

ChibiOS/RT 2.4.0

ARM/GCC Kernel Reference Manual

Contents

1 ChibiOS/RT	1
1.1 Copyright	1
1.2 Introduction	1
1.3 Related Documents	1
2 Kernel Concepts	2
2.1 Naming Conventions	2
2.2 API Name Suffixes	2
2.3 Interrupt Classes	3
2.4 System States	3
2.5 Scheduling	5
2.6 Thread States	6
2.7 Priority Levels	6
2.8 Thread Working Area	6
3 Module Index	8
3.1 Modules	8
4 Class Hierarchy Index	10
4.1 Class Hierarchy	10
5 Data Structure Index	11
5.1 Data Structures	11
6 File Index	12
6.1 File List	12
7 Module Documentation	14
7.1 Kernel	14
7.1.1 Detailed Description	14
7.2 Version Numbers and Identification	14
7.2.1 Detailed Description	14
7.2.2 Function Documentation	15
7.2.2.1 _idle_thread	15

7.2.3	Define Documentation	15
7.2.3.1	_CHIBIOS_RT_	15
7.2.3.2	CH_KERNEL_VERSION	15
7.2.3.3	CH_KERNEL_MAJOR	15
7.2.3.4	CH_KERNEL_MINOR	15
7.2.3.5	CH_KERNEL_PATCH	15
7.3	Configuration	15
7.3.1	Detailed Description	15
7.3.2	Define Documentation	17
7.3.2.1	CH_FREQUENCY	17
7.3.2.2	CH_TIME_QUANTUM	18
7.3.2.3	CH_MEMCORE_SIZE	18
7.3.2.4	CH_NO_IDLE_THREAD	18
7.3.2.5	CH_OPTIMIZE_SPEED	18
7.3.2.6	CH_USE_REGISTRY	18
7.3.2.7	CH_USE_WAITEXIT	19
7.3.2.8	CH_USE_SEMAPHORES	19
7.3.2.9	CH_USE_SEMAPHORES_PRIORITY	19
7.3.2.10	CH_USE_SEMSW	19
7.3.2.11	CH_USE_MUTEXES	19
7.3.2.12	CH_USE_CONDVARS	20
7.3.2.13	CH_USE_CONDVARS_TIMEOUT	20
7.3.2.14	CH_USE_EVENTS	20
7.3.2.15	CH_USE_EVENTS_TIMEOUT	20
7.3.2.16	CH_USE_MESSAGES	20
7.3.2.17	CH_USE_MESSAGES_PRIORITY	21
7.3.2.18	CH_USE_MAILBOXES	21
7.3.2.19	CH_USE_QUEUES	21
7.3.2.20	CH_USE_MEMCORE	21
7.3.2.21	CH_USE_HEAP	21
7.3.2.22	CH_USE_MALLOC_HEAP	22
7.3.2.23	CH_USE_MEMPOOLS	22
7.3.2.24	CH_USE_DYNAMIC	22
7.3.2.25	CH_DBG_SYSTEM_STATE_CHECK	22
7.3.2.26	CH_DBG_ENABLE_CHECKS	22
7.3.2.27	CH_DBG_ENABLE_ASSERTS	23
7.3.2.28	CH_DBG_ENABLE_TRACE	23
7.3.2.29	CH_DBG_ENABLE_STACK_CHECK	23
7.3.2.30	CH_DBG_FILL_THREADS	23
7.3.2.31	CH_DBG_THREADS_PROFILING	23

7.3.2.32	THREAD_EXT_FIELDS	24
7.3.2.33	THREAD_EXT_INIT_HOOK	24
7.3.2.34	THREAD_EXT_EXIT_HOOK	24
7.3.2.35	THREAD_CONTEXT_SWITCH_HOOK	24
7.3.2.36	IDLE_LOOP_HOOK	24
7.3.2.37	SYSTEM_TICK_EVENT_HOOK	25
7.3.2.38	SYSTEM_HALT_HOOK	25
7.4	Types	25
7.5	Base Kernel Services	25
7.5.1	Detailed Description	25
7.6	System Management	25
7.6.1	Detailed Description	25
7.6.2	Function Documentation	27
7.6.2.1	chSysInit	27
7.6.2.2	chSysTimerHandlerI	27
7.6.2.3	_idle_thread	28
7.6.3	Define Documentation	28
7.6.3.1	chSysGetIdleThread	28
7.6.3.2	chSysHalt	29
7.6.3.3	chSysSwitch	29
7.6.3.4	chSysDisable	29
7.6.3.5	chSysSuspend	30
7.6.3.6	chSysEnable	30
7.6.3.7	chSysLock	30
7.6.3.8	chSysUnlock	31
7.6.3.9	chSysLockFromIlsr	31
7.6.3.10	chSysUnlockFromIlsr	31
7.6.3.11	CH_IRQ_PROLOGUE	32
7.6.3.12	CH_IRQ_EPILOGUE	32
7.6.3.13	CH_IRQ_HANDLER	32
7.6.3.14	CH_FAST_IRQ_HANDLER	32
7.7	Scheduler	33
7.7.1	Detailed Description	33
7.7.2	Function Documentation	34
7.7.2.1	_scheduler_init	34
7.7.2.2	chSchReadyI	35
7.7.2.3	chSchGoSleepS	35
7.7.2.4	chSchGoSleepTimeoutS	36
7.7.2.5	chSchWakeUpS	36
7.7.2.6	chSchRescheduleS	37

7.7.2.7	chSchIsPreemptionRequired	38
7.7.2.8	chSchDoReschedule	38
7.7.3	Variable Documentation	38
7.7.3.1	rlist	38
7.7.4	Define Documentation	38
7.7.4.1	RDY_OK	38
7.7.4.2	RDY_TIMEOUT	39
7.7.4.3	RDY_RESET	39
7.7.4.4	NOPRIO	39
7.7.4.5	IDLEPRIO	39
7.7.4.6	LOWPRIO	39
7.7.4.7	NORMALPRIO	39
7.7.4.8	HIGHPRIO	39
7.7.4.9	ABSPRIO	39
7.7.4.10	TIME_IMMEDIATE	39
7.7.4.11	TIME_INFINITE	39
7.7.4.12	firstprio	39
7.7.4.13	currp	40
7.7.4.14	setcurrp	40
7.7.4.15	chSchIsRescRequiredl	40
7.7.4.16	chSchCanYieldS	40
7.7.4.17	chSchDoYieldS	40
7.7.4.18	chSchPreemption	41
7.8	Threads	41
7.8.1	Detailed Description	41
7.8.2	Function Documentation	44
7.8.2.1	_thread_init	44
7.8.2.2	_thread_memfill	44
7.8.2.3	chThdCreateI	44
7.8.2.4	chThdCreateStatic	45
7.8.2.5	chThdSetPriority	46
7.8.2.6	chThdResume	47
7.8.2.7	chThdTerminate	47
7.8.2.8	chThdSleep	48
7.8.2.9	chThdSleepUntil	48
7.8.2.10	chThdYield	48
7.8.2.11	chThdExit	48
7.8.2.12	chThdExitS	49
7.8.2.13	chThdWait	50
7.8.3	Define Documentation	51

7.8.3.1	THD_STATE_READY	51
7.8.3.2	THD_STATE_CURRENT	51
7.8.3.3	THD_STATE_SUSPENDED	51
7.8.3.4	THD_STATE_WTSEM	51
7.8.3.5	THD_STATE_WTMTX	51
7.8.3.6	THD_STATE_WTCOND	51
7.8.3.7	THD_STATE_SLEEPING	52
7.8.3.8	THD_STATE_WTEXIT	52
7.8.3.9	THD_STATE_WTOREVT	52
7.8.3.10	THD_STATE_WTANDEV	52
7.8.3.11	THD_STATE_SNDMSGQ	52
7.8.3.12	THD_STATE_SNDMSG	52
7.8.3.13	THD_STATE_WTMSG	52
7.8.3.14	THD_STATE_WTQUEUE	52
7.8.3.15	THD_STATE_FINAL	52
7.8.3.16	THD_STATE_NAMES	52
7.8.3.17	THD_MEM_MODE_MASK	52
7.8.3.18	THD_MEM_MODE_STATIC	53
7.8.3.19	THD_MEM_MODE_HEAP	53
7.8.3.20	THD_MEM_MODE_MEMPOOL	53
7.8.3.21	THD_TERMINATE	53
7.8.3.22	chThdSelf	53
7.8.3.23	chThdGetPriority	53
7.8.3.24	chThdGetTicks	53
7.8.3.25	chThdLSS	53
7.8.3.26	chThdTerminated	54
7.8.3.27	chThdShouldTerminate	54
7.8.3.28	chThdResumel	54
7.8.3.29	chThdSleepS	54
7.8.3.30	chThdSleepSeconds	55
7.8.3.31	chThdSleepMilliseconds	55
7.8.3.32	chThdSleepMicroseconds	55
7.8.4	Typedef Documentation	55
7.8.4.1	tfunc_t	56
7.9	Time and Virtual Timers	56
7.9.1	Detailed Description	56
7.9.2	Function Documentation	57
7.9.2.1	_vt_init	57
7.9.2.2	chVTSetl	57
7.9.2.3	chVTResetl	58

7.9.2.4	chTimelsWithin	58
7.9.3	Variable Documentation	59
7.9.3.1	vtlist	59
7.9.3.2	vtlist	59
7.9.4	Define Documentation	59
7.9.4.1	S2ST	59
7.9.4.2	MS2ST	59
7.9.4.3	US2ST	60
7.9.4.4	chVTDoTickl	60
7.9.4.5	chVTIsArmedl	60
7.9.4.6	chTimeNow	60
7.9.5	Typedef Documentation	61
7.9.5.1	vfunc_t	61
7.9.5.2	VirtualTimer	61
7.10	Synchronization	61
7.10.1	Detailed Description	61
7.11	Counting Semaphores	61
7.11.1	Detailed Description	61
7.11.2	Function Documentation	63
7.11.2.1	chSemInit	63
7.11.2.2	chSemReset	63
7.11.2.3	chSemResetl	64
7.11.2.4	chSemWait	65
7.11.2.5	chSemWaitS	65
7.11.2.6	chSemWaitTimeout	66
7.11.2.7	chSemWaitTimeoutS	67
7.11.2.8	chSemSignal	67
7.11.2.9	chSemSignall	68
7.11.2.10	chSemAddCounterl	69
7.11.2.11	chSemSignalWait	70
7.11.3	Define Documentation	70
7.11.3.1	_SEMAPHORE_DATA	70
7.11.3.2	SEMAPHORE_DECL	70
7.11.3.3	chSemFastWaitl	71
7.11.3.4	chSemFastSignall	71
7.11.3.5	chSemGetCounterl	71
7.11.4	Typedef Documentation	71
7.11.4.1	Semaphore	71
7.12	Binary Semaphores	71
7.12.1	Detailed Description	71

7.12.2 Define Documentation	73
7.12.2.1 _BSEMAPHORE_DATA	73
7.12.2.2 BSEMAPHORE_DECL	73
7.12.2.3 chBSemInit	73
7.12.2.4 chBSemWait	73
7.12.2.5 chBSemWaitS	74
7.12.2.6 chBSemWaitTimeout	74
7.12.2.7 chBSemWaitTimeoutS	74
7.12.2.8 chBSemReset	75
7.12.2.9 chBSemResetI	75
7.12.2.10 chBSemSignal	76
7.12.2.11 chBSemSignall	76
7.12.2.12 chBSemGetStatel	76
7.13 Mutexes	77
7.13.1 Detailed Description	77
7.13.2 Function Documentation	78
7.13.2.1 chMtxInit	78
7.13.2.2 chMtxLock	79
7.13.2.3 chMtxLockS	79
7.13.2.4 chMtxTryLock	80
7.13.2.5 chMtxTryLockS	81
7.13.2.6 chMtxUnlock	82
7.13.2.7 chMtxUnlockS	82
7.13.2.8 chMtxUnlockAll	83
7.13.3 Define Documentation	83
7.13.3.1 _MUTEX_DATA	83
7.13.3.2 MUTEX_DECL	84
7.13.3.3 chMtxQueueNotEmptyS	84
7.13.4 Typedef Documentation	84
7.13.4.1 Mutex	84
7.14 Condition Variables	84
7.14.1 Detailed Description	84
7.14.2 Function Documentation	85
7.14.2.1 chCondInit	85
7.14.2.2 chCondSignal	85
7.14.2.3 chCondSignall	86
7.14.2.4 chCondBroadcast	87
7.14.2.5 chCondBroadcastl	87
7.14.2.6 chCondWait	88
7.14.2.7 chCondWaitS	89

7.14.2.8	chCondWaitTimeout	90
7.14.2.9	chCondWaitTimeoutS	91
7.14.3	Define Documentation	92
7.14.3.1	_CONDVAR_DATA	92
7.14.3.2	CONDVAR_DECL	92
7.14.4	Typedef Documentation	92
7.14.4.1	CondVar	92
7.15	Event Flags	93
7.15.1	Detailed Description	93
7.15.2	Function Documentation	95
7.15.2.1	chEvtRegisterMask	95
7.15.2.2	chEvtUnregister	95
7.15.2.3	chEvtClearFlags	95
7.15.2.4	chEvtAddFlags	96
7.15.2.5	chEvtSignalFlags	96
7.15.2.6	chEvtSignalFlagsI	96
7.15.2.7	chEvtBroadcastFlags	97
7.15.2.8	chEvtBroadcastFlagsI	98
7.15.2.9	chEvtDispatch	98
7.15.2.10	chEvtWaitOneTimeout	98
7.15.2.11	chEvtWaitAnyTimeout	99
7.15.2.12	chEvtWaitAllTimeout	100
7.15.2.13	chEvtWaitOne	101
7.15.2.14	chEvtWaitAny	102
7.15.2.15	chEvtWaitAll	102
7.15.3	Define Documentation	103
7.15.3.1	_EVENTSOURCE_DATA	103
7.15.3.2	EVENTSOURCE_DECL	103
7.15.3.3	ALL_EVENTS	103
7.15.3.4	EVENT_MASK	103
7.15.3.5	chEvtRegister	103
7.15.3.6	chEvtInit	104
7.15.3.7	chEvtIsListeningI	104
7.15.3.8	chEvtBroadcast	104
7.15.3.9	chEvtBroadcastI	104
7.15.4	Typedef Documentation	105
7.15.4.1	EventSource	105
7.15.4.2	evhandler_t	105
7.16	Synchronous Messages	105
7.16.1	Detailed Description	105

