // that the Start/Done prints for the thread will come as a pair (i.e. // you won't see "Thread2: Start" then "Thread1: Start").
                        clMvMutex.Claim();
                          // Start our work (incrementing a counter). Notice that the Start and
                         // Done prints wind up as a pair when simuated with flAVR.
                         Kernel::DebugPrint("Thread1: Start\n");
                        u32Counter++:
                        while (u32Counter <= 1000000) { u32Counter++; }</pre>
                         u32Counter = 0;
                         Kernel::DebugPrint("Thread1: Done\n");
                         // Release the lock, allowing the other thread to do its thing.
                        clMyMutex.Release();
             }
```

```
}
void App2Main(void* unused_)
    while (1) {
         // Claim the mutex. This will prevent any other thread from claiming
         // this lock simulatenously. As a result, the other thread has to
         // wait until we're done before it can do its work. You will notice
         // that the Start/Done prints for the thread will come as a pair (i.e.
// you won't see "Thread2: Start" then "Thread1: Start").
         clMyMutex.Claim();
         \ensuremath{//} Start our work (incrementing a counter). Notice that the Start and
         // Done prints wind up as a pair when simuated with flAVR.
         Kernel::DebugPrint("Thread2: Start\n");
         u32Counter++;
         while (u32Counter <= 1000000) { u32Counter++; }</pre>
         u32Counter = 0;
         Kernel::DebugPrint("Thread2: Done\n");
         \ensuremath{//} Release the lock, allowing the other thread to do its thing.
         clMyMutex.Release();
} // anonymous namespace
using namespace Mark3;
int main(void)
     // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();
    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);
    clApp1Thread.Start();
    clApp2Thread.Start();
    // Initialize the mutex used in this example.
    clMyMutex.Init();
    Kernel::Start();
    return 0;
```

## 21.8 lab6\_timers/main.cpp

This example demonstrates how to create and use software timers.

```
Demonstration of the periodic and one-shot timer APIs provided by Mark3
Takeaway:
{\tt Mark3} can be used to provide flexible one-shot and periodic timers.
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point // function used by the application thread.
#define APP1_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApplThread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
void ApplMain(void* unused_);
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
}
void PeriodicCallback(Thread* owner, void* pvData_)
     // Timer callback function used to post a semaphore. Posting the semaphore
    // will wake up a thread that's pending on that semaphore.
    auto* pclSem = static_cast<Semaphore*>(pvData_);
    pclSem->Post();
void OneShotCallback(Thread* owner, void* pvData_)
    Kernel::DebugPrint("One-shot timer expired.\n");
void App1Main(void* unused_)
    Timer clMyTimer; // Periodic timer object Timer clOneShot; // One-shot timer object
    Semaphore clMySem; // Semaphore used to wake this thread
    // Initialize a binary semaphore (maximum value of one, initial value of
    clMySem.Init(0, 1);
    // Start a timer that triggers every 500\,\mathrm{ms} that will call PeriodicCallback. // This timer simulates an external stimulus or event that would require
     // an action to be taken by this thread, but would be serviced by an
     // interrupt or other high-priority context.
     // PeriodicCallback will post the semaphore which wakes the thread
    \ensuremath{//} up to perform an action. Here that action consists of a trivial message
    // print.
    clMyTimer.Start(true, 500, PeriodicCallback, (void*)&clMySem);
     \ensuremath{//} Set up a one-shot timer to print a message after 2.5 seconds, asynchronously
    \ensuremath{//} from the execution of this thread.
    clOneShot.Start(false, 2500, OneShotCallback, 0);
    while (1) {
         // Wait until the semaphore is posted from the timer expiry
         clMySem.Pend();
         \ensuremath{//} Take some action after the timer posts the semaphore to wake this
         // thread.
         Kernel::DebugPrint("Thread Triggered.\n");
} // anonymous namespace
using namespace Mark3;
```

```
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApplThread.Init(awApplStack, sizeof(awApplStack), 1, ApplMain, 0);
    clApplThread.Start();

    Kernel::Start();

    return 0;
}
```

## 21.9 lab7\_events/main.cpp

This example demonstrates how to create and use event groups

```
--[Mark3 Realtime Platform]---
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
Lab Example 7: using Event Flags
Lessons covered in this example include:
-Using the EventFlag Class to synchronize thread execution
-Explore the behavior of the EventFlagOperation::Any_Set and EventFlagOperation::All_Set, and the
 event-mask bitfield.
Takeaway:
Like Semaphores and Mutexes, EventFlag objects can be used to synchronize
the execution of threads in a system. The EventFlag class allows for many
threads to share the same object, blocking on different event combinations. This provides an efficient, robust way for threads to process asynchronous
system events that occur with a unified interface.
extern "C" {
void cxa pure virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
void ApplMain(void* unused_);
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point // function used by the application thread.
#define APP2_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
```

```
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
       App2Main(void* unused_);
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
}
EventFlag clFlags;
void ApplMain(void* unused_)
    while (1) {
          // Block this thread until any of the event flags have been set by
         ^{\prime\prime} some outside force (here, we use Thread 2). As an exercise to the ^{\prime\prime} user, try playing around with the event mask to see the effect it
         // has on which events get processed. Different threads can block on // different bitmasks - this allows events with different real-time
         // priorities to be handled in different threads, while still using
         // the same event-flag object.
         // Also note that EventFlagOperation::Any_Set indicates that the thread will be
         // unblocked whenever any of the flags in the mask are selected. If // you wanted to trigger an action that only takes place once multiple
         // you wanted to trigger an action that only takes place once much // bits are set, you could block the thread waiting for a specific // event bitmask with EventFlagOperation::All_Set specified.
         auto u16Flags = c1Flags.Wait(0xFFFF, EventFlagOperation::Any_Set);
         // Print a message indicaating which bit was set this time.
         switch (u16Flags) {
              case 0x0001: Kernel::DebugPrint("Event1\n"); break;
              case 0x0002: Kernel::DebugPrint("Event2\n"); break;
              case 0x0004: Kernel::DebugPrint("Event3\n"); break;
              case 0x0008: Kernel::DebugPrint("Event4\n"); break;
case 0x0010: Kernel::DebugPrint("Event5\n"); break;
              case 0x0020: Kernel::DebugPrint("Event6\n"); break;
              case 0x0040: Kernel::DebugPrint("Event7\n"); break;
              case 0x0080: Kernel::DebugPrint("Event8\n"); break;
              case 0x0100: Kernel::DebugPrint("Event9\n"); break;
              case 0x0200: Kernel::DebugPrint("Event10\n"); break;
              case 0x0400: Kernel::DebugPrint("Event11\n"); break;
case 0x0800: Kernel::DebugPrint("Event12\n"); break;
              case 0x1000: Kernel::DebugPrint("Event13\n"); break;
              case 0x2000: Kernel::DebugPrint("Event14\n"); break;
              case 0x4000: Kernel::DebugPrint("Event15\n"); break;
              case 0x8000: Kernel::DebugPrint("Event16\n"); break;
              default: break;
         // Clear the event-flag that we just printed a message about. This
          // will allow u16 to acknowledge further events in that bit in the future.
         clFlags.Clear(u16Flags);
}
void App2Main(void* unused_)
    uint16_t u16Flag = 1;
    while (1) {
         Thread::Sleep(100);
         // Event flags essentially map events to bits in a bitmap.
         // set one bit each 100ms. In this loop, we cycle through bits 0-15
         // repeatedly. Note that this will wake the other thread, which is
          // blocked, waiting for *any* of the flags in the bitmap to be set.
         clFlags.Set (u16Flag);
          // Bitshift the flag value to the left. This will be the flag we set
         // the next time this thread runs through its loop.
         if (u16Flag != 0x8000) {
              u16Flag <<= 1;
         } else {
              u16Flag = 1;
} // anonymous namespace
using namespace Mark3:
```

```
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApplThread.Init(awApplStack, sizeof(awApplStack), 1, ApplMain, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();
    clFlags.Init();

    Kernel::Start();

    return 0;
}
```

### 21.10 lab8\_messages/main.cpp

This example demonstrates how to pass data between threads using message passing.

```
--[Mark3 Realtime Platform]
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information
#include "mark3.h"
Lab Example 8: using messages for IPC.
In this example, we present a typical asynchronous producer/consumer pattern
using Mark3's message-driven IPC.
Lessons covered in this example include:
- use of Message and MessageQueue objects to send data between threads
- use of GlobalMessagePool to allocate and free message objects
Unlike cases presented in previous examples that relied on semaphores or
event flags, messages carry substantial context, specified in its "code" and "data" members. This mechanism can be used to pass data between threads extremely efficiently, with a simple and flexible API. Any number of threads
can write to/block on a single message queue, which give this method of
IPC even more flexibility.
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
// This block declares the thread data for one main application thread. It
^{\prime\prime} defines a thread object, stack (in word-array form), and the entry-point ^{\prime\prime} function used by the application thread.
#define APP1_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApplThread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
       ApplMain(void* unused_);
```

```
// This block declares the thread data for one main application thread. It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void App2Main(void* unused_);
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
    while (1) {}
}
MessageQueue clMsgQ;
#define MESSAGE POOL SIZE (3)
MessagePool s_clMessagePool;
           s_clMessages[MESSAGE_POOL_SIZE];
Message
void ApplMain(void* unused_)
    auto u16Data = 0;
    while (1) {
        // This thread grabs a message from the global message pool, sets a
        // code-value and the message data pointer, then sends the message to
        // a message queue object. Another thread (Thread2) is blocked, waiting
        // for a message to arrive in the queue.
        // Get the message object
        auto* pclMsg = s_clMessagePool.Pop();
        // Set the message object's data (contrived in this example)
        pclMsg->SetCode(0x1337);
        u16Data++:
        pclMsg->SetData(&u16Data):
         // Send the message to the shared message queue
        clMsgQ.Send(pclMsg);
        // Wait before sending another message.
        Thread::Sleep(200);
}
void App2Main(void* unused_)
    while (1) {
        // This thread waits until it receives a message on the shared global
        // message queue. When it gets the message, it prints out information
         // about the message's code and data, before returning the messaage object
        /\!/ back to the global message pool. In a more practical application, /\!/ the user would typically use the code to tell the receiving thread
        // what kind of message was sent, and what type of data to expect in the
        // data field.
        \ensuremath{//} Wait for a message to arrive on the specified queue. Note that once
         // this thread receives the message, it is "owned" by the thread, and
        \ensuremath{//} must be returned back to its source message pool when it is no longer
        // needed.
        auto* pclMsq = clMsqQ.Receive();
         // We received a message, now print out its information
        Kernel::DebugPrint("Received Message\n");
        // KernelAware::Trace(0, __LINE__, pclMsg->GetCode(), *((uint16_t*)pclMsg->GetData()));
        // Done with the message, return it back to the global message queue.
        s_clMessagePool.Push(pclMsg);
} // anonymous namespace
using namespace Mark3;
int main (void)
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
```

```
clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
clIdleThread.Start();

clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

clApp1Thread.Start();
clApp2Thread.Start();
clMsgQ.Init();

s_clMessagePool.Init();
for (int i = 0; i < MESSAGE_POOL_SIZE; i++) {
    s_clMessageS[i].Init();
    s_clMessagePool.Push(&s_clMessageS[i]);
}

Kernel::Start();
return 0;</pre>
```

### 21.11 lab9\_dynamic\_threads/main.cpp

This example demonstrates how to create and destroy threads dynamically at runtime.

```
--[Mark3 Realtime Platform]-
Copyright (c) 2012 - 2019 mOslevin, all rights reserved.
See license.txt for more information % \left( 1\right) =\left( 1\right) \left( 1\right) \left(
 #include "mark3.h"
 #include "memutil.h"
Lab Example 9: Dynamic Threading
Lessons covered in this example include:
   - Creating, pausing, and destorying dynamically-created threads at runtime
In addition to being able to specify a static set of threads during system initialization, Mark3 gives the user the ability to create and manipu32ate threads at runtime. These threads can act as "temporary workers" that can
be activated when needed, without impacting the responsiveness of the rest
of the application.
 extern "C" {
 void __cxa_pure_virtual(void) {}
 void DebugPrint(const char* szString_);
namespace
using namespace Mark3;
 // This block declares the thread data for one main application thread. It
  // defines a thread object, stack (in word-array form), and the entry-poi nt
  // function used by the application thread.
 Thread clapplThread;
  K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
 void ApplMain(void* unused_);
  // This block declares the thread stack data for a thread that we'll create
  // dynamically.
```

```
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void IdleMain(void* /*unused_*/)
     while (1) {}
#define MAX_THREADS (10)
Thread* apclActiveThreads[10];
uint32_t au32ActiveTime[10];
void PrintThreadSlack(void)
     Kernel::DebugPrint("Stack Slack");
     for (uint8_t i = 0; i < MAX_THREADS; i++) {</pre>
         if (apclActiveThreads[i] != 0) {
              char szStr[10];
              auto u16Slack = apclActiveThreads[i]->GetStackSlack();
              MemUtil::DecimalToHex((K_ADDR)apclActiveThreads[i], szStr);
              Kernel::DebugPrint(szStr);
              Kernel::DebugPrint(" ");
              MemUtil::DecimalToString(u16Slack, szStr);
              Kernel::DebugPrint(szStr);
              Kernel::DebugPrint("\n");
         }
    }
}
void PrintCPUUsage(void)
     Kernel::DebugPrint("Cpu usage\n");
for (int i = 0; i < MAX_THREADS; i++) {</pre>
         if (apclActiveThreads[i] != 0) {
              // KernelAware::Trace(0, __LINE__, (K_ADDR)apclActiveThreads[i], au16ActiveTime[i]);
}
void ThreadCreate(Thread* pclThread_)
     Kernel::DebugPrint("TC\n");
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
   if (apclActiveThreads[i] == 0) {
      apclActiveThreads[i] = pclThread_;
   }
}</pre>
              break;
     CriticalSection::Exit();
     PrintThreadSlack();
     PrintCPUUsage();
void ThreadExit(Thread* pclThread_)
     Kernel::DebugPrint("TX\n");
     CriticalSection::Enter();
     for (uint8_t i = 0; i < MAX_THREADS; i++) {</pre>
         if (apclActiveThreads[i] == pclThread_) {
              apclActiveThreads[i] = 0;
              au32ActiveTime[i]
                                      = 0:
              break:
     CriticalSection::Exit();
     PrintThreadSlack();
     PrintCPUUsage();
void ThreadContextSwitch(Thread* pclThread_)
     Kernel::DebugPrint("CS\n");
     static uint32_t u32LastTicks = 0;
                                     = Kernel::GetTicks();
                      u32Ticks
     auto
     CriticalSection::Enter();
     for (uint8_t i = 0; i < MAX_THREADS; i++) {</pre>
         if (apclActiveThreads[i] == pclThread_) {
    au32ActiveTime[i] += u32Ticks - u32LastTicks;
              break:
```

```
}
    CriticalSection::Exit();
    u32LastTicks = u32Ticks;
}
void WorkerMain1(void* arg_)
           pclSem = static_cast<Semaphore*>(arg_);
    auto*
    uint32_t u32Count = 0;
    // Do some work. Post a semaphore to notify the other thread that the
    \ensuremath{//} work has been completed.
    while (u32Count < 1000000) { u32Count++; }</pre>
    Kernel::DebugPrint("Worker1 -- Done Work\n");
    pclSem->Post();
    // Work is completed, just spin now. Let another thread destory u16.
    while (1) {}
void WorkerMain2(void* arg_)
    uint32_t u32Count = 0;
    while (u32Count < 1000000) { u32Count++; }</pre>
    Kernel::DebugPrint("Worker2 -- Done Work\n");
    // A dynamic thread can self-terminate as well:
    Scheduler::GetCurrentThread() ->Exit();
void ApplMain(void* unused_)
    Thread clMyThread;
    Semaphore clMySem;
    clMySem.Init(0, 1);
    while (1) {
        // Example 1 - create a worker thread at our current priority in order to
        // parallelize some work.
        clMyThread.Init(awApp2Stack, sizeof(awApp2Stack), 1, WorkerMain1, (void*)&clMySem);
        clMyThread.Start();
        // Do some work of our own in parallel, while the other thread works on its project.
        uint32_t u32Count = 0;
        while (u32Count < 100000) { u32Count++; }</pre>
        Kernel::DebugPrint("Thread -- Done Work\n");
        PrintThreadSlack():
        // Wait for the other thread to finish its job.
        clMySem.Pend();
        // Once the thread has signalled ul6, we can safely call "Exit" on the thread to
        \ensuremath{//} remove it from scheduling and recycle it later.
        clMyThread.Exit();
        // Spin the thread up again to do something else in parallel. This time, the thread
        // will run completely asynchronously to this thread.
        clMyThread.Init(awApp2Stack, sizeof(awApp2Stack), 1, WorkerMain2, 0);
        clMyThread.Start();
        u32Count = 0;
        while (u32Count < 1000000) { u32Count++; }</pre>
        Kernel::DebugPrint("Thread -- Done Work\n");
        // Check that we're sure the worker thread has terminated before we try running the
        // test loop again.
        while (clMyThread.GetState() != ThreadState::Exit) {}
        Kernel::DebugPrint(" Test Done\n");
        Thread::Sleep(1000);
        PrintThreadSlack();
} // anonymous namespace
using namespace Mark3;
int main (void)
```