7.16.2 Function Documentation	106
7.16.2.1 chMsgSend	106
7.16.2.2 chMsgWait	106
7.16.2.3 chMsgRelease	107
7.16.3 Define Documentation	107
7.16.3.1 chMsgIsPending	107
7.16.3.2 chMsgGet	108
7.16.3.3 chMsgGetS	108
7.16.3.4 chMsgReleaseS	108
7.17 Mailboxes	109
7.17.1 Detailed Description	109
7.17.2 Function Documentation	110
7.17.2.1 chMBInit	110
7.17.2.2 chMBReset	111
7.17.2.3 chMBPost	111
7.17.2.4 chMBPostS	112
7.17.2.5 chMBPostI	113
7.17.2.6 chMBPostAhead	114
7.17.2.7 chMBPostAheadS	114
7.17.2.8 chMBPostAheadI	115
7.17.2.9 chMBFetch	116
7.17.2.10 chMBFetchS	117
7.17.2.11 chMBFetchI	118
7.17.3 Define Documentation	119
7.17.3.1 chMBSizel	119
7.17.3.2 chMBGetFreeCountI	119
7.17.3.3 chMBGetUsedCountI	119
7.17.3.4 chMBPeekI	120
7.17.3.5 _MAILBOX_DATA	120
7.17.3.6 MAILBOX_DECL	120
7.18 Memory Management	120
7.18.1 Detailed Description	120
7.19 Core Memory Manager	121
7.19.1 Detailed Description	121
7.19.2 Function Documentation	122
7.19.2.1 _core_init	122
7.19.2.2 chCoreAlloc	122
7.19.2.3 chCoreAllocI	123
7.19.2.4 chCoreStatus	123
7.19.3 Define Documentation	123

7.19.3.1	MEM_ALIGN_SIZE	123
7.19.3.2	MEM_ALIGN_MASK	123
7.19.3.3	MEM_ALIGN_PREV	124
7.19.3.4	MEM_ALIGN_NEXT	124
7.19.3.5	MEM_IS_ALIGNED	124
7.19.4	Typedef Documentation	124
7.19.4.1	memgetfunc_t	124
7.20	Heaps	124
7.20.1	Detailed Description	124
7.20.2	Function Documentation	125
7.20.2.1	_heap_init	125
7.20.2.2	chHeapInit	125
7.20.2.3	chHeapAlloc	126
7.20.2.4	chHeapFree	126
7.20.2.5	chHeapStatus	126
7.21	Memory Pools	127
7.21.1	Detailed Description	127
7.21.2	Function Documentation	128
7.21.2.1	chPoolInit	128
7.21.2.2	chPoolAlloc	128
7.21.2.3	chPoolAlloc	129
7.21.2.4	chPoolFree	129
7.21.2.5	chPoolFree	130
7.21.3	Define Documentation	130
7.21.3.1	_MEMORYPOOL_DATA	130
7.21.3.2	MEMORYPOOL_DECL	131
7.22	Dynamic Threads	131
7.22.1	Detailed Description	131
7.22.2	Function Documentation	131
7.22.2.1	chThdAddRef	131
7.22.2.2	chThdRelease	132
7.22.2.3	chThdCreateFromHeap	132
7.22.2.4	chThdCreateFromMemoryPool	133
7.23	I/O Support	134
7.23.1	Detailed Description	134
7.24	Abstract Sequential Streams	134
7.24.1	Detailed Description	134
7.24.2	Define Documentation	135
7.24.2.1	_base_sequential_stream_methods	135
7.24.2.2	_base_sequential_stream_data	135

7.24.2.3	chSequentialStreamWrite	135
7.24.2.4	chSequentialStreamRead	136
7.25	Abstract File Streams	136
7.25.1	Detailed Description	136
7.25.2	Define Documentation	137
7.25.2.1	FILE_OK	137
7.25.2.2	FILE_ERROR	137
7.25.2.3	_base_file_stream_methods	137
7.25.2.4	_base_file_stream_data	138
7.25.2.5	chFileStreamClose	138
7.25.2.6	chFileStreamGetError	138
7.25.2.7	chFileStreamGetSize	138
7.25.2.8	chFileStreamGetPosition	139
7.25.2.9	chFileStreamSeek	139
7.25.3	Typedef Documentation	139
7.25.3.1	fileoffset_t	139
7.26	Abstract I/O Channels	139
7.26.1	Detailed Description	139
7.26.2	Define Documentation	141
7.26.2.1	_base_channel_methods	141
7.26.2.2	_base_channel_data	141
7.26.2.3	chIOPutWouldBlock	141
7.26.2.4	chIOGetWouldBlock	142
7.26.2.5	chIOPut	142
7.26.2.6	chIOPutTimeout	143
7.26.2.7	chIOGet	143
7.26.2.8	chIOGetTimeout	143
7.26.2.9	chIOWriteTimeout	144
7.26.2.10	chIORReadTimeout	144
7.26.2.11	IO_NO_ERROR	145
7.26.2.12	IO_CONNECTED	145
7.26.2.13	IO_DISCONNECTED	145
7.26.2.14	IO_INPUT_AVAILABLE	145
7.26.2.15	IO_OUTPUT_EMPTY	145
7.26.2.16	IO_TRANSMISSION_END	145
7.26.2.17	_base_asynchronous_channel_methods	145
7.26.2.18	_base_asynchronous_channel_data	146
7.26.2.19	chIOGetEventSource	146
7.26.2.20	chIOAddFlags!	146
7.26.2.21	chIOGetAndClearFlags	146

7.26.2.22 <code>_ch_get_and_clear_flags_impl</code>	147
7.26.3 <code>Typedef Documentation</code>	147
7.26.3.1 <code>ioflags_t</code>	147
7.27 <code>I/O Queues</code>	147
7.27.1 <code>Detailed Description</code>	147
7.27.2 <code>Function Documentation</code>	150
7.27.2.1 <code>chIQInit</code>	150
7.27.2.2 <code>chIQResetl</code>	150
7.27.2.3 <code>chIQPutl</code>	151
7.27.2.4 <code>chIQGetTimeout</code>	151
7.27.2.5 <code>chIQReadTimeout</code>	152
7.27.2.6 <code>chOQInit</code>	152
7.27.2.7 <code>chOQResetl</code>	153
7.27.2.8 <code>chOQPutTimeout</code>	153
7.27.2.9 <code>chOQGetl</code>	154
7.27.2.10 <code>chOQWriteTimeout</code>	155
7.27.3 <code>Define Documentation</code>	155
7.27.3.1 <code>Q_OK</code>	155
7.27.3.2 <code>Q_TIMEOUT</code>	156
7.27.3.3 <code>Q_RESET</code>	156
7.27.3.4 <code>Q_EMPTY</code>	156
7.27.3.5 <code>Q_FULL</code>	156
7.27.3.6 <code>chQSizeI</code>	156
7.27.3.7 <code>chQSpaceI</code>	156
7.27.3.8 <code>chIQGetFullI</code>	156
7.27.3.9 <code>chIQGetEmptyI</code>	157
7.27.3.10 <code>chIQIsEmptyI</code>	157
7.27.3.11 <code>chIQIsFullI</code>	158
7.27.3.12 <code>chIQGet</code>	158
7.27.3.13 <code>_INPUTQUEUE_DATA</code>	158
7.27.3.14 <code>INPUTQUEUE_DECL</code>	159
7.27.3.15 <code>chOQGetFullI</code>	159
7.27.3.16 <code>chOQGetEmptyI</code>	159
7.27.3.17 <code>chOQIsEmptyI</code>	160
7.27.3.18 <code>chOQIsFullI</code>	160
7.27.3.19 <code>chOQPut</code>	160
7.27.3.20 <code>_OUTPUTQUEUE_DATA</code>	161
7.27.3.21 <code>OUTPUTQUEUE_DECL</code>	161
7.27.4 <code>Typedef Documentation</code>	162
7.27.4.1 <code>GenericQueue</code>	162

7.27.4.2	qnotify_t	162
7.27.4.3	InputQueue	162
7.27.4.4	OutputQueue	162
7.28	Registry	162
7.28.1	Detailed Description	162
7.28.2	Function Documentation	163
7.28.2.1	chRegFirstThread	163
7.28.2.2	chRegNextThread	163
7.28.3	Define Documentation	164
7.28.3.1	chRegSetThreadName	164
7.28.3.2	chRegGetThreadName	164
7.28.3.3	REG_REMOVE	165
7.28.3.4	REG_INSERT	165
7.29	Debug	165
7.29.1	Detailed Description	165
7.29.2	Function Documentation	167
7.29.2.1	_trace_init	167
7.29.2.2	dbg_trace	167
7.29.2.3	dbg_check_disable	168
7.29.2.4	dbg_check_suspend	168
7.29.2.5	dbg_check_enable	168
7.29.2.6	dbg_check_lock	169
7.29.2.7	dbg_check_unlock	169
7.29.2.8	dbg_check_lock_from_isr	170
7.29.2.9	dbg_check_unlock_from_isr	170
7.29.2.10	dbg_check_enter_isr	170
7.29.2.11	dbg_check_leave_isr	171
7.29.2.12	chDbgCheckClassI	171
7.29.2.13	chDbgCheckClassS	172
7.29.2.14	chDbgPanic	172
7.29.3	Variable Documentation	172
7.29.3.1	dbg_panic_msg	172
7.29.3.2	dbg_isr_cnt	172
7.29.3.3	dbg_lock_cnt	172
7.29.3.4	dbg_trace_buffer	172
7.29.3.5	dbg_panic_msg	172
7.29.4	Define Documentation	173
7.29.4.1	CH_TRACE_BUFFER_SIZE	173
7.29.4.2	CH_STACK_FILL_VALUE	173
7.29.4.3	CH_THREAD_FILL_VALUE	173

7.29.4.4	chDbgCheck	173
7.29.4.5	chDbgAssert	173
7.30	Internals	174
7.30.1	Detailed Description	174
7.30.2	Function Documentation	175
7.30.2.1	prio_insert	175
7.30.2.2	queue_insert	175
7.30.2.3	fifo_remove	175
7.30.2.4	lifo_remove	176
7.30.2.5	dequeue	176
7.30.2.6	list_insert	176
7.30.2.7	list_remove	177
7.30.3	Define Documentation	177
7.30.3.1	queue_init	177
7.30.3.2	list_init	177
7.30.3.3	isempty	177
7.30.3.4	notempty	177
7.30.3.5	_THREADSQUEUE_DATA	178
7.30.3.6	THREADSQUEUE_DECL	178
7.31	ARM7/9	178
7.31.1	Detailed Description	178
7.31.2	Introduction	178
7.31.3	Mapping of the System States in the ARM7/9 port	178
7.31.4	The ARM7/9 port notes	179
7.31.5	ARM7/9 Interrupt Handlers	179
7.32	Configuration Options	180
7.33	Core Port Implementation	181
7.33.1	Detailed Description	181
7.33.2	Function Documentation	183
7.33.2.1	port_halt	183
7.33.3	Define Documentation	183
7.33.3.1	ARM_CORE_ARM7TDMI	183
7.33.3.2	ARM_CORE_ARM9	183
7.33.3.3	ARM_ENABLE_WFI_IDLE	183
7.33.3.4	CH_ARCHITECTURE_ARM	183
7.33.3.5	CH_ARCHITECTURE_ARMx	183
7.33.3.6	CH_ARCHITECTURE_NAME	183
7.33.3.7	CH_CORE_VARIANT_NAME	183
7.33.3.8	CH_PORT_INFO	184
7.33.3.9	CH_PORT_INFO	184

7.33.3.10 CH_COMPILER_NAME	184
7.33.3.11 SETUP_CONTEXT	184
7.33.3.12 PORT_IDLE_THREAD_STACK_SIZE	185
7.33.3.13 PORT_INT_REQUIRED_STACK	185
7.33.3.14 STACK_ALIGN	185
7.33.3.15 THD_WA_SIZE	185
7.33.3.16 WORKING_AREA	185
7.33.3.17 PORT_IRQ_PROLOGUE	185
7.33.3.18 PORT_IRQ_EPILOGUE	186
7.33.3.19 PORT_IRQ_HANDLER	186
7.33.3.20 PORT_FAST_IRQ_HANDLER	186
7.33.3.21 port_init	186
7.33.3.22 port_lock	186
7.33.3.23 port_unlock	186
7.33.3.24 port_lock_from_isr	187
7.33.3.25 port_unlock_from_isr	187
7.33.3.26 port_disable	187
7.33.3.27 port_suspend	187
7.33.3.28 port_enable	187
7.33.3.29 port_switch	188
7.33.3.30 INLINE	188
7.33.3.31 ROMCONST	188
7.33.3.32 PACK_STRUCT_STRUCT	188
7.33.3.33 PACK_STRUCT_BEGIN	188
7.33.3.34 PACK_STRUCT_END	189
7.33.4 Typedef Documentation	189
7.33.4.1 stkalign_t	189
7.33.4.2 regarm_t	189
7.33.4.3 bool_t	189
7.33.4.4 tmode_t	189
7.33.4.5 tstate_t	189
7.33.4.6 trefs_t	189
7.33.4.7 tprio_t	189
7.33.4.8 msg_t	189
7.33.4.9 eventid_t	189
7.33.4.10 eventmask_t	189
7.33.4.11 systime_t	190
7.33.4.12 cnt_t	190
7.34 Startup Support	190
7.34.1 Startup Process	190