```
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    Kernel::SetThreadCreateCallout(ThreadCreate);
    Kernel::SetThreadExitCallout(ThreadExit);
    Kernel::SetThreadContextSwitchCallout(ThreadContextSwitch);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApplThread.Init(awApplStack, sizeof(awApplStack), 1, ApplMain, 0);
    clApplThread.Start();
    Kernel::Start();
    return 0;
}
```

## Index

```
/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c. ← cpp, 475, 476
          cpp, 295, 346
                                                         /home/moslevin/projects/m3-repo/kernel/src/message. ←
/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/fake ← cpp, 478, 479
          _types.h, 358
                                                         /home/moslevin/projects/m3-repo/kernel/src/mutex.cpp,
/home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/mark3d.80, 481
         h, 360, 419
                                                         /home/moslevin/projects/m3-repo/kernel/src/notify.cpp,
/home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil. <
                                                                   483, 484
          cpp, 424, 425
                                                         /home/moslevin/projects/m3-repo/kernel/src/profile.cpp,
/home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil. 486
          h, 431, 432
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/atomic. ←
/home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamerh, 487, 488
         h, 433, 434
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/autoalloc. ←
/home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer. ←
                                                                   h, 489, 490
          cpp, 434, 435
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/blocking. ←
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/Rernetswi.
          cpp, 438
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/colist. ←
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/ktth neltimer. ←
          cpp. 439, 441
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/condvar. ←
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gtc/A04blit95ortcfg. ←
          h, 442, 447
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/coroutine. ←
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/dccAssiblits/threadport. ←
         h, 448, 450
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/cosched. ←
/home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gtcAtbreadport. ←
          cpp, 452, 453
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/criticalguard. ←
/home/moslevin/projects/m3-repo/kernel/src/atomic. ←
          cpp, 455
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/criticalsection. ←
/home/moslevin/projects/m3-repo/kernel/src/autoalloc. ←
          cpp, 456
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/eventflag. ←
/home/moslevin/projects/m3-repo/kernel/src/blocking. ←
                                                                   h. 500
          cpp. 457, 458
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/ithreadport. ←
/home/moslevin/projects/m3-repo/kernel/src/colist.cpp,
                                                                   h, 501, 502
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/kernel. ←
/home/moslevin/projects/m3-repo/kernel/src/condvar. ←
                                                                   h, 502, 503
          cpp, 460
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/kerneldebug. ←
/home/moslevin/projects/m3-repo/kernel/src/coroutine. ←
                                                                   h, 504, 505
          cpp, 461, 462
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.
/home/moslevin/projects/m3-repo/kernel/src/cosched. ←
                                                                   h. 506
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer. ←
/home/moslevin/projects/m3-repo/kernel/src/eventflag. ←
                                                                   h. 507
          cpp, 464
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/kerneltypes. ←
/home/moslevin/projects/m3-repo/kernel/src/kernel.cpp,
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore. ←
/home/moslevin/projects/m3-repo/kernel/src/ksemaphore. ←
                                                                   h, 509, 510
          cpp, 469, 470
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/II.h.
/home/moslevin/projects/m3-repo/kernel/src/ll.cpp, 472,
                                                                   510 511
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/lockguard. ←
/home/moslevin/projects/m3-repo/kernel/src/lockguard. ←
                                                                   h, 513, 514
          cpp, 474, 475
                                                         /home/moslevin/projects/m3-repo/kernel/src/public/mailbox.
/home/moslevin/projects/m3-repo/kernel/src/mailbox. ←
                                                                   h, 514, 515
```

```
/home/moslevin/projects/m3-repo/kernel/src/public/manual/home/moslevin/projects/m3-repo/kernel/src/timer.cpp,
                   h, 516, 517
                                                                                                                                   565, 566
/home/moslevin/projects/m3-repo/kernel/src/public/mark3. ₄home/moslevin/projects/m3-repo/kernel/src/timerlist. ←
                                                                                                                                   cpp, 567
                   h, 517, 518
/home/moslevin/projects/m3-repo/kernel/src/public/mark3cfgBlockingObject
                   h, 518, 521
                                                                                                                          Mark3::BlockingObject, 111
/home/moslevin/projects/m3-repo/kernel/src/public/message:Coroutine
                                                                                                                          Mark3::Coroutine, 125
                   h. 522, 523
/home/moslevin/projects/m3-repo/kernel/src/public/mutex. CriticalGuard
                                                                                                                          Mark3::CriticalGuard, 131
                   h, 524, 525
/home/moslevin/projects/m3-repo/kernel/src/public/notify. ~ EventFlag
                                                                                                                          Mark3::EventFlag, 137
                   h, 526, 527
/home/moslevin/projects/m3-repo/kernel/src/public/paniccodesckGuard
                                                                                                                          Mark3::LockGuard, 159
                   h, 527, 530
/home/moslevin/projects/m3-repo/kernel/src/public/priomap::Mailbox
                                                                                                                          Mark3::Mailbox, 163
                   h, 530, 531
/home/moslevin/projects/m3-repo/kernel/src/public/priomap {\tt MessagePool} \\
                                                                                                                          Mark3::MessagePool, 186
                   h, 531, 532
/home/moslevin/projects/m3-repo/kernel/src/public/priomapiz \underline{\text{MessageQueue}}
                                                                                                                          Mark3::MessageQueue, 190
                   h, 533, 534
/home/moslevin/projects/m3-repo/kernel/src/public/profile.\sim Mutex
                                                                                                                          Mark3::Mutex, 194
                   h, 535, 536
/home/moslevin/projects/m3-repo/kernel/src/public/profiling {\overset{Notify}{\sim}} 1.00 \times 10^{-10} \times 10^{
                                                                                                                          Mark3::Notify, 200
                    _results.h, 536
/home/moslevin/projects/m3-repo/kernel/src/public/quantu\widetilde{m}. \\ \underline{SchedulerGuard}
                                                                                                                          Mark3::SchedulerGuard, 228
                   h. 537
/home/moslevin/projects/m3-repo/kernel/src/public/readerwirter. \\ \\ \underline{\hspace{0.5cm}} Semaphore
                                                                                                                          Mark3::Semaphore, 230
                   h. 538, 539
/home/moslevin/projects/m3-repo/kernel/src/public/schedul\widetilde{e}r\overset{T}{\longleftarrow} r^{thread}
                                                                                                                          Mark3::Thread, 245
                   h, 539, 541
/home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard
                                                                                                                          Mark3::Timer, 275
                   h, 541, 542
/home/moslevin/projects/m3-repo/kernel/src/public/sizeprofiles/m/
                   h, 542
                                                                                                                         threadport.h, 449
/home/moslevin/projects/m3-repo/kernel/src/public/thread. AVR
                   h, 543, 544
                                                                                                                         portcfg.h, 443
/home/moslevin/projects/m3-repo/kernel/src/public/threadliatcquireReader
                   h, 546, 547
                                                                                                                          Mark3::ReaderWriterLock, 218
/home/moslevin/projects/m3-repo/kernel/src/public/threadliatigtuireReader i
                   h, 547, 548
                                                                                                                          Mark3::ReaderWriterLock, 218
/home/moslevin/projects/m3-repo/kernel/src/public/timer. ← Acquire Writer
                   h, 548, 549
                                                                                                                          Mark3::ReaderWriterLock, 219
/home/moslevin/projects/m3-repo/kernel/src/public/timerlistAcquireWriter i
                   h, 550, 551
                                                                                                                          Mark3::ReaderWriterLock, 219
/home/moslevin/projects/m3-repo/kernel/src/public/timerschectlylage-
                   h, 551, 552
                                                                                                                          Mark3::Coroutine, 125
/home/moslevin/projects/m3-repo/kernel/src/quantum. ← Add
                   cpp, 552, 553
                                                                                                                          Mark3::Atomic, 103
/home/moslevin/projects/m3-repo/kernel/src/readerwriter. ←
                                                                                                                         Mark3::CircularLinkList, 116
                   cpp, 554
                                                                                                                         Mark3::CoList, 118
/home/moslevin/projects/m3-repo/kernel/src/scheduler. ←
                                                                                                                         Mark3::DoubleLinkList, 134
                   cpp, 555, 556
                                                                                                                         Mark3::Scheduler, 223
/home/moslevin/projects/m3-repo/kernel/src/thread.cpp,
                                                                                                                         Mark3::ThreadList, 265
                   557, 558
                                                                                                                         Mark3::ThreadListList, 270
/home/moslevin/projects/m3-repo/kernel/src/threadlist. ←
                                                                                                                         Mark3::TimerList, 281
                   cpp, 563
                                                                                                                         Mark3::TimerScheduler, 284
/home/moslevin/projects/m3-repo/kernel/src/threadlistlist. ←
                                                                                                                         Mark3::TypedCircularLinkList, 287
                   cpp, 565
                                                                                                                         Mark3::TypedDoubleLinkList, 291
```

AddPriority	Atomic_Sub16
Mark3::ThreadList, 267	mark3c.cpp, 306
Alloc_ConditionVariable	mark3c.h, 381
mark3c.cpp, 300	Atomic_Sub32
Alloc_EventFlag	mark3c.cpp, 307
mark3c.cpp, 300	mark3c.h, 382
Alloc_Mailbox	Atomic_Sub8
mark3c.cpp, 300	mark3c.cpp, 307
mark3c.h, 375	mark3c.h, 382
Alloc_Memory	Atomic_TestAndSet
mark3c.cpp, 300	mark3c.cpp, 308
mark3c.h, 375	mark3c.h, 383
Alloc_Message	AutoAllocAllocator_t
mark3c.cpp, 300	Mark3, 95
mark3c.h, 376	AutoAllocFree_t
Alloc_MessagePool	Mark3, 95
mark3c.cpp, 301	AutoAllocType
mark3c.h, 376	Mark3, 97
Alloc_MessageQueue	Block
mark3c.cpp, 301	Mark3::BlockingObject, 111
mark3c.h, 376	BlockPriority
Alloc_Mutex	Mark3::BlockingObject, 112
mark3c.cpp, 301	Blocked
mark3c.h, 376	Mark3, 98
Alloc_Notify	BlockingObject
mark3c.cpp, 302	Mark3::BlockingObject, 111
mark3c.h, 377	Broadcast
Alloc_ReaderWriterLock	Mark3::ConditionVariable, 121
mark3c.cpp, 302	
Alloc_Semaphore	CONDITIONVARIABLE_SIZE
mark3c.cpp, 302	mark3c.h, 366
mark3c.h, 377	CanRead
Alloc_Thread	Mark3::Streamer, 236
mark3c.cpp, 303	CanWrite
mark3c.h, 377	Mark3::Streamer, 236
Alloc_Timer	Cancel
mark3c.cpp, 303	Mark3::Quantum, 214
mark3c.h, 378 Allocate	Checksum16
Mark3::AutoAlloc, 106	Mark3::MemUtil, 174
Atomic Add16	Checksum8
mark3c.cpp, 303	Mark3::MemUtil, 174
mark3c.h, 378	CircularLinkList
Atomic Add32	Mark3::CircularLinkList, 115
mark3c.cpp, 304	Mark3::LinkListNode, 157
mark3c.h, 379	Claim
Atomic Add8	Mark3::Mutex, 194, 195
mark3c.cpp, 304	Mark3::Streamer, 236
mark3c.h, 379	Claim_i
Atomic Set16	Mark3::Mutex, 195 Clear
mark3c.cpp, 305	Mark3::EventFlag, 137
mark3c.h, 380	Mark3::PriorityMapL1, 203
Atomic_Set32	Mark3::PriorityMapL2, 207
mark3c.cpp, 305	ClearInTimer
mark3c.h, 380	Mark3::Quantum, 214
Atomic Set8	ClearNode
mark3c.cpp, 306	Mark3::LinkListNode, 156
mark3c.h, 381	CoPrioMap
,	1

Mark3, 95	mark3c.h, 368
CompareMemory	DECLARE_THREAD
Mark3::MemUtil, 175	mark3c.h, 369
CompareStrings	DECLARE_TIMER
Mark3::MemUtil, 175, 176	mark3c.h, 369
CompleteStart	DebugPrint
Mark3::Kernel, 142	Mark3::Kernel, 142
ConditionVariable_Broadcast	DebugPrintFunction
mark3c.cpp, 308	kernel.h, 503
mark3c.h, 383	DecimalToHex
ConditionVariable_Init	Mark3::MemUtil, 177, 178
mark3c.cpp, 309	DecimalToString
mark3c.h, 384	Mark3::MemUtil, 178, 179
ConditionVariable_Signal mark3c.cpp, 309	DestroyObject Mark3::AutoAlloc, 106
mark3c.h, 384	DestroyRawData
ConditionVariable_TimedWait	Mark3::AutoAlloc, 106
mark3c.cpp, 310	DestroyUserTypeAllocation
mark3c.h, 384	Mark3::AutoAlloc, 107
ConditionVariable Wait	DoubleLinkList
mark3c.cpp, 310	Mark3::DoubleLinkList, 134
mark3c.h, 385	Mark3::LinkListNode, 157
ConditionVariable t	WartoEllinelistivode, 107
mark3c.h, 372	Enter
Config	Mark3::CriticalSection, 132
Mark3::KernelSWI, 150	EventFlagOperation
Mark3::KernelTimer, 151	Mark3, 98
ContextSwitchSWI	Exit
Mark3::Thread, 246	Mark3, 98
CoopYield	Mark3::CriticalSection, 132
Mark3::Thread, 246	Mark3::Thread, 246
CopyData	
Mark3::Mailbox, 163	Free
CopyMemory	Mark3::AutoAlloc, 107
Mark3::MemUtil, 176	Free_ConditionVariable
CopyString	mark3c.cpp, 311
Mark3::MemUtil, 177	Free_EventFlag
CoroutineHandler	mark3c.cpp, 311
Mark3, 96	Free_Mailbox
CriticalGuard	mark3c.cpp, 311
Mark3::CriticalGuard, 131	mark3c.h, 385
DECLARE CONDITIONIVARIABLE	Free_Memory
DECLARE_CONDITIONVARIABLE	mark3c.cpp, 311 mark3c.h, 386
mark3c.h, 366 DECLARE MAILBOX	Free_Message
mark3c.h, 366	mark3c.cpp, 312
DECLARE MESSAGEPOOL	mark3c.cpp, 312 mark3c.h, 386
mark3c.h, 367	Free_MessagePool
DECLARE MESSAGEQUEUE	mark3c.cpp, 312
mark3c.h, 367	mark3c.h, 386
DECLARE MESSAGE	Free_MessageQueue
mark3c.h, 366	mark3c.cpp, 312
DECLARE MUTEX	mark3c.h, 386
mark3c.h, 367	Free_Mutex
DECLARE NOTIFY	mark3c.cpp, 312
mark3c.h, 368	mark3c.h, 386
DECLARE READERWRITERLOCK	Free_Notify
<del>-</del>	
111a1K3C.11, 300	·
mark3c.h, 368 DECLARE SEMAPHORE	mark3c.cpp, 312 mark3c.h, 387

5 B   W"	M 10 T ID 111111 000
Free_ReaderWriterLock	Mark3::TypedDoubleLinkList, 292
mark3c.cpp, 312	GetHeadPointer
Free_Semaphore	Mark3::Mailbox, 163
mark3c.cpp, 313	GetID
mark3c.h, 387	Mark3::Thread, 248
Free_Thread	GetMask
mark3c.cpp, 313	Mark3::EventFlag, 137
mark3c.h, 387	GetName
Free Timer	Mark3::Thread, 248
mark3c.cpp, 313	GetNext
mark3c.h, 387	Mark3::LinkListNode, 156
	Mark3::TypedLinkListNode, 294
g_kwCriticalCount	GetNextThread
Mark3, 100	Mark3::Scheduler, 223
threadport.cpp, 453	GetOwner
g_kwSFR	
Mark3, 100	Mark3::Thread, 249
threadport.cpp, 453	GetPrev
g pclCurrent	Mark3::LinkListNode, 157
scheduler.cpp, 556	Mark3::TypedLinkListNode, 294
• • •	GetPrioMap
scheduler.h, 540	Mark3::CoScheduler, 129
g_pclNext	GetPriority
scheduler.cpp, 556	Mark3::Coroutine, 125
scheduler.h, 540	Mark3::Thread, 249
GetAvailable	GetQuantum
Mark3::Streamer, 237	Mark3::Thread, 249
GetAverage	GetStack
Mark3::ProfileTimer, 211	Mark3::Thread, 250
GetCoList	GetStackGuardThreshold
Mark3::CoScheduler, 128	
GetCode	Mark3::Kernel, 143
Mark3::Message, 183	GetStackSize
GetCount	Mark3::Thread, 250
Mark3::MessageQueue, 190	GetStackSlack
<del>-</del>	Mark3::Thread, 250
Mark3::Semaphore, 230	GetState
GetCurPriority	Mark3::Thread, 251
Mark3::Thread, 246	GetStopList
GetCurrent	Mark3::CoScheduler, 129
Mark3::ProfileTimer, 211	Mark3::Scheduler, 223
Mark3::Thread, 247	GetTail
GetCurrentThread	Mark3::LinkList, 153
Mark3::Scheduler, 223	Mark3::TypedCircularLinkList, 288
GetData	Mark3::TypedDoubleLinkList, 292
Mark3::Message, 183	GetTailPointer
GetEventFlagMask	
Mark3::Thread, 247	Mark3::Mailbox, 164
GetEventFlagMode	GetThreadContextSwitchCallout
Mark3::Thread, 247	Mark3::Kernel, 143
	GetThreadCreateCallout
GetExpired 040	Mark3::Kernel, 143
Mark3::Thread, 248	GetThreadExitCallout
GetExtendedContext	Mark3::Kernel, 143
Mark3::Thread, 248	GetThreadList
GetFreeSlots	Mark3::Scheduler, 224
Mark3::Mailbox, 163	GetTicks
GetHead	Mark3::Kernel, 143
Mark3::LinkList, 153	GetTimer
Mark3::MessagePool, 187	Mark3::Thread, 251
Mark3::ThreadListList, 271	
Mark3::TypedCircularLinkList, 288	HighestPriority

Mark3::PriorityMapL1, 204	IsStarted
Mark3::PriorityMapL2, 208	Mark3::Kernel, 144
HighestWaiter	K ADDD
Mark3::ThreadList, 267	K_ADDR
IOD	portcfg.h, 443
ISR	K_INT
kerneltimer.cpp, 440	portcfg.h, 443
Mark3, 99	K_WORD
IdleFunc	portcfg.h, 443
Mark3, 96	KERNEL_ASSERT
InheritPriority	kerneldebug.h, 505
Mark3::Thread, 251	KERNEL_CONTEXT_SWITCH_CALLOUT
Init	mark3cfg.h, 519
Mark3::AutoAlloc, 107	KERNEL_DEBUG
Mark3::CoScheduler, 129	mark3cfg.h, 519
Mark3::ConditionVariable, 121	KERNEL_EVENT_FLAGS
Mark3::Coroutine, 125	mark3cfg.h, 519
Mark3::EventFlag, 137	KERNEL_EXTENDED_CONTEXT
Mark3::Kernel, 144	mark3cfg.h, 520
Mark3::LinkList, 153	KERNEL_NAMED_THREADS
Mark3::Mailbox, 164	mark3cfg.h, 520
Mark3::Message, 184	KERNEL_NUM_PRIORITIES
Mark3::MessagePool, 187	portcfg.h, 443
Mark3::MessageQueue, 190	KERNEL_ROUND_ROBIN
Mark3::Mutex, 196	mark3cfg.h, 520
Mark3::Notify, 200	KERNEL_STACK_CHECK
Mark3::ProfileTimer, 211	mark3cfg.h, 520
Mark3::Quantum, 214	KERNEL_STACK_GUARD_DEFAULT
Mark3::ReaderWriterLock, 220	portcfg.h, 444
Mark3::Scheduler, 224	KERNEL_THREAD_CREATE_CALLOUT
Mark3::Semaphore, 230	mark3cfg.h, 521
Mark3::Streamer, 237	KERNEL_THREAD_EXIT_CALLOUT
Mark3::Thread, 252	mark3cfg.h, 521
Mark3::ThreadPort, 272	KERNEL_TIMERS_THREAD_PRIORITY
Mark3::Timer, 276	portcfg.h, 444
Mark3::TimerList, 281	kernel.h
Mark3::TimerScheduler, 284	DebugPrintFunction, 503
InitStack	Kernel_DebugPrint
Mark3::ThreadPort, 272	mark3c.cpp, 313
InsertNodeBefore	mark3c.h, 387
Mark3::CircularLinkList, 116	Kernel_GetStackGuardThreshold