7.34.2	Expected linker symbols	190
7.35	Specific Implementations	191
7.35.1	Detailed Description	191
7.36	AT91SAM7 Specific Parameters	191
7.36.1	Detailed Description	191
7.36.2	Define Documentation	191
7.36.2.1	ARM_CORE	191
7.36.2.2	port_wait_for_interrupt	191
7.37	AT91SAM7 Interrupt Vectors	192
7.37.1	Detailed Description	192
7.37.2	Function Documentation	192
7.37.2.1	_unhandled_exception	192
7.38	LPC214x Specific Parameters	192
7.38.1	Detailed Description	192
7.38.2	Define Documentation	192
7.38.2.1	ARM_CORE	192
7.38.2.2	port_wait_for_interrupt	192
7.39	LPC214x Interrupt Vectors	193
7.39.1	Detailed Description	193
7.39.2	Function Documentation	193
7.39.2.1	_unhandled_exception	193
8	Data Structure Documentation	194
8.1	BaseAsynchronousChannel Struct Reference	194
8.1.1	Detailed Description	194
8.1.2	Field Documentation	195
8.1.2.1	vmt	195
8.2	BaseAsynchronousChannelVMT Struct Reference	195
8.2.1	Detailed Description	195
8.3	BaseChannel Struct Reference	197
8.3.1	Detailed Description	197
8.3.2	Field Documentation	199
8.3.2.1	vmt	199
8.4	BaseChannelVMT Struct Reference	199
8.4.1	Detailed Description	199
8.5	BaseFileStream Struct Reference	200
8.5.1	Detailed Description	201
8.5.2	Field Documentation	202
8.5.2.1	vmt	202
8.6	BaseFileStreamVMT Struct Reference	202

8.6.1	Detailed Description	202
8.7	BaseSequentialStream Struct Reference	203
8.7.1	Detailed Description	203
8.7.2	Field Documentation	205
8.7.2.1	vmt	205
8.8	BaseSequentialStreamVMT Struct Reference	205
8.8.1	Detailed Description	205
8.9	BinarySemaphore Struct Reference	205
8.9.1	Detailed Description	206
8.10	ch_swc_event_t Struct Reference	207
8.10.1	Detailed Description	207
8.10.2	Field Documentation	209
8.10.2.1	se_time	209
8.10.2.2	se_tp	209
8.10.2.3	se_wtobjp	209
8.10.2.4	se_state	209
8.11	ch_trace_buffer_t Struct Reference	209
8.11.1	Detailed Description	209
8.11.2	Field Documentation	211
8.11.2.1	tb_size	211
8.11.2.2	tb_ptr	211
8.11.2.3	tb_buffer	211
8.12	CondVar Struct Reference	211
8.12.1	Detailed Description	211
8.12.2	Field Documentation	213
8.12.2.1	c_queue	213
8.13	context Struct Reference	213
8.13.1	Detailed Description	213
8.14	EventListener Struct Reference	213
8.14.1	Detailed Description	213
8.14.2	Field Documentation	215
8.14.2.1	el_next	215
8.14.2.2	el_listener	215
8.14.2.3	el_mask	215
8.15	EventSource Struct Reference	215
8.15.1	Detailed Description	215
8.15.2	Field Documentation	217
8.15.2.1	es_next	217
8.16	extctx Struct Reference	217
8.16.1	Detailed Description	217

8.17 GenericQueue Struct Reference	217
8.17.1 Detailed Description	217
8.17.2 Field Documentation	219
8.17.2.1 q_waiting	219
8.17.2.2 q_counter	219
8.17.2.3 q_buffer	219
8.17.2.4 q_top	219
8.17.2.5 q_wptr	219
8.17.2.6 q_rdptr	219
8.17.2.7 q_notify	219
8.18 heap_header Union Reference	219
8.18.1 Detailed Description	219
8.18.2 Field Documentation	220
8.18.2.1 next	220
8.18.2.2 heap	220
8.18.2.3 u	220
8.18.2.4 size	220
8.19 intctx Struct Reference	220
8.19.1 Detailed Description	220
8.20 Mailbox Struct Reference	221
8.20.1 Detailed Description	221
8.20.2 Field Documentation	222
8.20.2.1 mb_buffer	222
8.20.2.2 mb_top	222
8.20.2.3 mb_wptr	222
8.20.2.4 mb_rdptr	222
8.20.2.5 mb_fullsem	222
8.20.2.6 mb_emptysem	222
8.21 memory_heap Struct Reference	222
8.21.1 Detailed Description	222
8.21.2 Field Documentation	224
8.21.2.1 h_provider	224
8.21.2.2 h_free	224
8.21.2.3 h_mtx	224
8.22 MemoryPool Struct Reference	224
8.22.1 Detailed Description	224
8.22.2 Field Documentation	225
8.22.2.1 mp_next	225
8.22.2.2 mp_object_size	225
8.22.2.3 mp_provider	225

8.23 Mutex Struct Reference	225
8.23.1 Detailed Description	225
8.23.2 Field Documentation	227
8.23.2.1 m_queue	227
8.23.2.2 m_owner	227
8.23.2.3 m_next	227
8.24 pool_header Struct Reference	227
8.24.1 Detailed Description	227
8.24.2 Field Documentation	227
8.24.2.1 ph_next	227
8.25 ReadyList Struct Reference	228
8.25.1 Detailed Description	228
8.25.2 Field Documentation	230
8.25.2.1 r_queue	230
8.25.2.2 r_prio	230
8.25.2.3 r_ctx	230
8.25.2.4 r_newer	230
8.25.2.5 r_older	230
8.25.2.6 r_preempt	230
8.25.2.7 r_current	230
8.26 Semaphore Struct Reference	230
8.26.1 Detailed Description	230
8.26.2 Field Documentation	233
8.26.2.1 s_queue	233
8.26.2.2 s_cnt	233
8.27 Thread Struct Reference	233
8.27.1 Detailed Description	233
8.27.2 Field Documentation	236
8.27.2.1 p_next	236
8.27.2.2 p_prev	237
8.27.2.3 p_prio	237
8.27.2.4 p_ctx	237
8.27.2.5 p_newer	237
8.27.2.6 p_older	237
8.27.2.7 p_name	237
8.27.2.8 p_stklimit	237
8.27.2.9 p_state	237
8.27.2.10 p_flags	237
8.27.2.11 p_refs	237
8.27.2.12 p_time	237

8.27.2.13 rdymsg	238
8.27.2.14 exitcode	238
8.27.2.15 wtobjp	238
8.27.2.16 ewmask	238
8.27.2.17 p_waiting	238
8.27.2.18 p_msgqueue	238
8.27.2.19 p_msg	238
8.27.2.20 p_epending	238
8.27.2.21 p_mtxlist	239
8.27.2.22 p_realprio	239
8.27.2.23 p_mpool	239
8.28 ThreadsList Struct Reference	239
8.28.1 Detailed Description	239
8.28.2 Field Documentation	242
8.28.2.1 p_next	242
8.29 ThreadsQueue Struct Reference	242
8.29.1 Detailed Description	242
8.29.2 Field Documentation	245
8.29.2.1 p_next	245
8.29.2.2 p_prev	245
8.30 VirtualTimer Struct Reference	245
8.30.1 Detailed Description	245
8.30.2 Field Documentation	246
8.30.2.1 vt_next	246
8.30.2.2 vt_prev	246
8.30.2.3 vt_time	246
8.30.2.4 vt_func	246
8.30.2.5 vt_par	246
8.31 VTList Struct Reference	246
8.31.1 Detailed Description	246
8.31.2 Field Documentation	247
8.31.2.1 vt_next	247
8.31.2.2 vt_prev	247
8.31.2.3 vt_time	248
8.31.2.4 vt_systime	248
9 File Documentation	249
9.1 armparams.h File Reference	249
9.1.1 Detailed Description	249
9.2 armparams.h File Reference	249

9.2.1	Detailed Description	249
9.3	ch.h File Reference	249
9.3.1	Detailed Description	249
9.4	chbsem.h File Reference	251
9.4.1	Detailed Description	251
9.5	chcond.c File Reference	251
9.5.1	Detailed Description	251
9.6	chcond.h File Reference	252
9.6.1	Detailed Description	252
9.7	chconf.h File Reference	253
9.7.1	Detailed Description	253
9.8	chcore.c File Reference	255
9.8.1	Detailed Description	255
9.9	chcore.h File Reference	255
9.9.1	Detailed Description	255
9.10	chcoreasm.s File Reference	256
9.10.1	Detailed Description	256
9.11	chdebug.c File Reference	257
9.11.1	Detailed Description	257
9.12	chdebug.h File Reference	258
9.12.1	Detailed Description	258
9.13	chdynamic.c File Reference	258
9.13.1	Detailed Description	258
9.14	chdynamic.h File Reference	259
9.14.1	Detailed Description	259
9.15	chevents.c File Reference	259
9.15.1	Detailed Description	259
9.16	chevents.h File Reference	260
9.16.1	Detailed Description	260
9.17	chfiles.h File Reference	261
9.17.1	Detailed Description	261
9.18	chheap.c File Reference	262
9.18.1	Detailed Description	262
9.19	chheap.h File Reference	262
9.19.1	Detailed Description	262
9.20	chinline.h File Reference	263
9.20.1	Detailed Description	263
9.21	chioch.h File Reference	263
9.21.1	Detailed Description	263
9.22	chlsts.c File Reference	265

9.22.1 Detailed Description	265
9.23 chlists.h File Reference	265
9.23.1 Detailed Description	265
9.24 chmboxes.c File Reference	266
9.24.1 Detailed Description	266
9.25 chmboxes.h File Reference	267
9.25.1 Detailed Description	267
9.26 chmemcore.c File Reference	268
9.26.1 Detailed Description	268
9.27 chmemcore.h File Reference	268
9.27.1 Detailed Description	268
9.28 chmempools.c File Reference	269
9.28.1 Detailed Description	269
9.29 chmempools.h File Reference	269
9.29.1 Detailed Description	269
9.30 chmsg.c File Reference	270
9.30.1 Detailed Description	270
9.31 chmsg.h File Reference	270
9.31.1 Detailed Description	270
9.32 chmtx.c File Reference	271
9.32.1 Detailed Description	271
9.33 chmtx.h File Reference	271
9.33.1 Detailed Description	271
9.34 chqueues.c File Reference	272
9.34.1 Detailed Description	272
9.35 chqueues.h File Reference	273
9.35.1 Detailed Description	273
9.36 chregistry.c File Reference	275
9.36.1 Detailed Description	275
9.37 chregistry.h File Reference	275
9.37.1 Detailed Description	275
9.38 chschd.c File Reference	276
9.38.1 Detailed Description	276
9.39 chschd.h File Reference	276
9.39.1 Detailed Description	276
9.40 chsem.c File Reference	278
9.40.1 Detailed Description	278
9.41 chsem.h File Reference	278
9.41.1 Detailed Description	278
9.42 chstreams.h File Reference	279

9.42.1 Detailed Description	279
9.43 chsys.c File Reference	280
9.43.1 Detailed Description	280
9.44 chsys.h File Reference	280
9.44.1 Detailed Description	280
9.45 chthreads.c File Reference	281
9.45.1 Detailed Description	281
9.46 chthreads.h File Reference	282
9.46.1 Detailed Description	282
9.47 chtypes.h File Reference	284
9.47.1 Detailed Description	284
9.48 chvt.c File Reference	285
9.48.1 Detailed Description	285
9.49 chvt.h File Reference	285
9.49.1 Detailed Description	285
9.50 crt0.s File Reference	286
9.50.1 Detailed Description	286
9.51 vectors.s File Reference	286
9.51.1 Detailed Description	286
9.52 vectors.s File Reference	286
9.52.1 Detailed Description	286

Chapter 1

ChibiOS/RT

1.1 Copyright

Copyright (C) 2006..2011 Giovanni Di Sirio. All rights reserved.

Neither the whole nor any part of the information contained in this document may be adapted or reproduced in any form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Giovanni Di Sirio in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Giovanni Di Sirio shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

1.2 Introduction

This document is the Reference Manual for the ChibiOS/RT portable Kernel API and the ARM port for the GCC compiler.

1.3 Related Documents

- ChibiOS/RT General Architecture

Chapter 2

Kernel Concepts

ChibiOS/RT Kernel Concepts

- [Naming Conventions](#)
- [API Name Suffixes](#)
- [Interrupt Classes](#)
- [System States](#)
- [Scheduling](#)
- [Thread States](#)
- [Priority Levels](#)
- [Thread Working Area](#)

2.1 Naming Conventions

ChibiOS/RT APIs are all named following this convention: `ch<group><action><suffix>()`. The possible groups are: `Sys`, `Sch`, `Time`, `VT`, `Thd`, `Sem`, `Mtx`, `Cond`, `Evt`, `Msg`, `Reg`, `SequentialStream`, `IO`, `IQ`, `OQ`, `Dbg`, `Core`, `Heap`, `Pool`.

2.2 API Name Suffixes

The suffix can be one of the following:

- **None**, APIs without any suffix can be invoked only from the user code in the **Normal** state unless differently specified. See [System States](#).
- "**I**", I-Class APIs are invokable only from the **I-Locked** or **S-Locked** states. See [System States](#).
- "**S**", S-Class APIs are invokable only from the **S-Locked** state. See [System States](#).

Examples: `chThdCreateStatic()`, `chSemSignalI()`, `chIQGetTimeout()`.

2.3 Interrupt Classes

In ChibiOS/RT there are three logical interrupt classes:

- **Regular Interrupts.** Maskable interrupt sources that cannot preempt (small parts of) the kernel code and are thus able to invoke operating system APIs from within their handlers. The interrupt handlers belonging to this class must be written following some rules. See the [System Management APIs](#) group and the web article [How to write interrupt handlers](#).
- **Fast Interrupts.** Maskable interrupt sources with the ability to preempt the kernel code and thus have a lower latency and are less subject to jitter, see the web article [Response Time and Jitter](#). Such sources are not supported on all the architectures.
Fast interrupts are not allowed to invoke any operating system API from within their handlers. Fast interrupt sources may, however, pend a lower priority regular interrupt where access to the operating system is possible.
- **Non Maskable Interrupts.** Non maskable interrupt sources are totally out of the operating system control and have the lowest latency. Such sources are not supported on all the architectures.