Mark3::TypedCircularLinkList, 288	mark3c.cpp, 314
Invalid	Kernel_GetTicks
Mark3, 98	mark3c.cpp, 314
isAcquired	mark3c.h, 388
Mark3::LockGuard, 160	Kernel_Init
IsEmpty	mark3c.cpp, 314
Mark3::Mailbox, 165	mark3c.h, 388
Mark3::Streamer, 237	Kernel_IsPanic
IsEnabled	mark3c.cpp, 314
Mark3::Scheduler, 224	mark3c.h, 388
IsFull	Kernel_IsStarted
Mark3::Mailbox, 165	mark3c.cpp, 315
IsInitialized	mark3c.h, 389
Mark3::BlockingObject, 112	Kernel_Panic
Mark3::Thread, 253	mark3c.cpp, 315
Mark3::Timer, 276	mark3c.h, 389
IsPanic	Kernel_SetDebugPrintFunction
Mark3::Kernel, 144	mark3c.cpp, 316

mark3c.h, 390	m bSchedState
Kernel SetPanic	Mark3::SchedulerGuard, 228
mark3c.cpp, 316	m bTimerActive
mark3c.h, 390	Mark3::TimerList, 282
Kernel SetStackGuardThreshold	
<del>-</del>	m_clBlockList
mark3c.cpp, 316	Mark3::BlockingObject, 114
Kernel_Start	m_clGlobalMutex
mark3c.cpp, 317	Mark3::ReaderWriterLock, 221
mark3c.h, 390	m_clLinkList
kernel_debug_print_t	Mark3::MessageQueue, 192
mark3c.h, 372	m_clList
KernelTimer_Task	Mark3::MessagePool, 188
Mark3, 99	m_clMutex
kerneldebug.h	Mark3::ConditionVariable, 123
KERNEL_ASSERT, 505	Mark3::TimerList, 282
kerneltimer.cpp	m_clPrioMap
ISR, 440	Mark3::CoScheduler, 130
TCCR1B_INIT, 440	Mark3::Scheduler, 226
TIMER_IFR, 440	m clReaderMutex
TIMER_IMSK, 440	Mark3::ReaderWriterLock, 221
	m clRecvSem
LinkList	Mark3::Mailbox, 171
Mark3::LinkListNode, 157	m clSemaphore
LinkListNode	Mark3::ConditionVariable, 123
Mark3::LinkListNode, 156	
Lock	Mark3::MessageQueue, 193
Mark3::Streamer, 238	m_clSendSem
LockGuard	Mark3::Mailbox, 171
Mark3::LockGuard, 159	m_clStopList
	Mark3::CoScheduler, 130
m_aclPriorities	Mark3::Scheduler, 227
Mark3::CoScheduler, 130	m_clThreadListList
Mark3::Scheduler, 226	Mark3::ThreadListList, 271
m_auXPriorityMap	m_clTimer
Mark3::PriorityMapL2, 209	Mark3::Thread, 260
m bActive	m_clTimerList
Mark3::ProfileTimer, 212	Mark3::TimerScheduler, 285
m bEnabled	m_eFlagMode
Mark3::Scheduler, 226	Mark3::Thread, 260
m_bExpired	m_eState
Mark3::Thread, 260	Mark3::Thread, 260
m blnTimer	m_pau8Buffer
Mark3::Quantum, 216	Mark3::Streamer, 240
m_blsAcquired	m_pclActiveThread
Mark3::LockGuard, 160	Mark3::Quantum, 216
m blsPanic	m pclCurrent
Mark3::Kernel, 148	Mark3::Thread, 261
m blsStarted	m pclHead
<del>_</del>	Mark3::LinkList, 154
Mark3::Kernel, 148	m_pclMap
m_bPending	Mark3::ThreadList, 269
Mark3::Notify, 202	
m_bQueued	m_pclMutex
Mark3::Coroutine, 126	Mark3::LockGuard, 160
m_bQueuedSchedule	m_pclOwner
Mark3::Scheduler, 226	Mark3::Coroutine, 126
m_bReady	Mark3::Mutex, 198
Mark3::Mutex, 197	Mark3::Thread, 261
m_bRecursive	Mark3::Timer, 278
Mark3::Mutex, 197	m_pclPrioMap

Mark3::CoList, 120	m_u16Head
m_pclTail	Mark3::Mailbox, 172
Mark3::LinkList, 155	Mark3::Streamer, 241
m_pclTimerThread	m_u16Iterations
Mark3::Quantum, 216	Mark3::ProfileTimer, 212
m_pfAllocator	m_u16MaxValue
Mark3::AutoAlloc, 109	Mark3::Semaphore, 234
m pfCallback	m u16Quantum
Mark3::Timer, 278	Mark3::Thread, 262
m_pfDebugPrintFunction	m u16SetMask
Mark3::Kernel, 148	Mark3::EventFlag, 140
m_pfEntryPoint	m u16Size
Mark3::Thread, 261	Mark3::Streamer, 241
m pfFree	m u16StackSize
Mark3::AutoAlloc, 109	Mark3::Thread, 263
m_pfHandler	m_u16Tail
Mark3::Coroutine, 127	Mark3::Mailbox, 172
m_pfPanic	Mark3::Streamer, 241
Mark3::Kernel, 148	m_u16TicksRemain
m_pfThreadContextCallout	Mark3::Quantum, 216
Mark3::Kernel, 148	m_u16Value
m_pfThreadCreateCallout	Mark3::Semaphore, 234
Mark3::Kernel, 149	m_u32Cumulative
m_pfThreadExitCallout	Mark3::ProfileTimer, 213
Mark3::Kernel, 149	m_u32CurrentIteration
m_pu8LockAddr	Mark3::ProfileTimer, 213
Mark3::Streamer, 240	m u32Interval
m_pvArg	Mark3::Timer, 279
Mark3::Thread, 261	m_u32NextWakeup
m_pvBuffer	Mark3::TimerList, 283
Mark3::Mailbox, 171	m u32StartTicks
m_pvContext	Mark3::ProfileTimer, 213
Mark3::Coroutine, 127	m u32Ticks
	<del>-</del>
m_pvData	Mark3::Kernel, 149
Mark3::Message, 185	m_u32TimeLeft
Mark3::Timer, 279	Mark3::Timer, 279
m_pvExtendedContext	m_u8Flags
Mark3::Thread, 261	Mark3::Timer, 279
m_pwStack	m_u8Initialized
Mark3::Thread, 262	Mark3::BlockingObject, 114
m_pwStackTop	Mark3::Timer, 279
Mark3::Thread, 262	m_u8ReadCount
m_szName	Mark3::ReaderWriterLock, 221
Mark3::Thread, 262	m_u8Recurse
m_u16Avail	Mark3::Mutex, 198
Mark3::Streamer, 240	m u8ThreadID
m u16Code	Mark3::Thread, 263
Mark3::Message, 185	m u8Waiters
m u16Count	Mark3::ConditionVariable, 123
Mark3::Mailbox, 172	m_uBlockingInitCookie
m u16ElementSize	Mark3::BlockingObject, 114
Mark3::Mailbox, 172	m_uBlockingInvalidCookie
m_u16FlagMask	Mark3::BlockingObject, 114
Mark3::Thread, 262	m_uMaxPri
m_u16Free	Mark3::Mutex, 198
Mark3::Mailbox, 172	m_uNumPriorities
m_u16GuardThreshold	Mark3::Scheduler, 227
Mark3::Kernel, 149	m_uPriority

Mark3::CoList, 120	mark3c.h, 393
Mark3::Coroutine, 127	Mailbox_Send
m_uTimerInitCookie	mark3c.cpp, 320
Mark3::Timer, 280	mark3c.h, 394
m_uTimerInvalidCookie	Mailbox_SendTail
Mark3::Timer, 280	mark3c.cpp, 320
m_uXCurPriority	mark3c.h, 394
Mark3::Thread, 263	Mailbox_TimedReceive
m_uXPrioMapBitMask	mark3c.cpp, 321
Mark3::PriorityMapL1, 205	mark3c.h, 395
Mark3::PriorityMapL2, 209	Mailbox_TimedReceiveTail
m_uXPrioMapBits	mark3c.cpp, 321
Mark3::PriorityMapL1, 205	mark3c.h, 395
Mark3::PriorityMapL2, 209	Mailbox_TimedSend
m_uXPrioMapNumWords	mark3c.cpp, 322
Mark3::PriorityMapL2, 209	mark3c.h, 396
m_uXPrioMapShiftLUT	Mailbox_TimedSendTail
Mark3::PriorityMapL1, 205	mark3c.cpp, 322
Mark3::PriorityMapL2, 209	mark3c.h, 396
m_uXPrioMapWordShift	Mailbox_t
Mark3::PriorityMapL1, 205	mark3c.h, 372
Mark3::PriorityMapL2, 210	Mark3, 91
m_uXPriority	AutoAllocAllocator_t, 95
Mark3::Thread, 263	AutoAllocFree_t, 95
Mark3::ThreadList, 270	AutoAllocType, 97
m_uXPriorityMap	Blocked, 98
Mark3::PriorityMapL1, 205	CoPrioMap, 95
m_uXPriorityMapL2	CoroutineHandler, 96
Mark3::PriorityMapL2, 210	EventFlagOperation, 98
MAILBOX_SIZE	Exit, 98
mark3c.h, 369 MESSAGE SIZE	g_kwCriticalCount, 100
<del>-</del>	g_kwSFR, 100
mark3c.h, 369 MESSAGEPOOL SIZE	ISR, 99
mark3c.h, 370	IdleFunc, 96
MESSAGEQUEUE SIZE	Invalid, 98 KernelTimer_Task, 99
mark3c.h, 370	Message, 98
MSecondsToTicks	•
	Mutex, 98
Mark3::Timer, 276	naked, 100 Notify, 98
MUTEX_SIZE	PORT CLZ, 99
mark3c.h, 370 Mailbox GetFreeSlots	PORT_CLZ, 99 PORT_CS_ENTER, 99
mark3c.cpp, 317	PORT_CS_EXIT, 99
mark3c.h, 391	PORT CS NESTING, 99
Mailbox Init	PORT_CS_NESTING, 99 PORT_IRQ_DISABLE, 100
<del>_</del>	PORT_IRQ_ENABLE, 100
mark3c.cpp, 317	
mark3c.h, 391 Mailbox_IsEmpty	PanicFunc, 96 PriorityMap, 96
mark3c.cpp, 318	Raw, 98
• •	
mark3c.h, 392 Mailbox IsFull	Ready, 98 SR , 101
mark3c.cpp, 318	Sh_, 101 Semaphore, 98
mark3c.cpp, 318 mark3c.h, 392	Stop, 98
Mailbox Receive	Stop, 98 Thread, 98
mark3c.cpp, 319	Thread, 98 Thread_Switch, 100
mark3c.cpp, 319 mark3c.h, 393	Thread_Switch, 100 ThreadContextCallout, 96
Mailbox_ReceiveTail mark3c.cpp, 319	ThreadCreateCallout, 96 ThreadEntryFunc, 97
markoc.opp, o 19	inieauchityrunc, 9/

ThreadExitCallout, 97	GetPrioMap, 129
ThreadState, 98	GetStopList, 129
Timer, 98	Init, 129
TimerCallback, 97	m aclPriorities, 130
uMaxTimerTicks, 101	m clPrioMap, 130
uTimerFlagActive, 101	m_clStopList, 130
uTimerFlagCallback, 101	Schedule, 129
uTimerFlagExpired, 101	Mark3::ConditionVariable, 120
uTimerFlagOneShot, 102	Broadcast, 121
· ·	
uTimerTicksInvalid, 102	Init, 121
User, 98	m_clMutex, 123
Mark3::Atomic, 102	m_clSemaphore, 123
Add, 103	m_u8Waiters, 123
Set, 103	operator new, 122
Sub, 104	Signal, 122
TestAndSet, 104	Wait, 122
Mark3::AutoAlloc, 105	Mark3::Coroutine, 124
Allocate, 106	$\sim$ Coroutine, 125
DestroyObject, 106	Activate, 125
DestroyRawData, 106	GetPriority, 125
DestroyUserTypeAllocation, 107	Init, 125
Free, 107	m_bQueued, 126
Init, 107	m_pclOwner, 126
m_pfAllocator, 109	m pfHandler, 127
<del>_</del>	<del>-</del>
m_pfFree, 109	m_pvContext, 127
NewObject, 107	m_uPriority, 127
NewRawData, 108	Run, 126
NewUserTypeAllocation, 108	SetPriority, 126
SetAllocatorFunctions, 108	Mark3::CriticalGuard, 131
Mark3::BlockingObject, 110	$\sim$ CriticalGuard, 131
$\sim$ BlockingObject, 111	CriticalGuard, 131
Block, 111	Mark3::CriticalSection, 132
BlockPriority, 112	Enter, 132
BlockingObject, 111	Exit, 132
IsInitialized, 112	NestingCount, 133
m_clBlockList, 114	Mark3::DoubleLinkList, 133
m u8Initialized, 114	Add, 134
m uBlockingInitCookie, 114	DoubleLinkList, 134
m_uBlockingInvalidCookie, 114	operator new, 134
SetInitialized, 112	Remove, 135
UnBlock, 112	Mark3::EventFlag, 135
Mark3::CircularLinkList, 115	~EventFlag, 137
	3,
Add, 116	Clear, 137
CircularLinkList, 115	GetMask, 137
InsertNodeBefore, 116	Init, 137
operator new, 116	m_u16SetMask, 140
PivotBackward, 116	operator new, 137
PivotForward, 117	Set, 138
Remove, 117	Wait, 138, 139
Mark3::CoList, 117	Wait_i, 139
Add, 118	WakeMe, 140
m_pclPrioMap, 120	Mark3::Kernel, 140
m_uPriority, 120	CompleteStart, 142
Remove, 119	DebugPrint, 142
SetPrioMap, 119	GetStackGuardThreshold, 143
SetPriority, 119	GetThreadContextSwitchCallout, 143
Mark3::CoScheduler, 127	GetThreadCreateCallout, 143
GetCoList, 128	GetThreadCreateCallout, 143 GetThreadExitCallout, 143
GGIOULISI, 120	GETTHEAULAROUR, 140

0.7	
GetTicks, 143	GetHeadPointer, 163
Init, 144	GetTailPointer, 164
IsPanic, 144	Init, 164
IsStarted, 144	IsEmpty, 165
m_blsPanic, 148	IsFull, 165
m_blsStarted, 148	m_clRecvSem, 171
m_pfDebugPrintFunction, 148	m_clSendSem, 171
m_pfPanic, 148	m_pvBuffer, 171
m_pfThreadContextCallout, 148	m_u16Count, 172
m_pfThreadCreateCallout, 149	m_u16ElementSize, 172
m_pfThreadExitCallout, 149	m_u16Free, 172
m_u16GuardThreshold, 149	m_u16Head, 172
m_u32Ticks, 149	m_u16Tail, 172
Panic, 144	MoveHeadBackward, 165
SetDebugPrintFunction, 145	MoveHeadForward, 165
SetPanic, 145	MoveTailBackward, 166
SetStackGuardThreshold, 145	MoveTailForward, 166
SetThreadContextSwitchCallout, 146	operator new, 166
SetThreadCreateCallout, 146	Receive, 166, 167
SetThreadExitCallout, 147	Receive_i, 167
Start, 147	ReceiveTail, 168
Tick, 147	Send, 168, 169
Mark3::KernelSWI, 150	Send_i, 169
Config, 150	SendTail, 170
Start, 150	Mark3::MemUtil, 173
Trigger, 150	Checksum16, 174
Mark3::KernelTimer, 151	Checksum8, 174
Config, 151	CompareMemory, 175
Start, 151	CompareStrings, 175, 176
Stop, 152	CopyMemory, 176
Mark3::LinkList, 152	CopyString, 177
GetHead, 153	DecimalToHex, 177, 178
GetTail, 153	DecimalToString, 178, 179
Init, 153	SetMemory, 179
m_pclHead, 154	StringLength, 180
m_pclTail, 155	StringSearch, 180
SetHead, 154	StringToDecimal16, 180
SetTail, 154	StringToDecimal32, 181
Mark3::LinkListNode, 155	StringToDecimal64, 181
CircularLinkList, 157	StringToDecimal8, 181
ClearNode, 156	Tokenize, 182
DoubleLinkList, 157	Mark3::Message, 182
GetNext, 156	GetCode, 183
GetPrev, 157	GetData, 183
LinkList, 157	Init, 184
LinkListNode, 156	m_pvData, 185
next, 158	m_u16Code, 185
prev, 158	operator new, 184
Mark3::LockGuard, 158	SetCode, 184
$\sim$ LockGuard, 159	SetData, 185
isAcquired, 160	Mark3::MessagePool, 186
LockGuard, 159	~MessagePool, 186
m_blsAcquired, 160	GetHead, 187
m_pclMutex, 160	Init, 187
Mark3::Mailbox, 161	m_clList, 188
$\sim$ Mailbox, 163	operator new, 187
CopyData, 163	Pop, 187
GetFreeSlots, 163	Push, 188

Mark3::MessageQueue, 189	PriorityFromBitmap, 208
$\sim$ MessageQueue, 190	PriorityMapL2, 207
GetCount, 190	Set, 208
Init, 190	Mark3::PriorityMapL2< T, C >, 206
m_clLinkList, 192	Mark3::ProfileTimer, 210
m_clSemaphore, 193	GetAverage, 211
operator new, 190	GetCurrent, 211
Receive, 191	Init, 211
Receive i, 191	m_bActive, 212
Send, 192	m_u16Iterations, 212
Mark3::Mutex, 193	m_u32Cumulative, 213
$\sim$ Mutex, 194	m_u32CurrentIteration, 213
Claim, 194, 195	m_u32StartTicks, 213
Claim i, 195	Start, 212
Init, 196	Stop, 212
m bReady, 197	Mark3::Quantum, 213
m bRecursive, 197	Cancel, 214
m pclOwner, 198	ClearInTimer, 214
m_u8Recurse, 198	Init, 214
m uMaxPri, 198	m_blnTimer, 216
operator new, 196	m pclActiveThread, 216
Release, 196	m_pclActiveThread, 216
WakeMe, 197	<b>—</b>
•	m_u16TicksRemain, 216
WakeNext, 197	SetInTimer, 215
Mark3::Notify, 199	SetTimerThread, 215
~Notify, 200	Update, 215
Init, 200	Mark3::ReaderWriterLock, 216
m_bPending, 202	AcquireReader, 218
operator new, 200	AcquireReader_i, 218
Signal, 200	AcquireWriter, 219
Wait, 200, 201	AcquireWriter_i, 219
WakeMe, 201	Init, 220
Mark3::PriorityMapL1	m_clGlobalMutex, 221
Clear, 203	m_clReaderMutex, 221
HighestPriority, 204	m_u8ReadCount, 221
m_uXPrioMapBitMask, 205	operator new, 220
m_uXPrioMapBits, 205	ReleaseReader, 220
m_uXPrioMapShiftLUT, 205	ReleaseWriter, 220
m_uXPrioMapWordShift, 205	Mark3::Scheduler, 221
m_uXPriorityMap, 205	Add, 223
PrioBit, 204	GetCurrentThread, 223
PrioMapWordIndex, 204	GetNextThread, 223
PriorityFromBitmap, 204	GetStopList, 223
PriorityMapL1, 203	GetThreadList, 224
Set, 204	Init, 224
Mark3::PriorityMapL1 $<$ T, C $>$ , 202	IsEnabled, 224
Mark3::PriorityMapL2	m_aclPriorities, 226
Clear, 207	m_bEnabled, 226
HighestPriority, 208	m_bQueuedSchedule, 226
m_auXPriorityMap, 209	m_clPrioMap, 226
m_uXPrioMapBitMask, 209	m_clStopList, 227
m_uXPrioMapBits, 209	m_uNumPriorities, 227
m_uXPrioMapNumWords, 209	QueueScheduler, 225
m_uXPrioMapShiftLUT, 209	Remove, 225
m_uXPrioMapWordShift, 210	Schedule, 225
m_uXPriorityMapL2, 210	SetScheduler, 225
PrioBit, 208	Mark3::SchedulerGuard, 227
PrioMapWordIndex, 208	$\sim$ SchedulerGuard, 228

m_bSchedState, 228	m_eState, 260
SchedulerGuard, 228	m_pclCurrent, 261
Mark3::Semaphore, 229	m_pclOwner, 261
$\sim$ Semaphore, 230	m_pfEntryPoint, 261
GetCount, 230	m_pvArg, 261
Init, 230	m_pvExtendedContext, 261
m_u16MaxValue, 234	m_pwStack, 262
m_u16Value, 234	m_pwStackTop, 262
operator new, 231	m_szName, 262
Pend, 231	m_u16FlagMask, 262
Pend_i, 231	m_u16Quantum, 262
Post, 233	m_u16StackSize, 263
WakeMe, 233	m_u8ThreadID, 263
WakeNext, 233	m_uXCurPriority, 263
Mark3::Streamer, 234	m_uXPriority, 263
CanRead, 236	operator new, 253
CanWrite, 236	SetCurrent, 253
Claim, 236	SetEventFlagMask, 254
GetAvailable, 237	SetEventFlagMode, 254
Init, 237	SetExpired, 254
IsEmpty, 237	SetExtendedContext, 255
Lock, 238	SetID, 255
m pau8Buffer, 240	SetName, 255
m pu8LockAddr, 240	SetOwner, 257
m u16Avail, 240	SetPriority, 257
m_u16Head, 241	SetPriorityBase, 257
m u16Size, <mark>241</mark>	SetQuantum, 258
m u16Tail, <mark>241</mark>	SetState, 258
Read, 238	Sleep, 258
Unlock, 239	Start, 259
Write, 239, 240	Stop, 259
Mark3::Thread, 242	Thread, 245
$\sim$ Thread, 245	ThreadPort, 260
ContextSwitchSWI, 246	Yield, 259
CoopYield, 246	Mark3::ThreadList, 264
Exit, 246	Add, 265
GetCurPriority, 246	AddPriority, 267
GetCurrent, 247	HighestWaiter, 267
GetEventFlagMask, 247	m pclMap, 269
GetEventFlagMode, 247	m uXPriority, 270
GetExpired, 248	operator new, 267
GetExtendedContext, 248	Remove, 267
GetID, 248	SetMapPointer, 269
GetName, 248	SetPriority, 269
GetOwner, 249	ThreadList, 265
GetPriority, 249	Mark3::ThreadListList, 270
GetQuantum, 249	Add, 270
GetStack, 250	GetHead, 271
GetStackSize, 250	m_clThreadListList, 271
	Remove, 271
GetStackSlack, 250	
GetState, 251	Mark3::ThreadPort, 272
GetTimer, 251	Init, 272
InheritPriority, 251	InitStack, 272
Init, 252	StartThreads, 273
IsInitialized, 253	Thread, 273
m_bExpired, 260	Mark3::Timer, 273
m_clTimer, 260	∼Timer, 275
m_eFlagMode, 260	Init, 276

IsInitialized, 276	GetNext, 294
m_pclOwner, 278	GetPrev, 294
m_pfCallback, 278	Mark3::TypedLinkListNode $<$ T $>$ , 293
m_pvData, 279	mark3c.cpp
m_u32Interval, 279	Alloc_ConditionVariable, 300
m_u32TimeLeft, 279	Alloc_EventFlag, 300
m_u8Flags, 279	Alloc_Mailbox, 300
m_u8Initialized, 279	Alloc_Memory, 300
m_uTimerInitCookie, 280	Alloc_Message, 300
m_uTimerInvalidCookie, 280	Alloc_MessagePool, 301