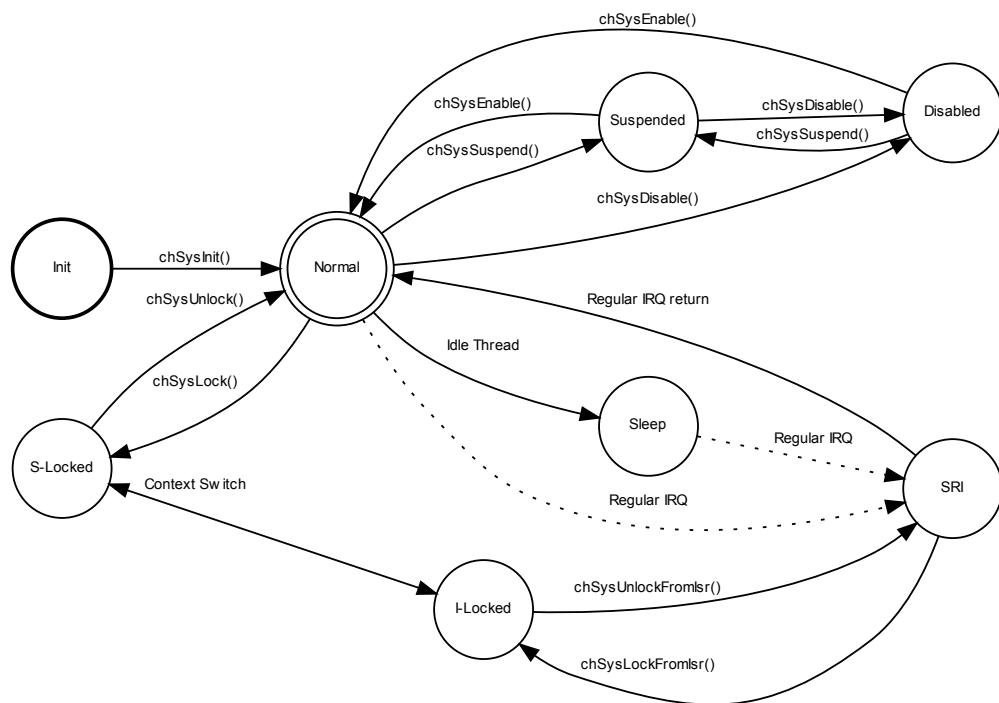
The mapping of the above logical classes into physical interrupts priorities is, of course, port dependent. See the documentation of the various ports for details.

2.4 System States

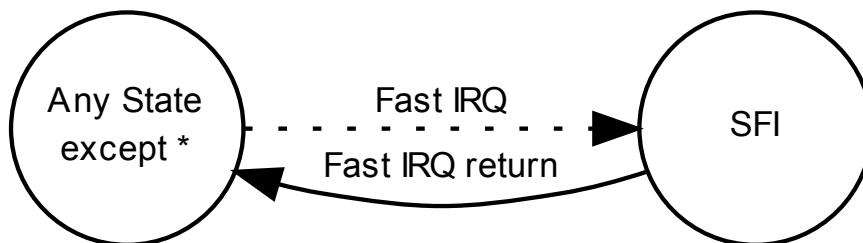
When using ChibiOS/RT the system can be in one of the following logical operating states:

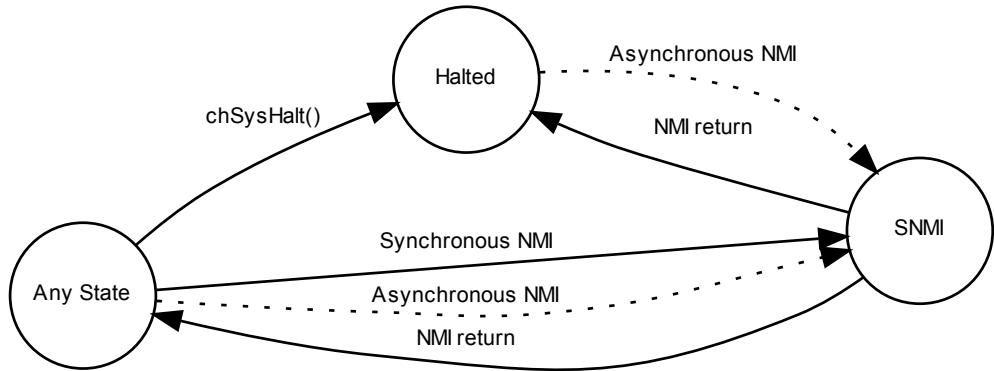
- **Init.** When the system is in this state all the maskable interrupt sources are disabled. In this state it is not possible to use any system API except [chSysInit\(\)](#). This state is entered after a physical reset.
- **Normal.** All the interrupt sources are enabled and the system APIs are accessible, threads are running.
- **Suspended.** In this state the fast interrupt sources are enabled but the regular interrupt sources are not. In this state it is not possible to use any system API except [chSysDisable\(\)](#) or [chSysEnable\(\)](#) in order to change state.
- **Disabled.** When the system is in this state both the maskable regular and fast interrupt sources are disabled. In this state it is not possible to use any system API except [chSysSuspend\(\)](#) or [chSysEnable\(\)](#) in order to change state.
- **Sleep.** Architecture-dependent low power mode, the idle thread goes in this state and waits for interrupts, after servicing the interrupt the Normal state is restored and the scheduler has a chance to reschedule.
- **S-Locked.** Kernel locked and regular interrupt sources disabled. Fast interrupt sources are enabled. [S-Class](#) and [I-Class](#) APIs are invokable in this state.
- **I-Locked.** Kernel locked and regular interrupt sources disabled. [I-Class](#) APIs are invokable from this state.
- **Serving Regular Interrupt.** No system APIs are accessible but it is possible to switch to the I-Locked state using [chSysLockFromIsr\(\)](#) and then invoke any [I-Class](#) API. Interrupt handlers can be preemptable on some architectures thus is important to switch to I-Locked state before invoking system APIs.
- **Serving Fast Interrupt.** System APIs are not accessible.
- **Serving Non-Maskable Interrupt.** System APIs are not accessible.
- **Halted.** All interrupt sources are disabled and system stopped into an infinite loop. This state can be reached if the debug mode is activated **and** an error is detected **or** after explicitly invoking [chSysHalt\(\)](#).

Note that the above states are just **Logical States** that may have no real associated machine state on some architectures. The following diagram shows the possible transitions between the states:



Note, the **SFI**, **Halted** and **SNMI** states were not shown because those are reachable from most states:

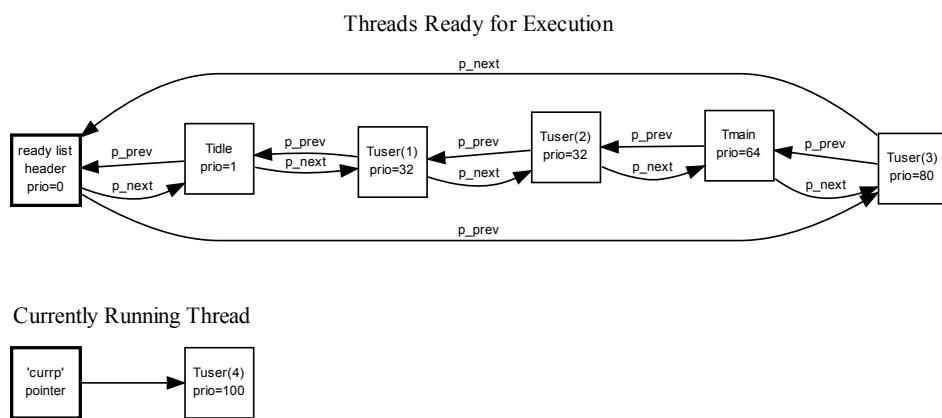


**Attention**

* except: **Init, Halt, SNMI, Disabled**.

2.5 Scheduling

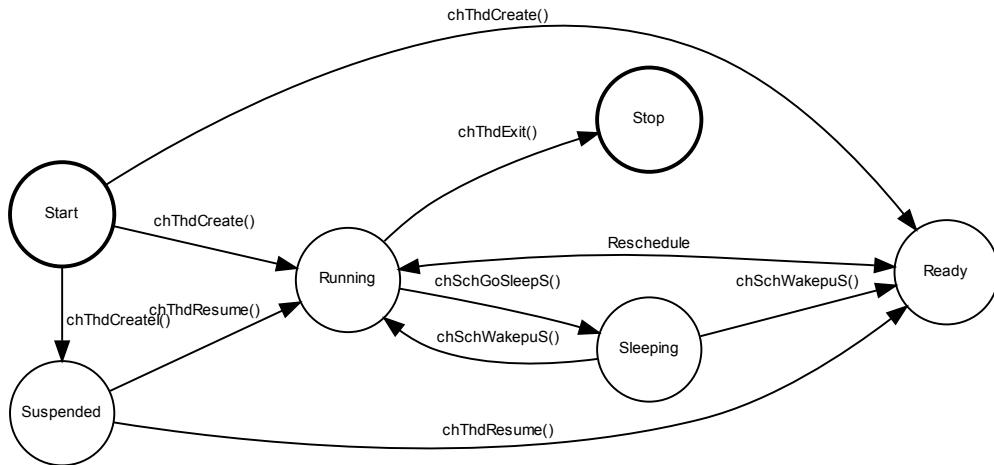
The strategy is very simple the currently ready thread with the highest priority is executed. If more than one thread with equal priority are eligible for execution then they are executed in a round-robin way, the CPU time slice constant is configurable. The ready list is a double linked list of threads ordered by priority.



Note that the currently running thread is not in the ready list, the list only contains the threads ready to be executed but still actually waiting.

2.6 Thread States

The image shows how threads can change their state in ChibiOS/RT.



2.7 Priority Levels

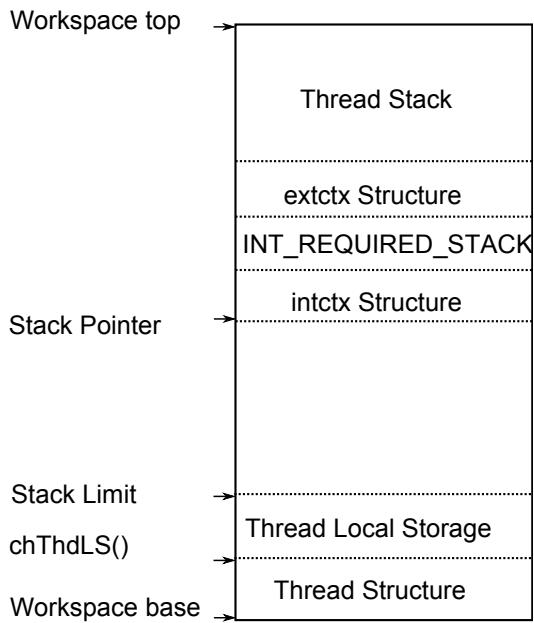
Priorities in ChibiOS/RT are a contiguous numerical range but the initial and final values are not enforced.

The following table describes the various priority boundaries (from lowest to highest):

- **IDLEPRIO**, this is the lowest priority level and is reserved for the idle thread, no other threads should share this priority level. This is the lowest numerical value of the priorities space.
- **LOWPRIO**, the lowest priority level that can be assigned to an user thread.
- **NORMALPRIO**, this is the central priority level for user threads. It is advisable to assign priorities to threads as values relative to NORMALPRIO, as example NORMALPRIO-1 or NORMALPRIO+4, this ensures the portability of code should the numerical range change in future implementations.
- **HIGHPRIO**, the highest priority level that can be assigned to an user thread.
- **ABSPRIO**, absolute maximum software priority level, it can be higher than HIGHPRIO but the numerical values above HIGHPRIO up to ABSPRIO (inclusive) are reserved. This is the highest numerical value of the priorities space.

2.8 Thread Working Area

Each thread has its own stack, a [Thread](#) structure and some preemption areas. All the structures are allocated into a "Thread Working Area", a thread private heap, usually statically declared in your code. Threads do not use any memory outside the allocated working area except when accessing static shared data.



Note that the preemption area is only present when the thread is not running (switched out), the context switching is done by pushing the registers on the stack of the switched-out thread and popping the registers of the switched-in thread from its stack. The preemption area can be divided in up to three structures:

- External Context.
- Interrupt Stack.
- Internal Context.

See the port documentation for details, the area may change on the various ports and some structures may not be present (or be zero-sized).

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Kernel	14
Version Numbers and Identification	14
Configuration	15
Types	25
Base Kernel Services	25
System Management	25
Scheduler	33
Threads	41
Time and Virtual Timers	56
Synchronization	61
Counting Semaphores	61
Binary Semaphores	71
Mutexes	77
Condition Variables	84
Event Flags	93
Synchronous Messages	105
Mailboxes	109
Memory Management	120
Core Memory Manager	121
Heaps	124
Memory Pools	127
Dynamic Threads	131
I/O Support	134
Abstract Sequential Streams	134
Abstract File Streams	136
Abstract I/O Channels	139
I/O Queues	147
Registry	162
Debug	165
Internals	174
ARM7/9	178
Configuration Options	180
Core Port Implementation	181
Startup Support	190
Specific Implementations	191
AT91SAM7 Specific Parameters	191
AT91SAM7 Interrupt Vectors	192

LPC214x Specific Parameters	192
LPC214x Interrupt Vectors	193

Chapter 4

Class Hierarchy Index

4.1 Class Hierarchy

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

BaseSequentialStream	203
BaseChannel	197
BaseAsynchronousChannel	194
BaseFileStream	200
BaseSequentialStreamVMT	205
BaseChannelVMT	199
BaseAsynchronousChannelVMT	195
BaseFileStreamVMT	202
ch_swc_event_t	207
ch_trace_buffer_t	209
CondVar	211
context	213
EventListener	213
EventSource	215
extctx	217
GenericQueue	217
heap_header	219
intctx	220
Mailbox	221
memory_heap	222
MemoryPool	224
Mutex	225
pool_header	227
Semaphore	230
BinarySemaphore	205
ThreadsList	239
ThreadsQueue	242
ReadyList	228
Thread	233
VirtualTimer	245
VTLList	246

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

BaseAsynchronousChannel (Base asynchronous channel class)	194
BaseAsynchronousChannelVMT (BaseAsynchronousChannel virtual methods table)	195
BaseChannel (Base channel class)	197
BaseChannelVMT (BaseChannel virtual methods table)	199
BaseFileStream (Base file stream class)	200
BaseFileStreamVMT (BaseFileStream virtual methods table)	202
BaseSequentialStream (Base stream class)	203
BaseSequentialStreamVMT (BaseSequentialStream virtual methods table)	205
BinarySemaphore (Binary semaphore type)	205
ch_swc_event_t (Trace buffer record)	207
ch_trace_buffer_t (Trace buffer header)	209
CondVar (CondVar structure)	211
context (Platform dependent part of the Thread structure)	213
EventListener (Event Listener structure)	213
EventSource (Event Source structure)	215
extctx (Interrupt saved context)	217
GenericQueue (Generic I/O queue structure)	217
heap_header (Memory heap block header)	219
intctx (System saved context)	220
Mailbox (Structure representing a mailbox object)	221
memory_heap (Structure describing a memory heap)	222
MemoryPool (Memory pool descriptor)	224
Mutex (Mutex structure)	225
pool_header (Memory pool free object header)	227
ReadyList (Ready list header)	228
Semaphore (Semaphore structure)	230
Thread (Structure representing a thread)	233
ThreadsList (Generic threads single link list, it works like a stack)	239
ThreadsQueue (Generic threads bidirectional linked list header and element)	242
VirtualTimer (Virtual Timer descriptor structure)	245
VTList (Virtual timers list header)	246

Chapter 6

File Index

6.1 File List

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

AT91SAM7/armparams.h (ARM7 AT91SAM7 Specific Parameters)	249
LPC214x/armparams.h (ARM7 LPC214x Specific Parameters)	249
ch.h (ChibiOS/RT main include file)	249
chbsem.h (Binary semaphores structures and macros)	251
chcond.c (Condition Variables code)	251
chcond.h (Condition Variables macros and structures)	252
chconf.h (Configuration file template)	253
chcore.c (ARM7/9 architecture port code)	255
chcore.h (ARM7/9 architecture port macros and structures)	255
chcoreasm.s (ARM7/9 architecture port low level code)	256
chdebug.c (ChibiOS/RT Debug code)	257
chdebug.h (Debug macros and structures)	258
chdynamic.c (Dynamic threads code)	258
chdynamic.h (Dynamic threads macros and structures)	259
chevents.c (Events code)	259
chevents.h (Events macros and structures)	260
chfiles.h (Data files)	261
chheap.c (Heaps code)	262
chheap.h (Heaps macros and structures)	262
chinline.h (Kernel inlined functions)	263
chioch.h (I/O channels)	263
chlists.c (Thread queues/lists code)	265
chlists.h (Thread queues/lists macros and structures)	265
chmboxes.c (Mailboxes code)	266
chmboxes.h (Mailboxes macros and structures)	267
chmemcore.c (Core memory manager code)	268
chmemcore.h (Core memory manager macros and structures)	268
chmpools.c (Memory Pools code)	269
chmpools.h (Memory Pools macros and structures)	269
chmsg.c (Messages code)	270
chmsg.h (Messages macros and structures)	270
chmtx.c (Mutexes code)	271
chmtx.h (Mutexes macros and structures)	271
chqueues.c (I/O Queues code)	272
chqueues.h (I/O Queues macros and structures)	273
chregistry.c (Threads registry code)	275
chregistry.h (Threads registry macros and structures)	275
chsched.c (Scheduler code)	276

chsched.h (Scheduler macros and structures)	276
chsem.c (Semaphores code)	278
chsem.h (Semaphores macros and structures)	278
chstreams.h (Data streams)	279
chsys.c (System related code)	280
chsys.h (System related macros and structures)	280
chthreads.c (Threads code)	281
chthreads.h (Threads macros and structures)	282
chtypes.h (ARM7/9 architecture port system types)	284
chvt.c (Time and Virtual Timers related code)	285
chvt.h (Time macros and structures)	285
crt0.s (Generic ARM7/9 startup file for ChibiOS/RT)	286
AT91SAM7/vectors.s (Interrupt vectors for the AT91SAM7 family)	286
LPC214x/vectors.s (Interrupt vectors for the LPC214x family)	286

Chapter 7

Module Documentation

7.1 Kernel

7.1.1 Detailed Description

The kernel is the portable part of ChibiOS/RT, this section documents the various kernel subsystems.

Modules

- Version Numbers and Identification
- Configuration
- Types
- Base Kernel Services
- Synchronization
- Memory Management
- I/O Support
- Registry
- Debug
- Internals

7.2 Version Numbers and Identification

7.2.1 Detailed Description

Kernel related info.

Functions

- void `_idle_thread` (void *p)
This function implements the idle thread infinite loop.

Kernel version

- #define `CH_KERNEL_MAJOR` 2
Kernel version major number.
- #define `CH_KERNEL_MINOR` 4
Kernel version minor number.

- `#define CH_KERNEL_PATCH 0`

Kernel version patch number.

Defines

- `#define _CHIBIOS_RT_`

ChibiOS/RT identification macro.

- `#define CH_KERNEL_VERSION "2.4.0"`

Kernel version string.

7.2.2 Function Documentation

7.2.2.1 `void _idle_thread(void * p)`

This function implements the idle thread infinite loop.

The function puts the processor in the lowest power mode capable to serve interrupts.

The priority is internally set to the minimum system value so that this thread is executed only if there are no other ready threads in the system.

Parameters

in

p the thread parameter, unused in this scenario

7.2.3 Define Documentation

7.2.3.1 `#define _CHIBIOS_RT_`

ChibiOS/RT identification macro.

7.2.3.2 `#define CH_KERNEL_VERSION "2.4.0"`

Kernel version string.

7.2.3.3 `#define CH_KERNEL_MAJOR 2`

Kernel version major number.

7.2.3.4 `#define CH_KERNEL_MINOR 4`

Kernel version minor number.

7.2.3.5 `#define CH_KERNEL_PATCH 0`

Kernel version patch number.

7.3 Configuration

7.3.1 Detailed Description

Kernel related settings and hooks.

Kernel parameters and options

- #define CH_FREQUENCY 1000
System tick frequency.
- #define CH_TIME_QUANTUM 20
Round robin interval.
- #define CH_MEMCORE_SIZE 0
Managed RAM size.
- #define CH_NO_IDLE_THREAD FALSE
Idle thread automatic spawn suppression.

Performance options

- #define CH_OPTIMIZE_SPEED TRUE
OS optimization.

Subsystem options

- #define CH_USE_REGISTRY TRUE
Threads registry APIs.
- #define CH_USE_WAITEXIT TRUE
Threads synchronization APIs.
- #define CH_USE_SEMAPHORES TRUE
Semaphores APIs.
- #define CH_USE_SEMAPHORES_PRIORITY FALSE
Semaphores queuing mode.
- #define CH_USE_SEMSW TRUE
Atomic semaphore API.
- #define CH_USE_MUTEXES TRUE
Mutexes APIs.
- #define CH_USE_CONDVARSL TRUE
Conditional Variables APIs.
- #define CH_USE_CONDVARSL_TIMEOUT TRUE
Conditional Variables APIs with timeout.
- #define CH_USE_EVENTS TRUE
Events Flags APIs.
- #define CH_USE_EVENTS_TIMEOUT TRUE
Events Flags APIs with timeout.
- #define CH_USE_MESSAGES TRUE
Synchronous Messages APIs.
- #define CH_USE_MESSAGES_PRIORITY FALSE
Synchronous Messages queuing mode.
- #define CH_USE_MAILBOXES TRUE
Mailboxes APIs.
- #define CH_USE_QUEUES TRUE
I/O Queues APIs.
- #define CH_USE_MEMCORE TRUE
Core Memory Manager APIs.
- #define CH_USE_HEAP TRUE
Heap Allocator APIs.
- #define CH_USE_MALLOC_HEAP FALSE

- `#define CH_USE_MEMPOOLS TRUE`
Memory Pools Allocator APIs.
- `#define CH_USE_DYNAMIC TRUE`
Dynamic Threads APIs.

Debug options

- `#define CH_DBG_SYSTEM_STATE_CHECK FALSE`
Debug option, system state check.
- `#define CH_DBG_ENABLE_CHECKS FALSE`
Debug option, parameters checks.
- `#define CH_DBG_ENABLE_ASSERTS FALSE`
Debug option, consistency checks.
- `#define CH_DBG_ENABLE_TRACE FALSE`
Debug option, trace buffer.
- `#define CH_DBG_ENABLE_STACK_CHECK FALSE`
Debug option, stack checks.
- `#define CH_DBG_FILL_THREADS FALSE`
Debug option, stacks initialization.
- `#define CH_DBG_THREADS_PROFILING TRUE`
Debug option, threads profiling.

Kernel hooks

- `#define THREAD_EXT_FIELDS`
Threads descriptor structure extension.
- `#define THREAD_EXT_INIT_HOOK(tp)`
Threads initialization hook.
- `#define THREAD_EXT_EXIT_HOOK(tp)`
Threads finalization hook.
- `#define THREAD_CONTEXT_SWITCH_HOOK(ntp, otp)`
Context switch hook.
- `#define IDLE_LOOP_HOOK()`
Idle Loop hook.
- `#define SYSTEM_TICK_EVENT_HOOK()`
System tick event hook.
- `#define SYSTEM_HALT_HOOK()`
System halt hook.

7.3.2 Define Documentation

7.3.2.1 `#define CH_FREQUENCY 1000`

System tick frequency.

Frequency of the system timer that drives the system ticks. This setting also defines the system tick time unit.

7.3.2.2 #define CH_TIME_QUANTUM 20

Round robin interval.

This constant is the number of system ticks allowed for the threads before preemption occurs. Setting this value to zero disables the preemption for threads with equal priority and the round robin becomes cooperative. Note that higher priority threads can still preempt, the kernel is always preemptive.

Note

Disabling the round robin preemption makes the kernel more compact and generally faster.

7.3.2.3 #define CH_MEMCORE_SIZE 0

Managed RAM size.

Size of the RAM area to be managed by the OS. If set to zero then the whole available RAM is used. The core memory is made available to the heap allocator and/or can be used directly through the simplified core memory allocator.

Note

In order to let the OS manage the whole RAM the linker script must provide the `__heap_base__` and `__heap_end__` symbols.

Requires `CH_USE_MEMCORE`.

7.3.2.4 #define CH_NO_IDLE_THREAD FALSE

Idle thread automatic spawn suppression.

When this option is activated the function `chSysInit()` does not spawn the idle thread automatically. The application has then the responsibility to do one of the following:

- Spawn a custom idle thread at priority IDLEPRIO.
- Change the `main()` thread priority to IDLEPRIO then enter an endless loop. In this scenario the `main()` thread acts as the idle thread.

Note

Unless an idle thread is spawned the `main()` thread must not enter a sleep state.

7.3.2.5 #define CH_OPTIMIZE_SPEED TRUE

OS optimization.

If enabled then time efficient rather than space efficient code is used when two possible implementations exist.

Note

This is not related to the compiler optimization options.

The default is `TRUE`.

7.3.2.6 #define CH_USE_REGISTRY TRUE

Threads registry APIs.

If enabled then the registry APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.7 #define CH_USE_WAITEXIT TRUE

Threads synchronization APIs.

If enabled then the [chThdWait\(\)](#) function is included in the kernel.

Note

The default is TRUE.

7.3.2.8 #define CH_USE_SEMAPHORES TRUE

Semaphores APIs.

If enabled then the Semaphores APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.9 #define CH_USE_SEMAPHORES_PRIORITY FALSE

Semaphores queuing mode.

If enabled then the threads are enqueued on semaphores by priority rather than in FIFO order.

Note

The default is FALSE. Enable this if you have special requirements.

Requires CH_USE_SEMAPHORES.

7.3.2.10 #define CH_USE_SEMSW TRUE

Atomic semaphore API.

If enabled then the semaphores the [chSemSignalWait\(\)](#) API is included in the kernel.

Note

The default is TRUE.

Requires CH_USE_SEMAPHORES.

7.3.2.11 #define CH_USE_MUTEXES TRUE

Mutexes APIs.

If enabled then the mutexes APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.12 #define CH_USE_CONDVAR TRUE

Conditional Variables APIs.

If enabled then the conditional variables APIs are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_MUTEXES.

7.3.2.13 #define CH_USE_CONDVAR_TIMEOUT TRUE

Conditional Variables APIs with timeout.

If enabled then the conditional variables APIs with timeout specification are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_CONDVAR.

7.3.2.14 #define CH_USE_EVENTS TRUE

Events Flags APIs.

If enabled then the event flags APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.15 #define CH_USE_EVENTS_TIMEOUT TRUE

Events Flags APIs with timeout.

If enabled then the events APIs with timeout specification are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_EVENTS.

7.3.2.16 #define CH_USE_MESSAGES TRUE

Synchronous Messages APIs.

If enabled then the synchronous messages APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.17 #define CH_USE_MESSAGES_PRIORITY FALSE

Synchronous Messages queuing mode.

If enabled then messages are served by priority rather than in FIFO order.

Note

The default is FALSE. Enable this if you have special requirements.

Requires CH_USE_MESSAGES.

7.3.2.18 #define CH_USE_MAILBOXES TRUE

Mailboxes APIs.

If enabled then the asynchronous messages (mailboxes) APIs are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_SEMAPHORES.

7.3.2.19 #define CH_USE_QUEUES TRUE

I/O Queues APIs.

If enabled then the I/O queues APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.20 #define CH_USE_MEMCORE TRUE

Core Memory Manager APIs.

If enabled then the core memory manager APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.21 #define CH_USE_HEAP TRUE

Heap Allocator APIs.

If enabled then the memory heap allocator APIs are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_MEMCORE and either CH_USE_MUTEXES or CH_USE_SEMAPHORES.

Mutexes are recommended.

7.3.2.22 #define CH_USE_MALLOC_HEAP FALSE

C-runtime allocator.

If enabled the the heap allocator APIs just wrap the C-runtime `malloc()` and `free()` functions.