MSecondsToTicks, 276	Alloc_MessageQueue, 301
operator new, 276	Alloc_Mutex, 301
SecondsToTicks, 276	Alloc_Notify, 302
SetInitialized, 277	Alloc_ReaderWriterLock, 302
Start, 277	Alloc_Semaphore, 302
Stop, 278	Alloc_Thread, 303
Timer, 275	Alloc_Timer, 303
TimerList, 278	Atomic_Add16, 303
USecondsToTicks, 278	Atomic_Add32, 304
Mark3::TimerList, 280	Atomic_Add8, 304
Add, 281	Atomic_Set16, 305
Init, 281	Atomic_Set32, 305
m_bTimerActive, 282	Atomic_Set8, 306
m_clMutex, 282	Atomic_Sub16, 306
m_u32NextWakeup, 283	Atomic_Sub32, 307
Process, 282	Atomic_Sub8, 307
Remove, 282	Atomic_TestAndSet, 308
Mark3::TimerScheduler, 283	ConditionVariable_Broadcast, 308
Add, 284	ConditionVariable_Init, 309
Init, 284	ConditionVariable_Signal, 309
m_clTimerList, 285	ConditionVariable_TimedWait, 310
Process, 284	ConditionVariable_Wait, 310
Remove, 284	Free_ConditionVariable, 311
Mark3::Token_t, 285	Free_EventFlag, 311
pcToken, 286	Free_Mailbox, 311
u8Len, 286	Free_Memory, 311
Mark3::TypedCircularLinkList	Free_Message, 312
Add, 287	Free_MessagePool, 312
GetHead, 288	Free_MessageQueue, 312
GetTail, 288	Free_Mutex, 312
InsertNodeBefore, 288	Free_Notify, 312
operator new, 289	Free_ReaderWriterLock, 312
Remove, 289	Free_Semaphore, 313
SetHead, 289	Free_Thread, 313
SetTail, 290	Free_Timer, 313
TypedCircularLinkList, 287	Kernel_DebugPrint, 313
Mark3::TypedCircularLinkList< T >, 286	Kernel_GetStackGuardThreshold, 314
Mark3::TypedDoubleLinkList	Kernel_GetTicks, 314
Add, 291	Kernel_Init, 314
GetHead, 292	Kernel_IsPanic, 314
GetTail, 292	Kernel_IsStarted, 315
operator new, 292	Kernel_Panic, 315
Remove, 292	Kernel_SetDebugPrintFunction, 316
SetHead, 293	Kernel_SetPanic, 316
SetTail, 293	Kernel_SetStackGuardThreshold, 316
TypedDoubleLinkList, 291	Kernel_Start, 317
Mark3::TypedDoubleLinkList< T >, 290	Mailbox_GetFreeSlots, 317
Mark3::TypedLinkListNode	Mailbox_Init, 317

Maille and In Franchis 040	Thursd OstOventure 040
Mailbox_IsEmpty, 318	Thread_SetQuantum, 343
Mailbox_IsFull, 318	Thread_Sleep, 343
Mailbox_Receive, 319	Thread_Start, 344
Mailbox_ReceiveTail, 319	Thread_Stop, 344
Mailbox_Send, 320	Thread_Yield, 344
Mailbox_SendTail, 320	Timer_Init, 345
Mailbox_TimedReceive, 321	Timer_Restart, 345
Mailbox_TimedReceiveTail, 321	Timer_Start, 345
Mailbox_TimedSend, 322	Timer_Stop, 346
Mailbox_TimedSendTail, 322	mark3c.h
Message_GetCode, 323	Alloc_Mailbox, 375
Message_GetData, 323	Alloc_Memory, 375
Message_Init, 324	Alloc_Message, 376
Message_SetCode, 324	Alloc_MessagePool, 376
Message_SetData, 325	Alloc_MessageQueue, 376
MessagePool_Init, 325	Alloc_Mutex, 376
MessagePool_Pop, 326	Alloc_Notify, 377
MessagePool_Push, 326	Alloc_Semaphore, 377
MessageQueue_GetCount, 327	Alloc_Thread, 377
MessageQueue_Init, 327	Alloc_Timer, 378
MessageQueue_Receive, 327	Atomic_Add16, 378
MessageQueue_Send, 328	Atomic_Add32, 379
MessageQueue_TimedReceive, 328	Atomic_Add8, 379
Mutex_Claim, 329	Atomic_Set16, 380
Mutex_Init, 329	Atomic_Set32, 380
Mutex_Release, 330	Atomic_Set8, 381
Mutex_TimedClaim, 330	Atomic_Sub16, 381
Notify_Init, 330	Atomic_Sub32, 382
Notify_Signal, 331	Atomic_Sub8, 382
Notify_TimedWait, 331	Atomic_TestAndSet, 383
Notify_Wait, 332	CONDITIONVARIABLE_SIZE, 366
ReaderWriterLock_AcquireReader, 332	ConditionVariable_Broadcast, 383
ReaderWriterLock_AcquireWriter, 333	ConditionVariable_Init, 384
ReaderWriterLock_Init, 333	ConditionVariable_Signal, 384
ReaderWriterLock_ReleaseReader, 333	ConditionVariable_TimedWait, 384
ReaderWriterLock_ReleaseWriter, 334	ConditionVariable_Wait, 385
ReaderWriterLock_TimedAcquireReader, 334	ConditionVariable_t, 372
ReaderWriterLock_TimedAcquireWriter, 335	DECLARE_CONDITIONVARIABLE, 366
Scheduler_Enable, 335	DECLARE_MAILBOX, 366
Scheduler_GetCurrentThread, 336	DECLARE_MESSAGEPOOL, 367
Scheduler_IsEnabled, 336	DECLARE_MESSAGEQUEUE, 367
Semaphore_Init, 336	DECLARE_MESSAGE, 366
Semaphore_Pend, 337	DECLARE_MUTEX, 367
Semaphore_Post, 337	DECLARE_NOTIFY, 368
Semaphore_TimedPend, 337	DECLARE_READERWRITERLOCK, 368
Thread_CoopYield, 338	DECLARE_SEMAPHORE, 368
Thread_Exit, 338	DECLARE_THREAD, 369
Thread_GetCurPriority, 339	DECLARE_TIMER, 369
Thread_GetID, 339	Free_Mailbox, 385
Thread_GetName, 340	Free_Memory, 386
Thread_GetPriority, 340	Free_Message, 386
Thread_GetQuantum, 340	Free_MessagePool, 386
Thread_GetStackSlack, 340	Free_MessageQueue, 386
Thread_GetState, 341	Free_Mutex, 386
Thread_Init, 341	Free_Notify, 387
Thread_SetID, 342	Free_Semaphore, 387
Thread SetName, 342	Free Thread, 387
Thread SetPriority, 342	Free Timer, 387
	<u> </u>

Kernel_DebugPrint, 387	ReaderWriterLock_AcquireWriter, 406
Kernel_GetTicks, 388	ReaderWriterLock_Init, 407
Kernel_Init, 388	ReaderWriterLock_ReleaseReader, 407
Kernel_IsPanic, 388	ReaderWriterLock_ReleaseWriter, 407
Kernel_IsStarted, 389	ReaderWriterLock_TimedAcquireReader, 408
Kernel_Panic, 389	ReaderWriterLock_TimedAcquireWriter, 408
Kernel_SetDebugPrintFunction, 390	ReaderWriterLock_t, 373
Kernel SetPanic, 390	SEMAPHORE SIZE, 370
Kernel_Start, 390	Scheduler_Enable, 409
kernel debug print t, 372	Scheduler_GetCurrentThread, 409
MAILBOX SIZE, 369	Scheduler IsEnabled, 410
MESSAGE_SIZE, 369	Semaphore Init, 410
MESSAGEPOOL_SIZE, 370	Semaphore_Pend, 410
	• —
MESSAGEQUEUE_SIZE, 370	Semaphore_Post, 411
MUTEX_SIZE, 370	Semaphore_TimedPend, 411
Mailbox_GetFreeSlots, 391	Semaphore_t, 373
Mailbox_Init, 391	THREAD_SIZE, 371
Mailbox_IsEmpty, 392	TIMER_SIZE, 371
Mailbox_IsFull, 392	TOKEN_1, 371
Mailbox_Receive, 393	TOKEN_2, 371
Mailbox_ReceiveTail, 393	Thread_CoopYield, 412
Mailbox_Send, 394	Thread_Exit, 412
Mailbox_SendTail, 394	Thread_GetCurPriority, 412
Mailbox_TimedReceive, 395	Thread_GetID, 413
Mailbox_TimedReceiveTail, 395	Thread_GetPriority, 413
Mailbox_TimedSend, 396	Thread_GetState, 414
Mailbox TimedSendTail, 396	Thread Init, 414
Mailbox_t, 372	Thread_SetID, 415
Message_GetCode, 397	Thread_SetPriority, 415
Message_GetData, 397	Thread_Sleep, 416
Message_Init, 398	Thread_Start, 416
Message_SetCode, 398	Thread_Stop, 416
Message_SetData, 399	Thread_Yield, 417
Message_t, 372	thread_entry_func_t, 374
MessagePool_Init, 399	thread state t, 374
MessagePool_Pop, 399	Thread_t, 374
· ·	
MessagePool_Push, 400	Timer_Init, 417
MessagePool_t, 372	Timer_Restart, 418
MessageQueue_GetCount, 400	Timer_Start, 418
MessageQueue_Init, 401	Timer_Stop, 419
MessageQueue_Receive, 401	timer_callback_t, 374
MessageQueue_Send, 401	Timer_t, 374
MessageQueue_TimedReceive, 402	WORD_ROUND, 371
MessageQueue_t, 372	mark3cfg.h
Mutex_Claim, 402	KERNEL_CONTEXT_SWITCH_CALLOUT, 519
Mutex_Init, 403	KERNEL_DEBUG, 519
Mutex_Release, 403	KERNEL_EVENT_FLAGS, 519
Mutex_TimedClaim, 404	KERNEL_EXTENDED_CONTEXT, 520
Mutex_t, 373	KERNEL_NAMED_THREADS, 520
NOTIFY_SIZE, 370	KERNEL_ROUND_ROBIN, 520
Notify_Init, 404	KERNEL_STACK_CHECK, 520
Notify_Signal, 405	KERNEL_THREAD_CREATE_CALLOUT, 521
Notify_TimedWait, 405	KERNEL_THREAD_EXIT_CALLOUT, 521
Notify_Wait, 405	Message
Notify_t, 373	Mark3, 98
panic_func_t, 373	Message_GetCode
READERWRITERLOCK_SIZE, 370	mark3c.cpp, 323
ReaderWriterLock_AcquireReader, 406	mark3c.h, 397
_ ·	

Message_GetData	Mutex_Release
mark3c.cpp, 323	mark3c.cpp, 330
mark3c.h, 397	mark3c.h, 403
Message_Init	Mutex_TimedClaim
mark3c.cpp, 324	mark3c.cpp, 330
mark3c.h, 398	mark3c.h, 404
Message_SetCode	Mutex_t
mark3c.cpp, 324	mark3c.h, 373
mark3c.h, 398	
Message_SetData	NOTIFY SIZE
mark3c.cpp, 325	mark3c.h, <mark>370</mark>
mark3c.h, 399	naked
Message_t	Mark3, 100
mark3c.h, 372	NestingCount
MessagePool_Init	Mark3::CriticalSection, 133
mark3c.cpp, 325	NewObject
mark3c.h, 399	Mark3::AutoAlloc, 107
MessagePool Pop	NewRawData
mark3c.cpp, 326	Mark3::AutoAlloc, 108
mark3c.h, 399	NewUserTypeAllocation
MessagePool_Push	Mark3::AutoAlloc, 108
mark3c.cpp, 326	next
mark3c.h, 400	
•	Mark3::LinkListNode, 158
MessagePool_t	Notify
mark3c.h, 372	Mark3, 98
MessageQueue_GetCount	Notify_Init
mark3c.cpp, 327	mark3c.cpp, 330
mark3c.h, 400	mark3c.h, 404
MessageQueue_Init	Notify_Signal
mark3c.cpp, 327	mark3c.cpp, 331
mark3c.h, 401	mark3c.h, 405
MessageQueue_Receive	Notify_TimedWait
mark3c.cpp, 327	mark3c.cpp, 331
mark3c.h, 401	mark3c.h, 405
MessageQueue_Send	Notify_Wait
mark3c.cpp, 328	mark3c.cpp, 332
mark3c.h, 401	mark3c.h, 405
MessageQueue_TimedReceive	Notify_t
mark3c.cpp, 328	mark3c.h, <mark>373</mark>
mark3c.h, 402	
MessageQueue_t	operator new
mark3c.h, 372	Mark3::CircularLinkList, 116
MoveHeadBackward	Mark3::ConditionVariable, 122
Mark3::Mailbox, 165	Mark3::DoubleLinkList, 134
MoveHeadForward	Mark3::EventFlag, 137
Mark3::Mailbox, 165	Mark3::Mailbox, 166
MoveTailBackward	Mark3::Message, 184
Mark3::Mailbox, 166	Mark3::MessagePool, 187
MoveTailForward	Mark3::MessageQueue, 190
Mark3::Mailbox, 166	Mark3::Mutex, 196
Mutex	Mark3::Notify, 200
Mark3, 98	Mark3::ReaderWriterLock, 220
Mutex Claim	Mark3::Semaphore, 231
mark3c.cpp, 329	Mark3::Thread, 253
mark3c.h, 402	Mark3::ThreadList, 267
Mutex Init	Mark3::Timer, 276
mark3c.cpp, 329	Mark3::TypedCircularLinkList, 289
mark3c.h, 403	Mark3::TypedDoubleLinkList, 292
manoon, 100	marron typodbodbioLiniclot, 202

PANIC_ACTIVE_COROUTINE_DESCOPED	PORT_SYSTEM_FREQ
paniccodes.h, 528	portcfg.h, 446
PANIC_ACTIVE_EVENTFLAG_DESCOPED	PORT_TIMER_COUNT_TYPE
paniccodes.h, 528	portcfg.h, 446
PANIC_ACTIVE_MAILBOX_DESCOPED	PORT_TIMER_FREQ
paniccodes.h, 528	portcfg.h, 446
PANIC_ACTIVE_MUTEX_DESCOPED	PORT_TOP_OF_STACK
paniccodes.h, 528	threadport.h, 449
PANIC_ACTIVE_NOTIFY_DESCOPED	PORT_USE_HW_CLZ
paniccodes.h, 528	portcfg.h, 446
PANIC ACTIVE SEMAPHORE DESCOPED	Panic
paniccodes.h, 528	Mark3::Kernel, 144
PANIC_ACTIVE_TIMER_DESCOPED	panic_func_t
paniccodes.h, 529	mark3c.h, 373
PANIC_ASSERT_FAILED	PanicFunc
paniccodes.h, 529	Mark3, 96
PANIC_AUTO_HEAP_EXHAUSTED	paniccodes.h
paniccodes.h, 529	PANIC_ACTIVE_COROUTINE_DESCOPED, 528
PANIC_LIST_UNLINK_FAILED	PANIC_ACTIVE_EVENTFLAG_DESCOPED, 528
paniccodes.h, 529	PANIC_ACTIVE_MAILBOX_DESCOPED, 528
PANIC_NO_READY_THREADS	PANIC_ACTIVE_MUTEX_DESCOPED, 528
paniccodes.h, 529	PANIC_ACTIVE_NOTIFY_DESCOPED, 528
PANIC_POWERMAN_EXHAUSTED	PANIC ACTIVE SEMAPHORE DESCOPED, 528
paniccodes.h, 529	PANIC ACTIVE TIMER DESCOPED, 529
PANIC_RUNNING_THREAD_DESCOPED	PANIC_ASSERT_FAILED, 529
paniccodes.h, 530	PANIC_AUTO_HEAP_EXHAUSTED, 529
PANIC_STACK_SLACK_VIOLATED	PANIC LIST UNLINK FAILED, 529
paniccodes.h, 530	PANIC_NO_READY_THREADS, 529
PORT_CLZ	PANIC_POWERMAN_EXHAUSTED, 529
Mark3, 99	PANIC_RUNNING_THREAD_DESCOPED, 530
PORT_COROUTINE_PRIORITIES	PANIC_STACK_SLACK_VIOLATED, 530
portcfg.h, 444	pcToken
PORT_CS_ENTER	Mark3::Token_t, 286
Mark3, 99	Pend
PORT_CS_EXIT	Mark3::Semaphore, 231
Mark3, 99	Pend_i
PORT_CS_NESTING	Mark3::Semaphore, 231
Mark3, 99	PivotBackward
PORT_IRQ_DISABLE	Mark3::CircularLinkList, 116
Mark3, 100	PivotForward
PORT_IRQ_ENABLE	Mark3::CircularLinkList, 117
Mark3, 100	Pop
PORT KERNEL DEFAULT STACK SIZE	Mark3::MessagePool, 187
portcfg.h, 444	•
•	portofg.h
PORT_KERNEL_TIMERS_THREAD_STACK	AVR, 443
portcfg.h, 444	K_ADDR, 443
PORT_MIN_TIMER_TICKS	K_INT, 443
portcfg.h, 445	K_WORD, 443
PORT_OVERLOAD_NEW	KERNEL_NUM_PRIORITIES, 443
portcfg.h, 445	KERNEL_STACK_GUARD_DEFAULT, 444
PORT_PRIO_MAP_WORD_SIZE	KERNEL_TIMERS_THREAD_PRIORITY, 444
portcfg.h, 445	PORT_COROUTINE_PRIORITIES, 444
PORT_PRIO_TYPE	PORT_KERNEL_DEFAULT_STACK_SIZE, 444
portcfg.h, 445	PORT_KERNEL_TIMERS_THREAD_STACK, 444
PORT_PUSH_TO_STACK	PORT MIN TIMER TICKS, 445
threadport.h, 449	PORT_OVERLOAD_NEW, 445
PORT_STACK_GROWS_DOWN	PORT_PRIO_MAP_WORD_SIZE, 445
portcfg.h, 445	PORT PRIO TYPE, 445
portoight, <del>170</del>	· OIII_I IIIO_I I I L, 770

PORT_STACK_GROWS_DOWN, 445	mark3c.cpp, 335
PORT_SYSTEM_FREQ, 446	mark3c.h, 408
PORT_TIMER_COUNT_TYPE, 446	ReaderWriterLock_t
PORT_TIMER_FREQ, 446	mark3c.h, 373
PORT USE HW CLZ, 446	Ready
THREAD QUANTUM DEFAULT, 447	Mark3, 98
Post	Receive
Mark3::Semaphore, 233	Mark3::Mailbox, 166, 167
prev	Mark3::MessageQueue, 191
Mark3::LinkListNode, 158	Receive i
PrioBit	Mark3::Mailbox, 167
Mark3::PriorityMapL1, 204	Mark3::MessageQueue, 191
Mark3::PriorityMapL2, 208	ReceiveTail
PrioMapWordIndex	Mark3::Mailbox, 168
Mark3::PriorityMapL1, 204	Release
Mark3::PriorityMapL2, 208	Mark3::Mutex, 196
PriorityFromBitmap	ReleaseReader
· · · · · · · · · · · · · · · · · · ·	
Mark3::PriorityMapL1, 204	Mark3::ReaderWriterLock, 220
Mark3::PriorityMapL2, 208	ReleaseWriter
PriorityMap	Mark3::ReaderWriterLock, 220
Mark3, 96	Remove
PriorityMapL1	Mark3::CircularLinkList, 117
Mark3::PriorityMapL1, 203	Mark3::CoList, 119
PriorityMapL2	Mark3::DoubleLinkList, 135
Mark3::PriorityMapL2, 207	Mark3::Scheduler, 225
Process	Mark3::ThreadList, 267
Mark3::TimerList, 282	Mark3::ThreadListList, 271
Mark3::TimerScheduler, 284	Mark3::TimerList, 282
Push	Mark3::TimerScheduler, 284
Mayle Ou Magagara Dool 100	Mankout and down last interest of the
Mark3::MessagePool, 188	Mark3::TypedCircularLinkList, 289
-	Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 292
QueueScheduler	•••
-	Mark3::TypedDoubleLinkList, 292
QueueScheduler Mark3::Scheduler, 225	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126 SEMAPHORE_SIZE
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126 SEMAPHORE_SIZE mark3c.h, 370
QueueScheduler Mark3::Scheduler, 225 READERWRITERLOCK_SIZE mark3c.h, 370 Raw	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370 SR_
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101
QueueScheduler Mark3::Scheduler, 225 READERWRITERLOCK_SIZE mark3c.h, 370 Raw	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370 SR_
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370 SR_ Mark3, 101 Schedule
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225 scheduler.cpp
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370 SR_ Mark3, 101 Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225 scheduler.cpp g_pclCurrent, 556
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370 SR_ Mark3, 101 Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225 scheduler.cpp g_pclCurrent, 556 g_pclNext, 556
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter mark3c.cpp, 334	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409  Scheduler_IsEnabled
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter mark3c.cpp, 334 mark3c.h, 407	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409  Scheduler_IsEnabled mark3c.cpp, 336
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter mark3c.cpp, 334 mark3c.h, 407  ReaderWriterLock_TimedAcquireReader	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409  Scheduler_IsEnabled mark3c.cpp, 336 mark3c.h, 410
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter mark3c.cpp, 334 mark3c.h, 407  ReaderWriterLock_TimedAcquireReader mark3c.cpp, 334 mark3c.h, 407  ReaderWriterLock_TimedAcquireReader mark3c.cpp, 334	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409  Scheduler_IsEnabled mark3c.cpp, 336 mark3c.h, 410  SchedulerGuard
QueueScheduler Mark3::Scheduler, 225  READERWRITERLOCK_SIZE mark3c.h, 370  Raw Mark3, 98  Read Mark3::Streamer, 238  ReaderWriterLock_AcquireReader mark3c.cpp, 332 mark3c.h, 406  ReaderWriterLock_AcquireWriter mark3c.cpp, 333 mark3c.h, 406  ReaderWriterLock_Init mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseReader mark3c.cpp, 333 mark3c.h, 407  ReaderWriterLock_ReleaseWriter mark3c.cpp, 334 mark3c.h, 407  ReaderWriterLock_TimedAcquireReader	Mark3::TypedDoubleLinkList, 292 Run Mark3::Coroutine, 126  SEMAPHORE_SIZE mark3c.h, 370  SR_ Mark3, 101  Schedule Mark3::CoScheduler, 129 Mark3::Scheduler, 225  scheduler.cpp g_pclCurrent, 556 g_pclNext, 556  scheduler.h g_pclCurrent, 540 g_pclNext, 540  Scheduler_Enable mark3c.cpp, 335 mark3c.h, 409  Scheduler_GetCurrentThread mark3c.cpp, 336 mark3c.h, 409  Scheduler_IsEnabled mark3c.cpp, 336 mark3c.h, 410