Note

The default is FALSE.

Requires CH_USE_HEAP.

The C-runtime may or may not require CH_USE_MEMCORE, see the appropriate documentation.

7.3.2.23 #define CH_USE_MEMPOOLS TRUE

Memory Pools Allocator APIs.

If enabled then the memory pools allocator APIs are included in the kernel.

Note

The default is TRUE.

7.3.2.24 #define CH_USE_DYNAMIC TRUE

Dynamic Threads APIs.

If enabled then the dynamic threads creation APIs are included in the kernel.

Note

The default is TRUE.

Requires CH_USE_WAITEXIT.

Requires CH_USE_HEAP and/or CH_USE_MEMPOOLS.

7.3.2.25 #define CH_DBG_SYSTEM_STATE_CHECK FALSE

Debug option, system state check.

If enabled the correct call protocol for system APIs is checked at runtime.

Note

The default is FALSE.

7.3.2.26 #define CH_DBG_ENABLE_CHECKS FALSE

Debug option, parameters checks.

If enabled then the checks on the API functions input parameters are activated.

Note

The default is FALSE.

7.3.2.27 #define CH_DBG_ENABLE_ASSERTS FALSE

Debug option, consistency checks.

If enabled then all the assertions in the kernel code are activated. This includes consistency checks inside the kernel, runtime anomalies and port-defined checks.

Note

The default is FALSE.

7.3.2.28 #define CH_DBG_ENABLE_TRACE FALSE

Debug option, trace buffer.

If enabled then the context switch circular trace buffer is activated.

Note

The default is FALSE.

7.3.2.29 #define CH_DBG_ENABLE_STACK_CHECK FALSE

Debug option, stack checks.

If enabled then a runtime stack check is performed.

Note

The default is FALSE.

The stack check is performed in a architecture/port dependent way. It may not be implemented or some ports.

The default failure mode is to halt the system with the global `panic_msg` variable set to NULL.

7.3.2.30 #define CH_DBG_FILL_THREADS FALSE

Debug option, stacks initialization.

If enabled then the threads working area is filled with a byte value when a thread is created. This can be useful for the runtime measurement of the used stack.

Note

The default is FALSE.

7.3.2.31 #define CH_DBG_THREADS_PROFILING TRUE

Debug option, threads profiling.

If enabled then a field is added to the [Thread](#) structure that counts the system ticks occurred while executing the thread.

Note

The default is TRUE.

This debug option is defaulted to TRUE because it is required by some test cases into the test suite.

7.3.2.32 #define THREAD_EXT_FIELDS

Threads descriptor structure extension.

User fields added to the end of the [Thread](#) structure.

7.3.2.33 #define THREAD_EXT_INIT_HOOK(tp)**Value:**

```
{                                     \
    /* Add threads initialization code here. */ \
}
```

Threads initialization hook.

User initialization code added to the [chThdInit\(\)](#) API.

Note

It is invoked from within [chThdInit\(\)](#) and implicitly from all the threads creation APIs.

7.3.2.34 #define THREAD_EXT_EXIT_HOOK(tp)**Value:**

```
{                                     \
    /* Add threads finalization code here. */ \
}
```

Threads finalization hook.

User finalization code added to the [chThdExit\(\)](#) API.

Note

It is inserted into lock zone.

It is also invoked when the threads simply return in order to terminate.

7.3.2.35 #define THREAD_CONTEXT_SWITCH_HOOK(ntp, otp)**Value:**

```
{                                     \
    /* System halt code here. */ \
}
```

Context switch hook.

This hook is invoked just before switching between threads.

7.3.2.36 #define IDLE_LOOP_HOOK()**Value:**

```
{                                     \
    /* Idle loop code here. */ \
}
```

Idle Loop hook.

This hook is continuously invoked by the idle thread loop.

7.3.2.37 #define SYSTEM_TICK_EVENT_HOOK()

Value:

```
{                                     \
    /* System tick event code here.*\
}                                     \
```

System tick event hook.

This hook is invoked in the system tick handler immediately after processing the virtual timers queue.

7.3.2.38 #define SYSTEM_HALT_HOOK()

Value:

```
{                                     \
    /* System halt code here.*\
}                                     \
```

System halt hook.

This hook is invoked in case to a system halting error before the system is halted.

7.4 Types

The system types are defined into the port layer, please refer to the core port implementation section.

7.5 Base Kernel Services

7.5.1 Detailed Description

Base kernel services, the base subsystems are always included in the OS builds.

Modules

- [System Management](#)
- [Scheduler](#)
- [Threads](#)
- [Time and Virtual Timers](#)

7.6 System Management

7.6.1 Detailed Description

System related APIs and services:

- Initialization.
- Locks.
- Interrupt Handling.
- Power Management.
- Abnormal Termination.

Functions

- void `chSysInit` (void)
ChibiOS/RT initialization.
- void `chSysTimerHandler1` (void)
Handles time ticks for round robin preemption and timer increments.
- void `_idle_thread` (void *p)
This function implements the idle thread infinite loop.

Macro Functions

- #define `chSysGetIdleThread()` ((`Thread` *)`_idle_thread_wa`)
Returns a pointer to the idle thread.
- #define `chSysHalt()` `port_halt()`
Halts the system.
- #define `chSysSwitch(ntp, otp)`
Performs a context switch.
- #define `chSysDisable()`
Raises the system interrupt priority mask to the maximum level.
- #define `chSysSuspend()`
Raises the system interrupt priority mask to system level.
- #define `chSysEnable()`
Lowers the system interrupt priority mask to user level.
- #define `chSysLock()`
Enters the kernel lock mode.
- #define `chSysUnlock()`
Leaves the kernel lock mode.
- #define `chSysLockFromIsr()`
Enters the kernel lock mode from within an interrupt handler.
- #define `chSysUnlockFromIsr()`
Leaves the kernel lock mode from within an interrupt handler.

ISRs abstraction macros

- #define `CH_IRQ_PROLOGUE()`
IRQ handler enter code.
- #define `CH_IRQ_EPILOGUE()`
IRQ handler exit code.
- #define `CH_IRQ_HANDLER(id)` `PORT_IRQ_HANDLER(id)`
Standard normal IRQ handler declaration.

Fast ISRs abstraction macros

- #define `CH_FAST_IRQ_HANDLER(id)` `PORT_FAST_IRQ_HANDLER(id)`
Standard fast IRQ handler declaration.

7.6.2 Function Documentation

7.6.2.1 void chSysInit (void)

ChibiOS/RT initialization.

After executing this function the current instructions stream becomes the main thread.

Precondition

Interrupts must be still disabled when `chSysInit()` is invoked and are internally enabled.

Postcondition

The main thread is created with priority NORMALPRIO.

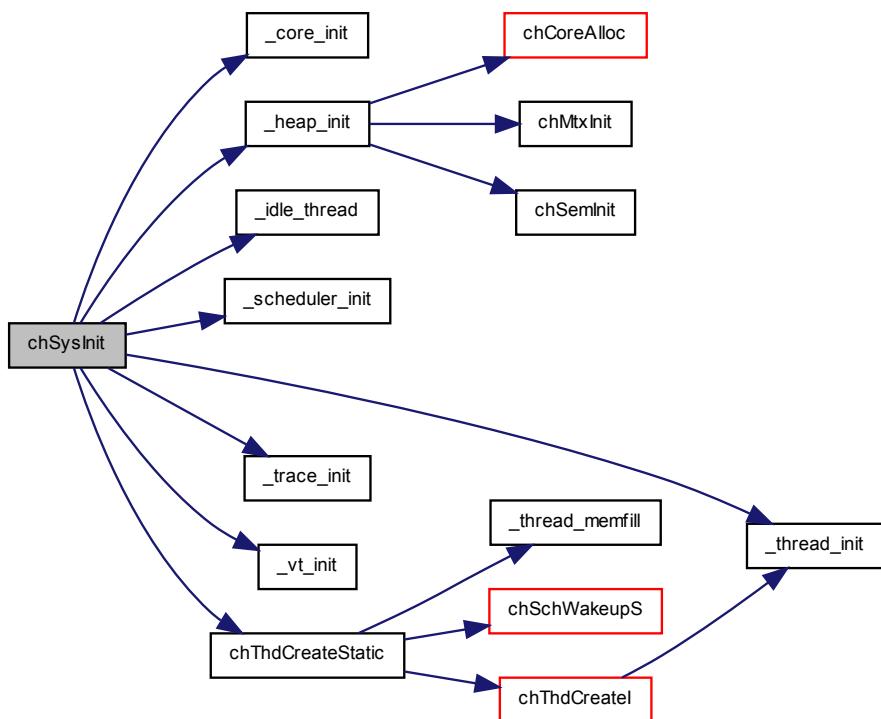
Note

This function has special, architecture-dependent, requirements, see the notes into the various port reference manuals.

Function Class:

Special function, this function has special requirements see the notes.

Here is the call graph for this function:



7.6.2.2 void chSysTimerHandler (void)

Handles time ticks for round robin preemption and timer increments.

Decrements the remaining time quantum of the running thread and preempts it when the quantum is used up.
Increments system time and manages the timers.

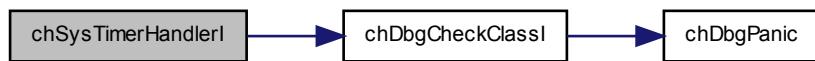
Note

The frequency of the timer determines the system tick granularity and, together with the CH_TIME_QUANTUM macro, the round robin interval.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.6.2.3 void _idle_thread(void * p)

This function implements the idle thread infinite loop.

The function puts the processor in the lowest power mode capable to serve interrupts.

The priority is internally set to the minimum system value so that this thread is executed only if there are no other ready threads in the system.

Parameters

in p the thread parameter, unused in this scenario

7.6.3 Define Documentation

7.6.3.1 #define chSysGetIdleThread()((Thread *)_idle_thread_wa)

Returns a pointer to the idle thread.

Precondition

In order to use this function the option CH_NO_IDLE_THREAD must be disabled.

Note

The reference counter of the idle thread is not incremented but it is not strictly required being the idle thread a static object.

Returns

Pointer to the idle thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.6.3.2 #define chSysHalt() port_halt()

Halts the system.

This function is invoked by the operating system when an unrecoverable error is detected, for example because a programming error in the application code that triggers an assertion while in debug mode.

Note

Can be invoked from any system state.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.3 #define chSysSwitch(ntp, otp)

Value:

```
{
    dbg_trace(otp);
    THREAD_CONTEXT_SWITCH_HOOK(ntp, otp);
    port_switch(ntp, otp);
}
```

Performs a context switch.

Note

Not a user function, it is meant to be invoked by the scheduler itself or from within the port layer.

Parameters

in	<i>ntp</i>	the thread to be switched in
in	<i>otp</i>	the thread to be switched out

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.4 #define chSysDisable()

Value:

```
{
    port_disable();
    dbg_check_disable();
}
```

Raises the system interrupt priority mask to the maximum level.

All the maskable interrupt sources are disabled regardless their hardware priority.

Note

Do not invoke this API from within a kernel lock.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.5 #define chSysSuspend()

Value:

```
{
    port_suspend();
    \dbg_check_suspend();
}
```

Raises the system interrupt priority mask to system level.

The interrupt sources that should not be able to preempt the kernel are disabled, interrupt sources with higher priority are still enabled.

Note

Do not invoke this API from within a kernel lock.

This API is no replacement for [chSysLock\(\)](#), the [chSysLock\(\)](#) could do more than just disable the interrupts.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.6 #define chSysEnable()

Value:

```
{
    \dbg_check_enable();
    port_enable();
}
```

Lowers the system interrupt priority mask to user level.

All the interrupt sources are enabled.

Note

Do not invoke this API from within a kernel lock.

This API is no replacement for [chSysUnlock\(\)](#), the [chSysUnlock\(\)](#) could do more than just enable the interrupts.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.7 #define chSysLock()

Value:

```
{
    port_lock();
    \dbg_check_lock();
}
```

Enters the kernel lock mode.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.8 #define chSysUnlock()

Value:

```
{
    \           \
    dbg_check_unlock();
    port_unlock();
}
```

Leaves the kernel lock mode.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.9 #define chSysLockFromIsr()

Value:

```
{
    \           \
    port_lock_from_isr();
    \           \
    dbg_check_lock_from_isr();
}
```

Enters the kernel lock mode from within an interrupt handler.

Note

This API may do nothing on some architectures, it is required because on ports that support preemptable interrupt handlers it is required to raise the interrupt mask to the same level of the system mutual exclusion zone.

It is good practice to invoke this API before invoking any I-class syscall from an interrupt handler.

This API must be invoked exclusively from interrupt handlers.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.10 #define chSysUnlockFromIsr()

Value:

```
{
    \           \
    dbg_check_unlock_from_isr();
    port_unlock_from_isr();
}
```

Leaves the kernel lock mode from within an interrupt handler.

Note

This API may do nothing on some architectures, it is required because on ports that support preemptable interrupt handlers it is required to raise the interrupt mask to the same level of the system mutual exclusion zone.

It is good practice to invoke this API after invoking any I-class syscall from an interrupt handler.

This API must be invoked exclusively from interrupt handlers.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.11 #define CH_IRQ_PROLOGUE()**Value:**

```
PORT_IRQ_PROLOGUE();  
    \  
    dbg_check_enter_isr();
```

IRQ handler enter code.

Note

Usually IRQ handlers functions are also declared naked.
On some architectures this macro can be empty.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.12 #define CH_IRQ_EPILOGUE()**Value:**

```
dbg_check_leave_isr();  
    \  
    PORT_IRQ_EPILOGUE();
```

IRQ handler exit code.

Note

Usually IRQ handlers function are also declared naked.
This macro usually performs the final reschedule by using [chSchIsPreemptionRequired\(\)](#) and [chSchDoReschedule\(\)](#).

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.13 #define CH_IRQ_HANDLER(id) PORT_IRQ_HANDLER(id)

Standard normal IRQ handler declaration.

Note

id can be a function name or a vector number depending on the port implementation.

Function Class:

Special function, this function has special requirements see the notes.