Mark3::Timer, 276         SetMapPointer           Semaphore         Mark3::ThreadList, 269           Semaphore_Init         Mark3::MemUtil, 179           mark3c.pp, 336         Mark3::MemUtil, 179           semaphore_Pend         Mark3::Thread, 255           setOwner         Mark3::Thread, 257           setOwner         Mark3::Thread, 257           setPrioMap         Mark3::Thread, 257           setPanic         Mark3::CoList, 119           setPriority         Mark3::Coroutine, 126           mark3c.h, 411         Mark3::Coroutine, 126           setPriority         Mark3::Thread, 257           setPriority         Mark3::Thread, 258           setPriorityBase         SetPriorityBase           setPriorityBase         Mark3::Thread, 258           setStackGuardThreshold         Mark3::Thread, 258 </th <th></th> <th></th>		
Mark3, 98         SetMemory           Semaphore_Init mark3c.cpp, 336 mark3c.h, 410         SetName Mark3::Thread, 255           Semaphore_Pend mark3c.pp, 337 mark3c.h, 410         SetOwner Mark3::Thread, 257           Semaphore_Post mark3c.cpp, 337 mark3c.h, 411         Mark3::Thread, 257           Semaphore_TimedPend mark3c.cpp, 337 mark3c.h, 411         Mark3::CoList, 119           Semaphore_I mark3c.h, 411         Mark3::CoList, 119           Semaphore_I mark3c.h, 411         Mark3::CoList, 119           Semaphore_I mark3c.h, 411         Mark3::Thread, 257           Semaphore_I mark3c.h, 411         Mark3::Thread, 257           Semaphore_I mark3c.h, 411         Mark3::Thread, 257           Send Mark3::Mailbox, 168, 169 Mark3::Mailbox, 168, 169 Mark3::Mailbox, 169         Mark3::Thread, 257           Send Ji Mark3::Mailbox, 170         SetScheduler           Set Send Ji Mark3::Mailbox, 170         SetScheduler           Set State Mark3::Scheduler, 225         SetState           Mark3::Mailbox, 170         SetStackGuardThreshold           Mark3::EventFlag, 138 Mark3::PriorityMapL1, 204 Mark3::Mersage, 184         SetTail           SetCode         Mark3::Mersage, 185           SetCode         Mark3::Mersage, 186           SetData Mark3::Message, 186         SetThreadCreateCallout Mark3::Kernel, 146           SetData Mark3::Thread, 254 </td <td>Mark3::Timer, 276</td> <td>SetMapPointer</td>	Mark3::Timer, 276	SetMapPointer
Semaphore_Init mark3c.pp, 336 mark3c.h, 410         SetName           Semaphore_Pend mark3c.pp, 337 mark3c.h, 410         SetOwner           Semaphore_Pend mark3c.cpp, 337 mark3c.h, 410         SetOwner           Semaphore_Post mark3c.cp, 337 mark3c.h, 411         Mark3::Kernel, 145           Semaphore_TimedPend mark3c.cpp, 337 mark3c.h, 411         Mark3::Coroutine, 126           Semaphore_I mark3c.h, 373         SetPriority           Semaphore_I mark3c.h, 373         Mark3::Coroutine, 126           Semaphore_I mark3c.h, 373         Mark3::Coroutine, 126           Semaphore_I mark3c.h, 373         Mark3::Trread.ets, 269           Send         Mark3::Mersad, 257           Mark3::MessageQueue, 192         SetPiorityBase           Send i         Mark3::Thread, 257           Mark3::Mersad, 169         Mark3::Thread, 257           Mark3::Mersad, 169         Mark3::Thread, 258           SetQuantum         SetScheduler           Set Mark3::Mershi, 103         Mark3::Thread, 258           Mark3::AutoAlloc, 103         Mark3::Mershi, 145           SetCode         Mark3::Mershi, 145           SetCode         Mark3::Mershi, 145           SetDebugPrintFunction         Mark3::Mershi, 145           Mark3::Thread, 253         SetThreadContext           Mark3::Thread, 254	•	
mark3c.pp, 336         SetName           mark3c.h, 410         Mark3::Thread, 255           Semaphore Pend         Mark3::Thread, 257           mark3c.pp, 337         Mark3::Thread, 257           mark3c.pp, 337         Mark3::Kernel, 145           mark3c.pp, 337         Mark3::Colist, 119           mark3c.pp, 337         Mark3::Colist, 119           mark3c.ph, 411         Mark3::Coroutine, 126           Semaphore_1         Mark3::Thread, 257           mark3c.h, 411         Mark3::Thread, 257           Semd         Mark3::Thread, 257           Send         Mark3::Thread, 257           Send         Mark3::Mailbox, 168, 169           Mark3::Mailbox, 169         SetConatum           Send i         Mark3::Mailbox, 169           Set Condia         Mark3::Mailbox, 170           Set         SetScheduler           Mark3::Mailbox, 170         SetSaleacGuardThreshold           Mark3::PriorityMapL1, 204         Mark3::Friend, 258           Mark3::PriorityMapL1, 204         Mark3::Friend, 258           Mark3::PriorityMapL2, 208         SetTalloactority           SetCode         Mark3::InkList, 154           Mark3::Mernel, 145         SetTalloactority           SetDabugPrintFunction         Mark3::Markan		•
mark3c.h, 410         Mark3::Thread, 255           Semaphore_Pand         SetOwner           mark3c.cp, 337         Mark3::Thread, 257           mark3c.cp, 337         Mark3::Thread, 257           mark3c.cp, 337         Mark3::Colsit, 119           Semaphore_TimedPend         SetPriorMap           mark3c.cp, 337         Mark3::Coroutine, 126           Semaphore 1         Mark3::Coroutine, 126           Semaphore 1         Mark3::Thread, 257           mark3c.h, 373         SetPriorityBase           Send         Mark3::Mailbox, 168, 169           Mark3::MessageQueue, 192         SetPriorityBase           Send_I         Mark3::Thread, 257           SetScheduler         SetScheduler           SetCode         Mark3::Mailbox, 170         SetStackQuardThreshold           Set         Mark3::Mailbox, 170         SetState Mark3::FriorityMapL1, 204         Mark3::FriorityMapL1, 204           Mark3::PriorityMapL1, 208         Mark3::Message, 185         SetTlail           SetCode         Mark3::Message, 184         Mark3::Thread, 258           SetData         Mark3::Message, 185         Mark3::Mernel, 146           SetEventPlagMask         Mark3::Mernel, 146         SetTimeadContext Mark3::Thread, 254           SetEventPlagMask         Mark3::	• —	
Semaphore_Pend mark3c.cpp, 337 mark3c.b, 410         SetOwner           Semaphore_Post mark3c.cpp, 337 mark3c.b, 411         Mark3::Kernel, 145           Semaphore_TimedPend mark3c.cpp, 337 mark3c.b, 411         SetPriorMap Mark3::Colist, 119           Semaphore_TimedPend mark3c.b, 411         SetPriority Mark3::Crorutine, 126           Semaphore t mark3c.h, 373         Mark3::Crorutine, 126           Send         Mark3::Mailbox, 168, 169 Mark3::MessageQueue, 192           Send_i         Mark3::Mailbox, 169           SendTail         Mark3::Thread, 258           SendIail         Mark3::Scheduler, 225           SendIail         Mark3::Scheduler, 225           SendIail         Mark3::Scheduler, 225           SendIail         Mark3::Scheduler, 225           SetStackGuardThreshold         Mark3::Scheduler, 225           SetStackGuardThreshold         Mark3::EventFlag, 138           Mark3::EventFlag, 138         Mark3::EventFlag, 145           SetSlack GuardThreshold         Mark3::Frread, 258           SetAllocatorFunctions         Mark3::Thread, 258           Mark3::AutoAlloc, 108         SetTaii           SetCode         Mark3::Mernel, 146           SetCode         Mark3::Mernel, 146           SetDeata         Mark3::Mernel, 147           SetEventFlagMote         <	···	
mark3c.pp, 337         Mark3::Thread, 257           Semaphore Post         Mark3::Kernel, 145           mark3c.pp, 337         SelPrioMap           mark3c.pp, 337         Mark3::CoList, 119           Semaphore_TimedPend         SelPriority           mark3c.b, 411         Mark3::CoList, 119           Semaphore_t         Mark3::Coroutine, 126           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Mark3::Mailbox, 169         SelPriorityBase           Send         SelPriorityBase           Mark3::MessageQueue, 192         SetOuantum           Send I         Mark3::Thread, 257           Mark3::Mailbox, 169         SelSackGuardThreshold           Set State         Mark3::Thread, 258           SetSackGuardThreshold         Mark3::Kernel, 145           Set SalackGuardThreshold         Mark3::Kernel, 145           Mark3::PriorityMapL1, 204         Mark3::Mailbox, 108           Mark3::PriorityMapL1, 208         Mark3::Mailbox, 108           SetClode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Mernel, 145         SetThreadContextSwitchCallout           Mark3::Mernel, 145         Mark3::Mernel, 146           SetEventElagMask         Self		
mark3c.h, at 10         SetPanic           Semaphore, Post         Mark3::Kernel, 145           mark3c.h, 411         SetPrioMap           Semaphore, TimedPend         Mark3::CoList, 119           mark3c.h, 411         Mark3::CoList, 119           Semaphore, 1         Mark3::CoList, 119           Semaphore, 2         Mark3::Thread, 257           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Mark3::Mailbox, 169         SetPriorityBase           Mark3::Mailbox, 169         Mark3::Thread, 258           SendTail         Mark3::Thread, 258           Mark3::Mailbox, 170         SetStackGuard Threshold           Set         Mark3::Automic, 103           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL2, 208         SetState           SetAllocator-functions         Mark3::Thread, 258           Mark3::AutoAlloc, 108         SetThat           SetCode         Mark3::Thread, 253           SetData         SetData           Mark3::Mernel, 145         SetEventFlagMode           Mark3::Thread, 253         Mark3::Mernel, 146           SetEventFlagMode         Mark3::Thread, 254           SetEventFlagMode         Mark3::Thread, 256           Mark3::Thread, 254	• —	
Semaphore_Post         Mark3::Kernel, 145           mark3c.cpp, 337         SetPrioMap           mark3c.cpp, 337         Mark3::CoList, 119           mark3c.cpp, 337         Mark3::CoList, 119           mark3c.bn, 411         Mark3::Coroutine, 126           Semaphore_t         Mark3::Thread, 257           mark3c.h, 373         Mark3::Thread, 257           Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Send_i         Mark3::Mailbox, 169           SendTail         Mark3::Thread, 256           Mark3::Mailbox, 170         SetScheduler           Set         SetScheduler, 225           SetState         Mark3::Sernel, 145           Mark3::PriorityMapL1, 204         Mark3::FireintyMapL2, 208           SetAllocatorFunctions         Mark3::InkList, 154           Mark3::AutoAlloc, 108         SetTail           SetCode         Mark3::Thread, 258           SetDebugPrintFunction         Mark3::Mernel, 146           Mark3::Message, 185         SetThreadCortaetCallout           Mark3::Mernel, 145         Mark3::Mernel, 146           SetDebugPrintFunction         Mark3::Mernel, 147           Mark3::Thread, 254         Mark3::Mernel, 147           SetEventFlagMask	···	
mark3c.cpp, 337         SetPrioMap           mark3c.h, 411         Mark3::CoList, 119           Semaphore_TimedPend         SetPriority           mark3c.pp, 337         Mark3::CoList, 119           mark3c.h, 411         Mark3::Cocorutine, 126           Semaphore_t         Mark3::Thread, 257           mark3c.h, 373         Mark3::Thread, 257           Send         Mark3::MessageQueue, 192           Send_i         Mark3::MessageQueue, 192           Send_ai         Mark3::Mark3::Thread, 258           Mark3::Mailbox, 169         SetScheduler           Send_ai         Mark3::Thread, 258           Mark3::Atomic, 103         Mark3::Thread, 258           Mark3::Atomic, 103         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL2, 208         SetState           SetAllocatorFunctions         Mark3::LinkList, 154           Mark3::Message, 184         SetTimeadContextSwitchCallout           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Mernel, 145         SetTimeadExtiCallout           Mark3::Mernel, 146         SetTimeadExtiCallout           Mark3::Thread, 254         Mark3::Mernel, 147           SetExentFlagMode         Mark3::Mernel, 147		
mark3c.h, 411         Mark3::CoList, 119           Semaphore, TimedPend mark3c.pp, 337         Mark3::CoList, 119           mark3c.h, 411         Mark3::Coroutine, 126           Semaphore, Timerk3c.h, 373         Mark3::Thread. 257           Send         Mark3::Thread.st, 269           Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           SendTail         Mark3::Thread, 258           SendTail         Mark3::Scheduler           SendTail         Mark3::Scheduler, 225           SendTail         Mark3::Scheduler, 225           SetStackGuardThreshold         Mark3::Kernel, 145           Set         SetStackGuardThreshold           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           SetCode         Mark3::Thread, 258           SetCode         Mark3::Thread, 258           SetCode         Mark3::TypedCircularLinkList, 290           Mark3::Message, 185         Mark3::Mernel, 146           SetDebugPrintFunction         Mark3::Mernel, 147           Mark3::Thread, 254         Mark3::Mernel, 147           SetEventFlagMask         Signal	• —	
Semaphore_TimedPend         SetPriority           mark3c.cpp, 337         Mark3::CoList, 119           mark3c.h, 411         Mark3::Thread, 257           Semaphore_t         Mark3::Thread, 257           mark3c.h, 373         SetPriorityBase           Send         SetPriorityBase           Mark3::MessageQueue, 192         SetQuantum           Send_i         Mark3::Mread, 258           Mark3::Mailbox, 169         SetStackGuardThreshold           Mark3::Mailbox, 170         SetStackGuardThreshold           Mark3::Atomic, 103         SetStackGuardThreshold           Mark3::Atomic, 103         SetState           Mark3::PriorityMapL1, 204         Mark3::FriorityMapL1, 208           Mark3::PriorityMapL2, 208         SetTail           SetAllocatorFunctions         Mark3::Message, 184           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Message, 185         SetThreadContextSwitchCallout           Mark3::Message, 185         SetThreadCreateCallout           SetEventFlagMose         Mark3::Kernel, 146           SetEventFlagMask         Signal           Mark3::Thread, 254         SetEmapter           SetExpired         Mark3::ConditionVariable, 122           Mark3::Thread, 255         SetExtendedContext </td <td>···</td> <td>•</td>	···	•
mark3c.cpp, 337         Mark3::Coroutine, 126           semaphore_1         Mark3::Thread, 257           mark3c.h, 373         Mark3::Thread, 257           Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Mark3::Mailbox, 169         SetQuantum           SendTail         Mark3::Mailbox, 170         SetScheduler           Set         Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::EventFlaq, 138         Mark3::Kernel, 145           Mark3::PriorityMapL1, 204         Mark3::Fhread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::AutoAlloc, 108         SetTail           SetCode         Mark3::Thread, 258           SetCode         SetTrneadContext5witchCallout           Mark3::Message, 184         Mark3::Mernel, 146           SetOeturent         SetThreadContext5witchCallout           Mark3::Message, 185         SetThreadContext5witchCallout           Mark3::Mernel, 145         SetTimeafExitCallout           SetEventFlagMask         Signal           Mark3::Thread, 254         SetEventFlagMask           Mark3::Thread, 254         Seep           SetExpired         Mark3::Thread, 255           SetH		
mark3c.h, 411         Mark3::Coroutine, 126           Semaphore_t         Mark3::Thread, 257           mark3c.h, 373         Mark3::Thread, 257           Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 258           Mark3::Mailbox, 169         SetQuantum           SendTail         Mark3::Scheduler           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           Set         Mark3::FriorityMapl_1, 204           Mark3::PriorityMapl_2, 208         Mark3::Thread, 258           SetAllocatorFunctions         Mark3::Thread, 258           SetCode         SetTail           Mark3::Message, 184         Mark3::TypedDoubleLinkList, 290           Mark3::Message, 184         Mark3::Kernel, 146           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 185         SetThreadCreateCallout           Mark3::Message, 185         SetThreadExitCallout           Mark3::Mernel, 145         Mark3::Gernel, 146           SetEventFlagMosk         Mark3::Gernel, 147           SetEvertEngdMask         Mark3::Guantum, 215           SetExtendedContext         Mark3::Thread, 254           SetExtendedContext         Mark3::Thread, 255	• —	SetPriority
Semaphore_1         Mark3::Thread, 257           mark3c.h, 373         SetPriorityBase           SetPriorityBase         SetPriorityBase           Mark3::MessageQueue, 192         SetQuantum           Send_i         Mark3::Mread, 258           Mark3::Mailbox, 169         SetScheduler           SendTail         Mark3::Scheduler, 225           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::EventFlag, 138           Mark3::PriorityMapL1, 204         SetState           Mark3::PriorityMapL1, 204         SetTail           Mark3::PriorityMapL2, 208         Mark3::Thread, 258           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Mernel, 146         SetThreadExitCallout           Mark3::Mernel, 145         SetThreadExitCallout           Mark3::Mernel, 145         SetTimeafLiread           SetDebugPrintFunction         Mark3::Grnel, 147           Mark3::Thread, 254         SetTimerThread           SetEventFlagMode         Mark3::Thread, 254           SetExpired         Mark3::Thread, 258           SetExpired         Mark3::Thread, 25	···	
mark3c.h, 373         Mark3::ThreadList, 269           Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Send_j         Mark3::Thread, 258           SendTail         Mark3::Scheduler           Mark3::Mailbox, 170         SetStackduardThreshold           Set         Mark3::Schenduler, 225           Mark3::PriorityMapL1, 204         Mark3::FriorityMapL2, 208           Mark3::PriorityMapL2, 208         Mark3::Thread, 258           SetCode         Mark3::TypedCircularLinkList, 290           Mark3::Message, 184         Mark3::TypedDoubleLinkList, 293           SetCorent         SetThreadContextSwitchCallout           Mark3::Message, 185         SetThreadCreateCallout           Mark3::Mernel, 146         SetThreadCreateCallout           Mark3::Mernel, 145         SetTireadCreateCallout           Mark3::Mernel, 145         Mark3::Kernel, 147           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetExpired         Mark3::Thread, 254           SetExpired         Mark3::Thread, 254           SetExpired         Mark3::Thread, 255           SetExpired         Mark3::Thread, 255           SetExpired         Mark3::Thread, 2		
Send         SetPriorityBase           Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Mark3::MessageQueue, 192         SetQuantum           Send_i         Mark3::Thread, 258           SendTail         Mark3::Scheduler           SendTail         Mark3::Scheduler           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           SetState         Mark3::Kernel, 145           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           SetAllocatorFunctions         Mark3::InkList, 154           Mark3::PriorityMapL2, 208         Mark3::InkList, 154           Mark3::AutoAlloc, 108         Mark3::InkList, 154           SetAllocatorFunctions         Mark3::Horeal, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadCreateCallout           SetDebugPrintFunction         Mark3::Mernel, 146           Mark3::Message, 185         SetThreadExitCallout           Mark3::Mernel, 147         Mark3::Mernel, 147           SetEventFlagMode         Mark3::Mernel, 147           Mark3::Thread, 254         Set	• —	•
Mark3::Mailbox, 168, 169         Mark3::Thread, 257           Mark3::Mailbox, 169         SetQuantum           Send_i         Mark3::Thread, 258           SendTail         Mark3::Scheduler           SetScheduler         Mark3::Scheduler, 225           Set Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           Set Mark3::PriorityMapL1, 204         Mark3::EventFlag, 138           Mark3::PriorityMapL2, 208         Mark3::Thread, 258           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedCoircularLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         Mark3::Mernel, 146           SetOata         SetThreadCreateCallout           Mark3::Message, 185         SetThreadCreateCallout           Mark3::Message, 185         Mark3::Kernel, 146           SetDebugPrintFunction         SetTimeration           Mark3::Mernel, 145         Mark3::Countium, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::Countium, 215           SetExpired         Mark3::ConditionVariable, 122           Mark3::Thread, 254         Start           SetEeded	mark3c.h, 373	
Mark3::MessageQueue, 192         SetQuantum           Send i         Mark3::Mailbox, 169           SendTail         Mark3::Scheduler           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           SetState         Mark3::Kernel, 145           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::TypedCircularLinkList, 154           Mark3::PriorityMapL2, 208         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedCircularLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         Mark3::TypedDoubleLinkList, 293           SetOurent         SetThreadCreateCallout           Mark3::Hernel, 146         SetThreadCreateCallout           Mark3::Message, 185         Mark3::Kernel, 146           SetDebugPrintFunction         SetTimeradCreateCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetEventFlagMosk         Mark3::Guantum, 215           SetEventFlagMode         Mark3::ConditionVariable, 122           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::Thread, 258           Mark3::Thread, 255         Mark3::Mernel, 147	Send	-
Send_i         Mark3::Thread, 258           Mark3::Mailbox, 169         SetScheduler           SendTail         Mark3::Scheduler, 225           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         SetState           Mark3::PriorityMapL2, 208         Mark3::Thread, 258           SetAllocatorFunctions         Mark3::LinkList, 154           Mark3::AutoAlloc, 108         Mark3::TypedCircularLinkList, 290           Mark3::Message, 184         Mark3::ThreadContextSwitchCallout           Mark3::Message, 184         SetThreadCreateCallout           Mark3::Mernel, 146         SetThreadCreateCallout           Mark3::Hernel, 146         SetThreadSxitCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Guantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::Marka           SetEventFlagMask         Signal           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start     <	Mark3::Mailbox, 168, 169	Mark3::Thread, 257
Mark3::Mailbox, 169         SetScheduler           SendTail         Mark3::Scheduler, 225           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         Mark3::Thread, 258           Mark3::PriorityMapL2, 208         Mark3::InkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Thread, 253         SetThreadCreateCallout           Mark3::Message, 185         SetThreadEreateCallout           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 146         SetTimerThread           Mark3::Kernel, 147         SetEventFlagMask           Mark3::Thread, 254         Signal           SetEventFlagMode         Mark3::ConditionVariable, 122           Mark3::Thread, 254         Sleep           SetExtendedContext         Mark3::FrorileTimer, 151           Mark3::Thread, 255         Mark3::FrorileTimer, 212           Mark3::Thread, 255         Mark3::FrorileTimer, 277 </td <td>Mark3::MessageQueue, 192</td> <td>SetQuantum</td>	Mark3::MessageQueue, 192	SetQuantum
SendTail         Mark3::Scheduler, 225           Mark3::Mailbox, 170         SetStackGuardThreshold           Set         Mark3::Kernel, 145           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         SetTail           Mark3::PriorityMapL2, 208         Mark3::LinkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadCreateCallout           SetOuter         SetThreadCreateCallout           Mark3::Message, 185         SetThreadExitCallout           SetData         SetThreadExitCallout           Mark3::Mernel, 145         SetTimerThread           SetEventFlagMask         Signal           Mark3::GronditionVariable, 122         Mark3::ConditionVariable, 122           MetExpired         Mark3::ConditionVariable, 122           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::KernelSWI, 150           Mark3::MernelS	Send_i	Mark3::Thread, 258
Mark3::Mailbox, 170         SetStackGuardThreshold Mark3::Kernel, 145           Mark3::EventFlag, 138         Mark3::Thread, 258           Mark3::PriorityMapL1, 204         SetTail           Mark3::PriorityMapL2, 208         Mark3::InkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout Mark3::Kernel, 146           SetCurrent         SetThreadCreateCallout Mark3::Kernel, 146           Mark3::Message, 185         SetThreadCreateCallout Mark3::Kernel, 147           SetData         SetThreadCreateCallout Mark3::Kernel, 147           Mark3::Message, 185         SetTimerThread           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Quantum, 215           SetEventFlagMode         Mark3::Quantum, 215           Mark3::Thread, 254         Signal           Mark3::Thread, 254         Start           SetExpired         Mark3::Notify, 200           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelTimer, 151           Mark3::TypedDoubleLinkList, 289         Mark3::Thread, 259           Mark3::Th	Mark3::Mailbox, 169	SetScheduler
Set         Mark3::Atomic, 103         SetState           Mark3::PriorityMapL1, 204         SetTail           Mark3::PriorityMapL2, 208         Mark3::InkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::Message, 184         Mark3::Message, 184           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadCreateCallout           SetDebug Mark3::Message, 185         SetThreadCreateCallout           Mark3::Message, 185         SetThreadExitCallout           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         SetTimerThread           SetEventFlagMask         Signal           Mark3::Thread, 254         SetEventFlagMode           Mark3::Thread, 254         SetExpired           Mark3::Thread, 254         Start           SetExpired         Mark3::Mernel, 147           Mark3::Thread, 254         Start           SetExendedContext         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelFilmer, 151           Mark3::Thread, 255         Mark3::FrofileTimer, 212           Mark3::Thread, 259         Mark3::Timer, 277           SetID         Mark3::ThreadPort, 273           SetID         Mar	SendTail	Mark3::Scheduler, 225
Mark3::Atomic, 103         SetState           Mark3::PriorityMapL1, 204         SetTail           Mark3::PriorityMapL1, 208         Mark3::LinkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         SetThreadCreateCallout           SetData         SetThreadCreateCallout           Mark3::Thread, 253         SetThreadExitCallout           SetData         SetThreadExitCallout           Mark3::Message, 185         SetTimerThread           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Guantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetExpired         Mark3::OnditionVariable, 122           Mark3::Thread, 254         Sleep           SetExtendedContext         Mark3::Thread, 258           Mark3::Thread, 255         Mark3::KernelTimer, 151           Mark3::TypedCircularLinkList, 289         Mark3::Friead, 259           Mark3::Thread, 255         Mark3::Thread, 259           Mark3::Thread, 255         Mark3::Thread           Set	Mark3::Mailbox, 170	SetStackGuardThreshold
Mark3::EventFlag, 138         Mark3::PriorityMapL1, 204           Mark3::PriorityMapL2, 208         Mark3::LinkList, 154           SetAllocatorFunctions         Mark3::TypedCircularLinkList, 290           Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         Mark3::Kernel, 146           SetCurrent         SetThreadCreateCallout           Mark3::Mersel, 253         Mark3::Kernel, 146           SetData         SetThreadExitCallout           Mark3::Message, 185         SetThreadExitCallout           Mark3::Message, 185         SetThreadExitCallout           Mark3::Message, 185         SetTimerThread           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 147         Mark3::Guantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           MetExpired         Mark3::Notify, 200           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 255         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelTimer, 151           Mark3::TypedCircularLinkList, 289         Mark3::Thread, 259	Set	Mark3::Kernel, 145
Mark3::PriorityMapL1, 204 Mark3::PriorityMapL2, 208  SetAllocatorFunctions Mark3::AutoAlloc, 108 SetCode Mark3::Message, 184 SetCurrent Mark3::Thread, 253 SetData Mark3::Kernel, 146 SetData Mark3::Kernel, 145 SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetBata Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHoad Mark3::Thread, 255 SetID Mark3::Thread, 255 SetID Mark3::Thread, 255 SetInitimer Mark3::Thread, 255 SetInitimer Mark3::ConditionVariable, 122 Mark3::Rernel, 147 Mark3::Rernel, 147 Mark3::Rernel, 147 Mark3::RernelSWI, 150 Mark3::Thread, 255 SetInitimer Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 255 SetInitimer Mark3::ConditionVariable, 122 Mark3::RernelSWI, 150 Mark3::RernelSWI, 150 Mark3::Thread, 255 SetInitimer Mark3::Thread, 259 Mark3::Thread, 250 Mark	Mark3::Atomic, 103	SetState
Mark3::PriorityMapL2, 208  SetAllocatorFunctions Mark3::AutoAlloc, 108  SetCode Mark3::Message, 184  SetCurrent Mark3::Thread, 253  SetData Mark3::Message, 185  SetDebugPrintFunction Mark3::Mernel, 146  SetDebugPrintFunction Mark3::Kernel, 147  SetDebugPrintFunction Mark3::Thread, 254  SetEventFlagMask Mark3::Thread, 254  SetExpired Mark3::Thread, 254  SetExpired Mark3::Thread, 254  SetExtendedContext Mark3::Thread, 255  SetExtendedContext Mark3::Thread, 255  SetHad Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293  SetID  Mark3::Thread, 255  SetInTimer Mark3::Guantum, 215  SetIntialized Mark3::BlockingObject, 112  Mark3::BrofileTimer, 152 Mark3::ProfileTimer, 152 Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212	Mark3::EventFlag, 138	Mark3::Thread, 258