7.6.3.14 #define CH_FAST_IRQ_HANDLER(id) PORT_FAST_IRQ_HANDLER(id)

Standard fast IRQ handler declaration.

Note

id can be a function name or a vector number depending on the port implementation.
Not all architectures support fast interrupts.

Function Class:

Special function, this function has special requirements see the notes.

7.7 Scheduler

7.7.1 Detailed Description

This module provides the default portable scheduler code, scheduler functions can be individually captured by the port layer in order to provide architecture optimized equivalents. When a function is captured its default code is not built into the OS image, the optimized version is included instead.

Data Structures

- struct [ReadyList](#)

Ready list header.

Functions

- void [_scheduler_init](#) (void)
Scheduler initialization.
- Thread * [chSchReady](#) (Thread *tp)
Inserts a thread in the Ready List.
- void [chSchGoSleepS](#) (tstate_t newstate)
Puts the current thread to sleep into the specified state.
- msg_t [chSchGoSleepTimeoutS](#) (tstate_t newstate, systime_t time)
Puts the current thread to sleep into the specified state with timeout specification.
- void [chSchWakeupS](#) (Thread *ntp, msg_t msg)
Wakes up a thread.
- void [chSchRescheduleS](#) (void)
Performs a reschedule if a higher priority thread is runnable.
- bool_t [chSchIsPreemptionRequired](#) (void)
Evaluates if preemption is required.
- void [chSchDoReschedule](#) (void)
Switches to the first thread on the runnable queue.

Variables

- ReadyList rlist

Ready list header.

Wakeup status codes

- #define [RDY_OK](#) 0
Normal wakeup message.
- #define [RDY_TIMEOUT](#) -1
Wakeup caused by a timeout condition.
- #define [RDY_RESET](#) -2
Wakeup caused by a reset condition.

Priority constants

- `#define NOPRIO 0`
Ready list header priority.
- `#define IDLEPRIO 1`
Idle thread priority.
- `#define LOWPRIO 2`
Lowest user priority.
- `#define NORMALPRIO 64`
Normal user priority.
- `#define HIGHPRIO 127`
Highest user priority.
- `#define ABSPRIO 255`
Greatest possible priority.

Special time constants

- `#define TIME_IMMEDIATE ((systime_t)0)`
Zero time specification for some functions with a timeout specification.
- `#define TIME_INFINITE ((systime_t)-1)`
Infinite time specification for all functions with a timeout specification.

Macro Functions

- `#define chSchIsRescRequired()` (`firstprio(&rlist.r_queue) > currp->p_prio`)
Determines if the current thread must reschedule.
- `#define chSchCanYieldS()` (`firstprio(&rlist.r_queue) >= currp->p_prio`)
Determines if yielding is possible.
- `#define chSchDoYieldS()`
Yields the time slot.
- `#define chSchPreemption()`
Inlineable preemption code.

Defines

- `#define firstprio(rlp) ((rlp)->p_next->p_prio)`
Returns the priority of the first thread on the given ready list.
- `#define currp rlist.r_current`
Current thread pointer access macro.
- `#define setcurrp(tp) (currp = (tp))`
Current thread pointer change macro.

7.7.2 Function Documentation

7.7.2.1 void _scheduler_init (void)

Scheduler initialization.

Function Class:

Not an API, this function is for internal use only.

7.7.2.2 Thread * chSchReady(Thread * tp)

Inserts a thread in the Ready List.

Precondition

The thread must not be already inserted in any list through its `p_next` and `p_prev` or list corruption would occur.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

Parameters

in `tp` the thread to be made ready

Returns

The thread pointer.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.7.2.3 void chSchGoSleepS(tstate_t newstate)

Puts the current thread to sleep into the specified state.

The thread goes into a sleeping state. The possible [Thread States](#) are defined into `threads.h`.

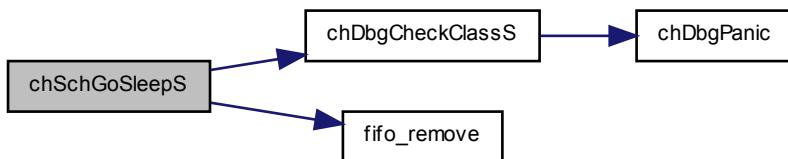
Parameters

in `newstate` the new thread state

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.7.2.4 `msg_t chSchGoSleepTimeoutS(tstate_t newstate, systime_t time)`

Puts the current thread to sleep into the specified state with timeout specification.

The thread goes into a sleeping state, if it is not awakened explicitly within the specified timeout then it is forcibly awakened with a `RDY_TIMEOUT` low level message. The possible [Thread States](#) are defined into `threads.h`.

Parameters

in	<code>newstate</code>	the new thread state
in	<code>time</code>	the number of ticks before the operation timeouts, the special values are handled as follow:
<ul style="list-style-type: none"> • <code>TIME_INFINITE</code> the thread enters an infinite sleep state, this is equivalent to invoking chSchGoSleepS() but, of course, less efficient. • <code>TIME_IMMEDIATE</code> this value is not allowed. 		

Returns

The wakeup message.

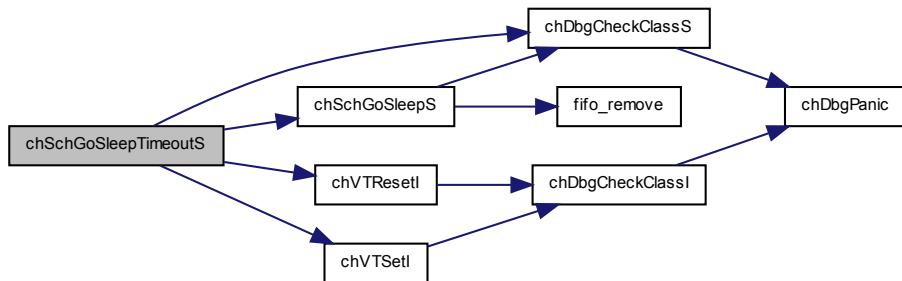
Return values

`RDY_TIMEOUT` if a timeout occurs.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.7.2.5 `void chSchWakeupS(Thread * ntp, msg_t msg)`

Wakes up a thread.

The thread is inserted into the ready list or immediately made running depending on its relative priority compared to the current thread.

Precondition

The thread must not be already inserted in any list through its `p_next` and `p_prev` or list corruption would occur.

Note

It is equivalent to a [chSchReadyI\(\)](#) followed by a [chSchRescheduleS\(\)](#) but much more efficient.

The function assumes that the current thread has the highest priority.

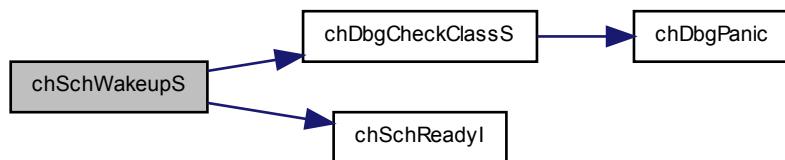
Parameters

in	<i>ntp</i>	the Thread to be made ready
in	<i>msg</i>	message to the awakened thread

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.7.2.6 void chSchRescheduleS (void)

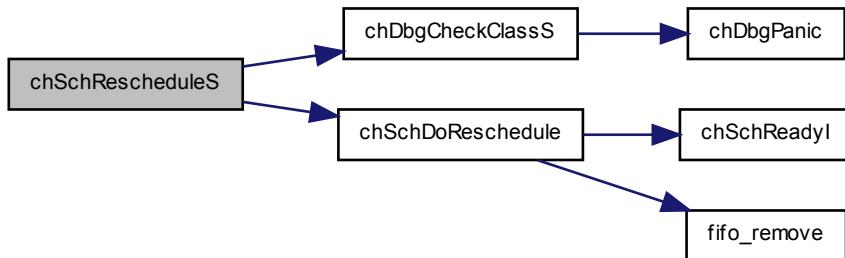
Performs a reschedule if a higher priority thread is runnable.

If a thread with a higher priority than the current thread is in the ready list then make the higher priority thread running.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.7.2.7 `bool_t chSchIsPreemptionRequired(void)`

Evaluates if preemption is required.

The decision is taken by comparing the relative priorities and depending on the state of the round robin timeout counter.

Note

Not a user function, it is meant to be invoked by the scheduler itself or from within the port layer.

Return values

TRUE if there is a thread that must go in running state immediately.

FALSE if preemption is not required.

Function Class:

Special function, this function has special requirements see the notes.

7.7.2.8 `void chSchDoReschedule(void)`

Switches to the first thread on the runnable queue.

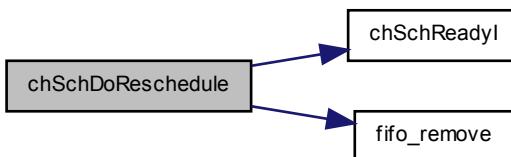
Note

Not a user function, it is meant to be invoked by the scheduler itself or from within the port layer.

Function Class:

Special function, this function has special requirements see the notes.

Here is the call graph for this function:



7.7.3 Variable Documentation

7.7.3.1 ReadyList rlist

Ready list header.

7.7.4 Define Documentation

7.7.4.1 #define RDY_OK 0

Normal wakeup message.

7.7.4.2 #define RDY_TIMEOUT -1

Wakeup caused by a timeout condition.

7.7.4.3 #define RDY_RESET -2

Wakeup caused by a reset condition.

7.7.4.4 #define NOPRIO 0

Ready list header priority.

7.7.4.5 #define IDLEPRIO 1

Idle thread priority.

7.7.4.6 #define LOWPRIO 2

Lowest user priority.

7.7.4.7 #define NORMALPRIO 64

Normal user priority.

7.7.4.8 #define HIGHPRIO 127

Highest user priority.

7.7.4.9 #define ABSPRIO 255

Greatest possible priority.

7.7.4.10 #define TIME_IMMEDIATE ((systime_t)0)

Zero time specification for some functions with a timeout specification.

Note

Not all functions accept TIME_IMMEDIATE as timeout parameter, see the specific function documentation.

7.7.4.11 #define TIME_INFINITE ((systime_t)-1)

Infinite time specification for all functions with a timeout specification.

7.7.4.12 #define firstprio(rlp) ((rlp)->p_next->p_prio)

Returns the priority of the first thread on the given ready list.

Function Class:

Not an API, this function is for internal use only.

7.7.4.13 #define currp rlist.r_current

Current thread pointer access macro.

Note

This macro is not meant to be used in the application code but only from within the kernel, use the `chThdSelf()` API instead.

It is forbidden to use this macro in order to change the pointer (`currp = something`), use `setcurrp()` instead.

7.7.4.14 #define setcurrp(tp) (currp = (tp))

Current thread pointer change macro.

Note

This macro is not meant to be used in the application code but only from within the kernel.

Function Class:

Not an API, this function is for internal use only.

7.7.4.15 #define chSchIsRescRequired() (firstprio(&rlist.r_queue) > currp->p_prio)

Determines if the current thread must reschedule.

This function returns `TRUE` if there is a ready thread with higher priority.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.7.4.16 #define chSchCanYieldS() (firstprio(&rlist.r_queue) >= currp->p_prio)

Determines if yielding is possible.

This function returns `TRUE` if there is a ready thread with equal or higher priority.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.7.4.17 #define chSchDoYieldS()**Value:**

```
{           \
    if (chSchCanYieldS()) \
        chSchDoReschedule(); \
}
```

Yields the time slot.

Yields the CPU control to the next thread in the ready list with equal or higher priority, if any.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.7.4.18 #define chSchPreemption()

Value:

```
{
    tprio_t p1 = firstprio(&rlist.r_queue);
    tprio_t p2 = currp->p_prio;
    if (rlist.r_preempt) {
        if (p1 > p2)
            chSchDoReschedule();
    }
    else {
        if (p1 >= p2)
            chSchDoReschedule();
    }
}
```

Inlineable preemption code.

This is the common preemption code, this function must be invoked exclusively from the port layer.

Function Class:

Special function, this function has special requirements see the notes.

7.8 Threads

7.8.1 Detailed Description

Threads related APIs and services.

Operation mode

A thread is an abstraction of an independent instructions flow. In ChibiOS/RT a thread is represented by a "C" function owning a processor context, state informations and a dedicated stack area. In this scenario static variables are shared among all threads while automatic variables are local to the thread.

Operations defined for threads:

- **Create**, a thread is started on the specified thread function. This operation is available in multiple variants, both static and dynamic.
- **Exit**, a thread terminates by returning from its top level function or invoking a specific API, the thread can return a value that can be retrieved by other threads.
- **Wait**, a thread waits for the termination of another thread and retrieves its return value.
- **Resume**, a thread created in suspended state is started.
- **Sleep**, the execution of a thread is suspended for the specified amount of time or the specified future absolute time is reached.
- **SetPriority**, a thread changes its own priority level.
- **Yield**, a thread voluntarily renounces to its time slot.

The threads subsystem is implicitly included in kernel however some of its part may be excluded by disabling them in `chconf.h`, see the `CH_USE_WAITEXIT` and `CH_USE_DYNAMIC` configuration options.

Data Structures

- struct **Thread**
Structure representing a thread.

Functions

- `Thread * _thread_init (Thread *tp, tprio_t prio)`
Initializes a thread structure.
- `void _thread_memfill (uint8_t *startp, uint8_t *endp, uint8_t v)`
Memory fill utility.
- `Thread * chThdCreateL (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `Thread * chThdCreateStatic (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `tprio_t chThdSetPriority (tprio_t newprio)`
Changes the running thread priority level then reschedules if necessary.
- `Thread * chThdResume (Thread *tp)`
Resumes a suspended thread.
- `void chThdTerminate (Thread *tp)`
Requests a thread termination.
- `void chThdSleep (systime_t time)`
Suspends the invoking thread for the specified time.
- `void chThdSleepUntil (systime_t time)`
Suspends the invoking thread until the system time arrives to the specified value.
- `void chThdYield (void)`
Yields the time slot.
- `void chThdExit (msg_t msg)`
Terminates the current thread.
- `void chThdExitS (msg_t msg)`
Terminates the current thread.
- `msg_t chThdWait (Thread *tp)`
Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.