SetAllocatorFunctions Mark3::AutoAlloc, 108 SetCode SetThreadContextSwitchCallout Mark3::Message, 184 SetCurrent Mark3::Thread, 253 SetData SetDebugPrintFunction Mark3::Message, 185 SetDebugPrintFunction Mark3::Kernel, 146 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHad  Mark3::Thread, 255  SetHad  Mark3::Thread, 255  Mark3::Thread, 259  Mark3::Thread, 259  Mark3::Thread, 255 SetInTimer  Mark3::Guantum, 215 SetInTimer  Mark3::Guantum, 215  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212  Mark3::ProfileTimer, 212	Mark3::PriorityMapL1, 204	SetTail
Mark3::AutoAlloc, 108 SetCode Mark3::Message, 184 SetCurrent Mark3::TypedDoubleLinkList, 293 SetThreadContextSwitchCallout Mark3::Message, 184 SetData Mark3::Message, 185 SetData Mark3::Kernel, 146 SetThreadExitCallout Mark3::Message, 185 SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInitialized Mark3::Quantum, 215 Mark3::ConditionVariable, 122 Mark3::Rernel, 147 Mark3::Thread, 258 Start  Mark3::Rernel, 147 Mark3::RernelSWI, 150 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInitialized Mark3::ConditionVariable, 122 Mark3::Thread, 258 Mark3::Thread, 258 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 255 Mark3::ThreadPort, 273 SetID Mark3::ConditionVariable, 122 Mark3::ThreadPort, 273 SetInitialized Mark3::ConditionVariable, 122 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	Mark3::PriorityMapL2, 208	Mark3::LinkList, 154
Mark3::AutoAlloc, 108         Mark3::TypedDoubleLinkList, 293           SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         Mark3::Kernel, 146           SetCurrent         SetThreadCreateCallout           Mark3::Thread, 253         Mark3::Kernel, 146           SetData         SetThreadExitCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Quantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetEventFlagMode         Mark3::Notify, 200           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelSWI, 150           SetHead         Mark3::KernelTimer, 151           Mark3::TypedCircularLinkList, 289         Mark3::FrofileTimer, 212           Mark3::TypedDoubleLinkList, 293         StartThread, 259           Mark3::Thread, 255         Mark3::ThreadPort, 273           SetID         Mark3::ThreadPort, 273           SetInTimer         Mar		Mark3::TypedCircularLinkList, 290
SetCode         SetThreadContextSwitchCallout           Mark3::Message, 184         Mark3::Kernel, 146           SetCurrent         SetThreadCreateCallout           Mark3::Thread, 253         Mark3::Kernel, 146           SetData         SetThreadExitCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Quantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetEventFlagMode         Mark3::Notify, 200           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start           SetExpired         Mark3::Kernel, 147           Mark3::Thread, 254         Start           SetExpired         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelSWI, 150           SetHead         Mark3::KernelSWI, 150           Mark3::TypedCircularLinkList, 289         Mark3::FrofileTimer, 212           Mark3::TypedCircularLinkList, 289         Mark3::Thread, 259           Mark3::Thread, 255         StartThread           SetInTimer         StartThread	Mark3::AutoAlloc, 108	<u>.</u> .
SetCurrent         SetThreadCreateCallout           Mark3::Thread, 253         Mark3::Kernel, 146           SetData         SetThreadExitCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Quantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetEventFlagMode         Mark3::Notify, 200           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelSWI, 150           SetHead         Mark3::KernelTimer, 151           Mark3::Thread, 255         Mark3::ProfileTimer, 212           Mark3::Thread, 259         Mark3::Thread, 259           Mark3::Thread, 259         Mark3::ThreadPort, 273           SetInTimer         Stop           Mark3::Quantum, 215         Mark3, 98           SetInitialized         Mark3::ProfileTimer, 212           Mark3::BlockingObject, 112         Mark3::ProfileTimer, 212		<u>.</u> .
SetCurrent         SetThreadCreateCallout           Mark3::Thread, 253         Mark3::Kernel, 146           SetData         SetThreadExitCallout           Mark3::Message, 185         Mark3::Kernel, 147           SetDebugPrintFunction         SetTimerThread           Mark3::Kernel, 145         Mark3::Quantum, 215           SetEventFlagMask         Signal           Mark3::Thread, 254         Mark3::ConditionVariable, 122           SetEventFlagMode         Mark3::Notify, 200           Mark3::Thread, 254         Sleep           SetExpired         Mark3::Thread, 258           Mark3::Thread, 254         Start           SetExtendedContext         Mark3::Kernel, 147           Mark3::Thread, 255         Mark3::KernelSWI, 150           SetHead         Mark3::KernelTimer, 151           Mark3::Thread, 255         Mark3::ProfileTimer, 212           Mark3::Thread, 259         Mark3::Thread, 259           Mark3::Thread, 259         Mark3::ThreadPort, 273           SetInTimer         Stop           Mark3::Quantum, 215         Mark3, 98           SetInitialized         Mark3::ProfileTimer, 212           Mark3::BlockingObject, 112         Mark3::ProfileTimer, 212	Mark3::Message, 184	Mark3::Kernel, 146
SetData Mark3::Message, 185 SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 255 SetInTimer Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 Mark3::Quantum, 215 Mark3::Quantum, 215 Mark3::RernelTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212		
SetData Mark3::Message, 185 SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Thread, 255 SetInTimer Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 Mark3::Quantum, 215 Mark3::Quantum, 215 Mark3::RernelTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	Mark3::Thread, 253	Mark3::Kernel, 146
SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 SetInTimer Mark3::Quantum, 215 SetIntialized Mark3::BlockingObject, 112 Mark3::BlockingObject, 112 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	SetData	SetThreadExitCallout
SetDebugPrintFunction Mark3::Kernel, 145 SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 SetInTimer Mark3::Quantum, 215 SetIntialized Mark3::BlockingObject, 112 Mark3::BlockingObject, 112 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	Mark3::Message, 185	Mark3::Kernel, 147
Mark3::Kernel, 145  SetEventFlagMask     Mark3::Thread, 254  SetEventFlagMode     Mark3::Thread, 254  SetExpired     Mark3::Thread, 254  SetExpired     Mark3::Thread, 258  Mark3::Thread, 254  SetExtendedContext     Mark3::Thread, 255  SetHead     Mark3::LinkList, 154     Mark3::TypedCircularLinkList, 289     Mark3::TypedDoubleLinkList, 293  SetID     Mark3::Thread, 255  SetInTimer     Mark3::Quantum, 215  SetInTimer     Mark3::Quantum, 215  SetInTimer     Mark3::BlockingObject, 112  Mark3::BlockingObject, 112  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212  Mark3::ProfileTimer, 152  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212	<del>_</del>	
SetEventFlagMask Mark3::Thread, 254 SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::Thread, 255 SetHead Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetInTimer Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 SetIntitalized Mark3::BlockingObject, 112  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212 Mark3::ProfileTimer, 151 Mark3::RernelTimer, 277 StartThreads Mark3::Thread, 255 Mark3::ThreadFort, 273  SetInTimer Mark3::Guantum, 215 Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212	<u> </u>	Mark3::Quantum, 215
Mark3::Thread, 254  SetEventFlagMode Mark3::Thread, 254  SetExpired Mark3::Thread, 258  Mark3::Thread, 254  SetExtendedContext Mark3::Thread, 255  SetHead Mark3::Thread, 255  SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293  SetID Mark3::Thread, 255  SetInTimer Mark3::Quantum, 215  SetInitialized Mark3::BlockingObject, 112  Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212		
SetEventFlagMode Mark3::Thread, 254 SetExpired Mark3::Thread, 254 SetExtendedContext Mark3::Thread, 255 SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 SetID Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 SetInitialized Mark3::BlockingObject, 112 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	<del>-</del>	
Mark3::Thread, 254  SetExpired Mark3::Thread, 254  SetExtendedContext Mark3::Thread, 255  SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293  SetID Mark3::Thread, 255  SetInTimer Mark3::Quantum, 215  SetInitialized Mark3::BlockingObject, 112  Start  Mark3::Thread, 258  Mark3::Thread, 259 Mark3::ThreadPort, 277  Stop Mark3::Quantum, 215  Mark3::BlockingObject, 112  StartThreads Mark3::ProfileTimer, 152 Mark3::ProfileTimer, 212	,	•
SetExpired Mark3::Thread, 254  SetExtendedContext Mark3::Kernel, 147  Mark3::Thread, 255  SetHead Mark3::LinkList, 154  Mark3::TypedCircularLinkList, 289  Mark3::TypedDoubleLinkList, 293  SetID StartThread, 255  Mark3::Thread, 255  SetInTimer Stop  Mark3::Quantum, 215  SetInitialized Mark3::BlockingObject, 112  Mark3::ProfileTimer, 258  Mark3::Thread, 259  Mark3::ThreadPort, 273  SetOp  Mark3::Guantum, 215  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212	_	
Mark3::Thread, 254  SetExtendedContext Mark3::Thread, 255  SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293  SetID Mark3::Thread, 255  SetInTimer Mark3::Quantum, 215  SetInitialized Mark3::BlockingObject, 112  Start  Mark3::Kernel, 147 Mark3::KernelSWI, 150 Mark3::KernelTimer, 151 Mark3::FrofileTimer, 212  Mark3::Thread, 259 Mark3::Thread, 259 Mark3::Threads Mark3::Threads Mark3::Threads Mark3::ThreadFort, 273  SetInTimer Mark3::Quantum, 215  Mark3::RernelTimer, 152 Mark3::ProfileTimer, 212	,	•
SetExtendedContext Mark3::Thread, 255 Mark3::Kernel, 147 Mark3::Thread, 255 Mark3::KernelSWI, 150 SetHead Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 Mark3::TypedDoubleLinkList, 293 SetID StartThreads Mark3::Thread, 255 Mark3::Thread, 255 SetInTimer Mark3::Quantum, 215 SetIntialized Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212	•	
Mark3::Thread, 255  SetHead  Mark3::LinkList, 154  Mark3::TypedCircularLinkList, 289  Mark3::TypedDoubleLinkList, 293  SetID  Mark3::Thread, 255  Mark3::Thread, 255  SetInTimer  Mark3::Quantum, 215  SetInitialized  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 150  Mark3::KernelSWI, 150  Mark3::FrofileTimer, 212  Mark3::KernelTimer, 152  Mark3::ProfileTimer, 212		
SetHead Mark3::KernelTimer, 151 Mark3::LinkList, 154 Mark3::ProfileTimer, 212 Mark3::TypedCircularLinkList, 289 Mark3::Thread, 259 Mark3::TypedDoubleLinkList, 293 Mark3::Timer, 277 SetID StartThreads Mark3::Thread, 255 Mark3::ThreadPort, 273 SetInTimer Stop Mark3::Quantum, 215 Mark3, 98 SetInitialized Mark3::KernelTimer, 152 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212		
Mark3::LinkList, 154 Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293 Mark3::TypedDoubleLinkList, 293 Mark3::Timer, 277 SetID StartThreads Mark3::Thread, 255 Mark3::ThreadPort, 273 SetInTimer Mark3::Quantum, 215 SetInitialized Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212		•
Mark3::TypedCircularLinkList, 289 Mark3::TypedDoubleLinkList, 293  SetID StartThreads Mark3::Thread, 255 Mark3::ThreadPort, 273  SetInTimer Mark3::Quantum, 215  SetInitialized Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212		
Mark3::TypedDoubleLinkList, 293  SetID  StartThreads  Mark3::Thread, 255  SetInTimer  Mark3::Quantum, 215  SetInitialized  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212		
SetID StartThreads Mark3::Thread, 255 Mark3::ThreadPort, 273 SetInTimer Stop Mark3::Quantum, 215 Mark3, 98 SetInitialized Mark3::KernelTimer, 152 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212	- ·	
Mark3::Thread, 255  SetInTimer  Mark3::Quantum, 215  SetInitialized  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212	• •	
SetInTimer Stop  Mark3::Quantum, 215 Mark3, 98  SetInitialized Mark3::KernelTimer, 152  Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212		
Mark3::Quantum, 215  SetInitialized  Mark3::BlockingObject, 112  Mark3::ProfileTimer, 212		
SetInitialized Mark3::KernelTimer, 152 Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212		•
Mark3::BlockingObject, 112 Mark3::ProfileTimer, 212		
wark3::Timer, 277 Mark3::Thread, 259	- ·	
	iviark3::Timer, 2//	Mark3::1 nread, 259

M 10 T' 070	TI 1 0 101 1
Mark3::Timer, 278	Thread_GetState
StringLength	mark3c.cpp, 341
Mark3::MemUtil, 180	mark3c.h, 414
StringSearch	Thread_Init
Mark3::MemUtil, 180	mark3c.cpp, 341
StringToDecimal16	mark3c.h, 414
Mark3::MemUtil, 180	Thread RestoreContext
StringToDecimal32	threadport.h, 449
Mark3::MemUtil, 181	Thread SaveContext
StringToDecimal64	threadport.h, 449
Mark3::MemUtil, 181	•
,	Thread_SetID
StringToDecimal8	mark3c.cpp, 342
Mark3::MemUtil, 181	mark3c.h, 415
Sub	Thread_SetName
Mark3::Atomic, 104	mark3c.cpp, 342
TOOD IN IT	Thread_SetPriority
TCCR1B_INIT	mark3c.cpp, 342
kerneltimer.cpp, 440	mark3c.h, 415
THREAD_QUANTUM_DEFAULT	Thread SetQuantum
portcfg.h, 447	mark3c.cpp, 343
THREAD_SIZE	Thread Sleep
mark3c.h, <mark>371</mark>	mark3c.cpp, 343
TIMER_IFR	• •
kerneltimer.cpp, 440	mark3c.h, 416
TIMER_IMSK	Thread_Start
kerneltimer.cpp, 440	mark3c.cpp, 344
TIMER_SIZE	mark3c.h, 416
 mark3c.h, 371	Thread_Stop
TOKEN 1	mark3c.cpp, 344
mark3c.h, 371	mark3c.h, 416
TOKEN 2	Thread_Switch
mark3c.h, 371	Mark3, 100
TestAndSet	Thread Yield
	mark3c.cpp, 344
Mark3::Atomic, 104	mark3c.h, 417
Thread	thread_entry_func_t
Mark3, 98	mark3c.h, 374
Mark3::Thread, 245	thread_state_t
Mark3::ThreadPort, 273	mark3c.h, 374
Thread_CoopYield	
mark3c.cpp, 338	Thread_t
mark3c.h, 412	mark3c.h, 374
Thread_Exit	ThreadContextCallout
mark3c.cpp, 338	Mark3, 96
mark3c.h, 412	ThreadCreateCallout
Thread_GetCurPriority	Mark3, 96
mark3c.cpp, 339	ThreadEntryFunc
mark3c.h, 412	Mark3, 97
Thread GetID	ThreadExitCallout
mark3c.cpp, 339	Mark3, 97
mark3c.h, 413	ThreadList
Thread GetName	Mark3::ThreadList, 265
mark3c.cpp, 340	ThreadPort
Thread_GetPriority	Mark3::Thread, 260
_ ·	ThreadState
mark3c.cpp, 340	
mark3c.h, 413	Mark3, 98
Thread_GetQuantum	threadport.cpp
mark3c.cpp, 340	g_kwCriticalCount, 453
Thread_GetStackSlack	g_kwSFR, 453
mark3c.cpp, 340	threadport.h

ASM, 449	Update
PORT_PUSH_TO_STACK, 449	Mark3::Quantum, 215
PORT_TOP_OF_STACK, 449	User
Thread_RestoreContext, 449	Mark3, 98
Thread_SaveContext, 449	
Tick	WORD_ROUND
Mark3::Kernel, 147	mark3c.h, 371
Timer	Wait
Mark3, 98	Mark3::ConditionVariable, 122
Mark3::Timer, 275	Mark3::EventFlag, 138, 139
Timer Init	Mark3::Notify, 200, 201
mark3c.cpp, 345	Wait_i
mark3c.h, 417	Mark3::EventFlag, 139
Timer Restart	WakeMe
mark3c.cpp, 345	Mark3::EventFlag, 140
mark3c.h, 418	Mark3::Mutex, 197
Timer Start	Mark3::Notify, 201
mark3c.cpp, 345	Mark3::Semaphore, 233
mark3c.h, 418	WakeNext
Timer_Stop	Mark3::Mutex, 197
mark3c.cpp, 346	Mark3::Semaphore, 233
mark3c.h, 419	Write
timer callback t	Mark3::Streamer, 239, 240
mark3c.h, 374	, ,
Timer t	Yield
mark3c.h, 374	Mark3::Thread, 259
TimerCallback	
Mark3, 97	
TimerList	
Mark3::Timer, 278	
Tokenize	
Mark3::MemUtil, 182	
Trigger	
Mark3::KernelSWI, 150 TypedCircularLinkList	
Mark3::TypedCircularLinkList, 287	
•	
TypedDoubleLinkList  Mark3: TypedDoubleLinkList 201	
Mark3::TypedDoubleLinkList, 291	
u8Len	
Mark3::Token_t, 286	
uMaxTimerTicks	
Mark3, 101	
USecondsToTicks	
Mark3::Timer, 278	
uTimerFlagActive	
Mark3, 101	
uTimerFlagCallback	
Mark3, 101	
uTimerFlagExpired	
Mark3, 101	
uTimerFlagOneShot	
Mark3, 102	
uTimerTicksInvalid	
Mark3, 102	
UnBlock	
Mark3::BlockingObject, 112	
Unlock	
Mark3::Streamer, 239	
ividi NoOli Gailleli, 203	