Thread states

- `#define THD_STATE_READY 0`
Waiting on the ready list.
- `#define THD_STATE_CURRENT 1`
Currently running.
- `#define THD_STATE_SUSPENDED 2`
Created in suspended state.
- `#define THD_STATE_WTSEM 3`
Waiting on a semaphore.
- `#define THD_STATE_WTMTX 4`
Waiting on a mutex.
- `#define THD_STATE_WTCOND 5`
Waiting on a condition variable.
- `#define THD_STATE_SLEEPING 6`
Waiting in `chThdSleep()` or `chThdSleepUntil()`.
- `#define THD_STATE_WTEXIT 7`
Waiting in `chThdWait()`.
- `#define THD_STATE_WTOREVT 8`
Waiting for an event.
- `#define THD_STATE_WTANDEVT 9`
Waiting for several events.

- #define THD_STATE SNDMSGQ 10
Sending a message, in queue.
- #define THD_STATE SNDMSG 11
Sent a message, waiting answer.
- #define THD_STATE_WTMSG 12
Waiting for a message.
- #define THD_STATE_WTQUEUE 13
Waiting on an I/O queue.
- #define THD_STATE_FINAL 14
Thread terminated.
- #define THD_STATE_NAMES
Thread states as array of strings.

Thread flags and attributes

- #define THD_MODE_MASK 3
Thread memory mode mask.
- #define THD_MODE_STATIC 0
Static thread.
- #define THD_MODE_HEAP 1
Thread allocated from a Memory Heap.
- #define THD_MODE_MEMPOOL 2
Thread allocated from a Memory Pool.
- #define THD_TERMINATE 4
Termination requested flag.

Macro Functions

- #define chThdSelf() currp
Returns a pointer to the current Thread.
- #define chThdGetPriority() (currp->p_prio)
Returns the current thread priority.
- #define chThdGetTicks(tp) ((tp)->p_time)
Returns the number of ticks consumed by the specified thread.
- #define chThdLS() (void *) (currp + 1)
Returns the pointer to the Thread local storage area, if any.
- #define chThdTerminated(tp) ((tp)->p_state == THD_STATE_FINAL)
Verifies if the specified thread is in the THD_STATE_FINAL state.
- #define chThdShouldTerminate() (currp->p_flags & THD_TERMINATE)
Verifies if the current thread has a termination request pending.
- #define chThdResumel(tp) chSchReadyl(tp)
Resumes a thread created with chThdCreateI().
- #define chThdSleepS(time) chSchGoSleepTimeoutS(THD_STATE_SLEEPING, time)
Suspends the invoking thread for the specified time.
- #define chThdSleepSeconds(sec) chThdSleep(S2ST(sec))
Delays the invoking thread for the specified number of seconds.
- #define chThdSleepMilliseconds(msec) chThdSleep(MS2ST(msec))
Delays the invoking thread for the specified number of milliseconds.
- #define chThdSleepMicroseconds(usec) chThdSleep(US2ST(usec))
Delays the invoking thread for the specified number of microseconds.

Typedefs

- `typedef msg_t(* tfunc_t)(void *)`

Thread function.

7.8.2 Function Documentation

7.8.2.1 Thread * _thread_init (Thread * tp, tprio_t prio)

Initializes a thread structure.

Note

This is an internal functions, do not use it in application code.

Parameters

in	<i>tp</i>	pointer to the thread
in	<i>prio</i>	the priority level for the new thread

Returns

The same thread pointer passed as parameter.

Function Class:

Not an API, this function is for internal use only.

7.8.2.2 void _thread_memfill (uint8_t * startp, uint8_t * endp, uint8_t v)

Memory fill utility.

Parameters

in	<i>startp</i>	first address to fill
in	<i>endp</i>	last address to fill +1
in	<i>v</i>	filler value

Function Class:

Not an API, this function is for internal use only.

7.8.2.3 Thread * chThdCreate (void * wsp, size_t size, tprio_t prio, tfunc_t pf, void * arg)

Creates a new thread into a static memory area.

The new thread is initialized but not inserted in the ready list, the initial state is THD_STATE_SUSPENDED.

Postcondition

The initialized thread can be subsequently started by invoking `chThdResume()`, `chThdResumeI()` or `chSchWakeupS()` depending on the execution context.

Note

A thread can terminate by calling `chThdExit()` or by simply returning from its main function.

Threads created using this function do not obey to the `CH_DBG_FILL_THREADS` debug option because it would keep the kernel locked for too much time.

Parameters

out	wsp	pointer to a working area dedicated to the thread stack
in	size	size of the working area
in	prio	the priority level for the new thread
in	pf	the thread function
in	arg	an argument passed to the thread function. It can be NULL.

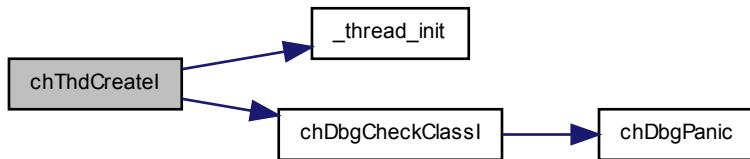
Returns

The pointer to the [Thread](#) structure allocated for the thread into the working space area.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:

**7.8.2.4 Thread * chThdCreateStatic (void * wsp, size_t size, tprio_t prio, tfunc_t pf, void * arg)**

Creates a new thread into a static memory area.

Note

A thread can terminate by calling [chThdExit \(\)](#) or by simply returning from its main function.

Parameters

out	wsp	pointer to a working area dedicated to the thread stack
in	size	size of the working area
in	prio	the priority level for the new thread
in	pf	the thread function
in	arg	an argument passed to the thread function. It can be NULL.

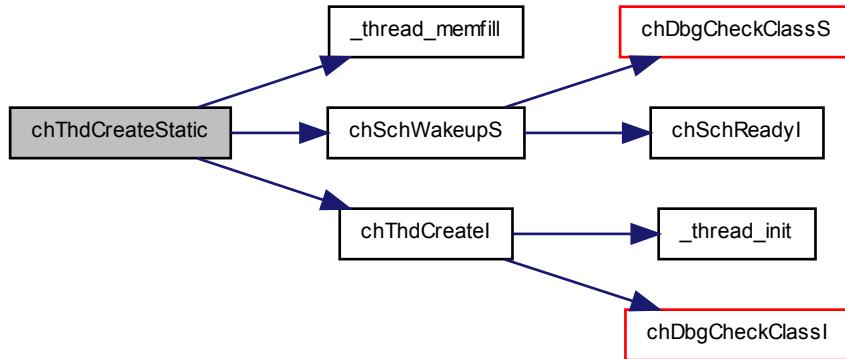
Returns

The pointer to the [Thread](#) structure allocated for the thread into the working space area.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.8.2.5 `tprio_t chThdSetPriority (tprio_t newprio)`

Changes the running thread priority level then reschedules if necessary.

Note

The function returns the real thread priority regardless of the current priority that could be higher than the real priority because the priority inheritance mechanism.

Parameters

in `newprio` the new priority level of the running thread

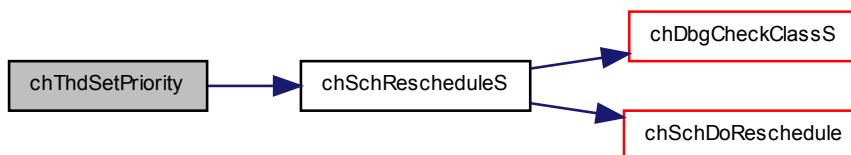
Returns

The old priority level.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.8.2.6 Thread * chThdResume (Thread * *tp*)

Resumes a suspended thread.

Precondition

The specified thread pointer must refer to an initialized thread in the THD_STATE_SUSPENDED state.

Postcondition

The specified thread is immediately started or put in the ready list depending on the relative priority levels.

Note

Use this function to start threads created with chThdInit () .

Parameters

in *tp* pointer to the thread

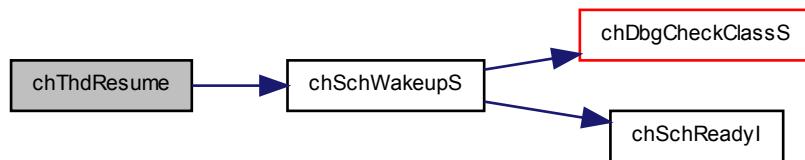
Returns

The pointer to the thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.8.2.7 void chThdTerminate (Thread * *tp*)

Requests a thread termination.

Precondition

The target thread must be written to invoke periodically [chThdShouldTerminate \(\)](#) and terminate cleanly if it returns TRUE.

Postcondition

The specified thread will terminate after detecting the termination condition.

Parameters

in *tp* pointer to the thread

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.2.8 void chThdSleep(systime_t time)

Suspends the invoking thread for the specified time.

Parameters

- in *time* the delay in system ticks, the special values are handled as follow:
- *TIME_INFINITE* the thread enters an infinite sleep state.
 - *TIME_IMMEDIATE* this value is not allowed.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.2.9 void chThdSleepUntil(systime_t time)

Suspends the invoking thread until the system time arrives to the specified value.

Parameters

- in *time* absolute system time

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.2.10 void chThdYield(void)

Yields the time slot.

Yields the CPU control to the next thread in the ready list with equal priority, if any.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.2.11 void chThdExit(msg_t msg)

Terminates the current thread.

The thread goes in the THD_STATE_FINAL state holding the specified exit status code, other threads can retrieve the exit status code by invoking the function [chThdWait\(\)](#).

Postcondition

Eventual code after this function will never be executed, this function never returns. The compiler has no way to know this so do not assume that the compiler would remove the dead code.

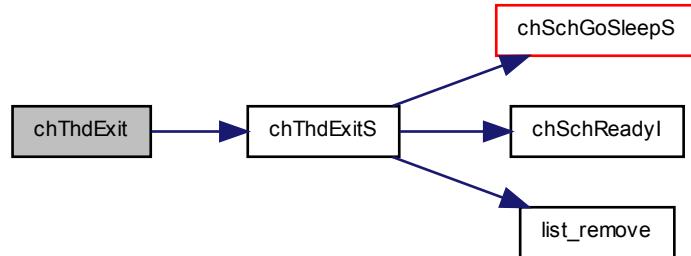
Parameters

- in *msg* thread exit code

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.8.2.12 void chThdExitS (msg_t msg)

Terminates the current thread.

The thread goes in the THD_STATE_FINAL state holding the specified exit status code, other threads can retrieve the exit status code by invoking the function [chThdWait \(\)](#).

Postcondition

Eventual code after this function will never be executed, this function never returns. The compiler has no way to know this so do not assume that the compiler would remove the dead code.

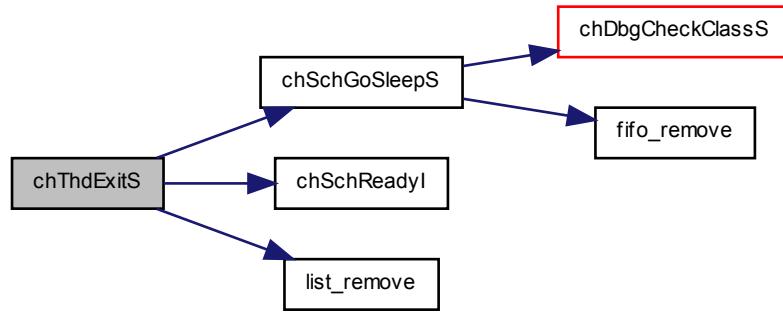
Parameters

in *msg* thread exit code

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.8.2.13 msg_t chThdWait (Thread * tp)

Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.

This function waits for the specified thread to terminate then decrements its reference counter, if the counter reaches zero then the thread working area is returned to the proper allocator.

The memory used by the exited thread is handled in different ways depending on the API that spawned the thread:

- If the thread was spawned by [chThdCreateStatic\(\)](#) or by [chThdInit\(\)](#) then nothing happens and the thread working area is not released or modified in any way. This is the default, totally static, behavior.
- If the thread was spawned by [chThdCreateFromHeap\(\)](#) then the working area is returned to the system heap.
- If the thread was spawned by [chThdCreateFromMemoryPool\(\)](#) then the working area is returned to the owning memory pool.

Precondition

The configuration option `CH_USE_WAITEXIT` must be enabled in order to use this function.

Postcondition

Enabling [chThdWait\(\)](#) requires 2-4 (depending on the architecture) extra bytes in the `Thread` structure. After invoking [chThdWait\(\)](#) the thread pointer becomes invalid and must not be used as parameter for further system calls.

Note

If `CH_USE_DYNAMIC` is not specified this function just waits for the thread termination, no memory allocators are involved.

Parameters

in *tp* pointer to the thread

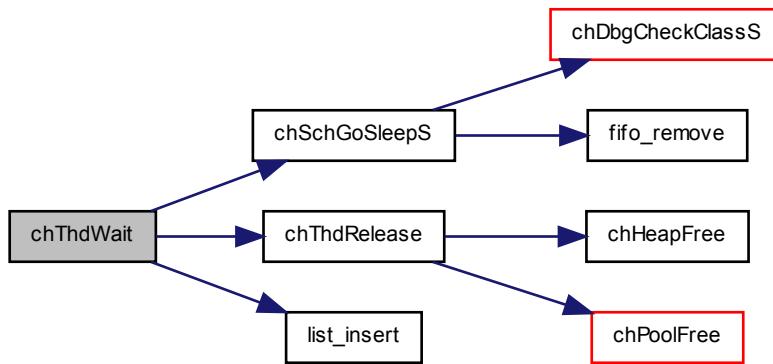
Returns

The exit code from the terminated thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.8.3 Define Documentation

7.8.3.1 #define THD_STATE_READY 0

Waiting on the ready list.

7.8.3.2 #define THD_STATE_CURRENT 1

Currently running.

7.8.3.3 #define THD_STATE_SUSPENDED 2

Created in suspended state.

7.8.3.4 #define THD_STATE_WTSEM 3

Waiting on a semaphore.

7.8.3.5 #define THD_STATE_WTMUX 4

Waiting on a mutex.

7.8.3.6 #define THD_STATE_WTCOND 5

Waiting on a condition variable.

7.8.3.7 #define THD_STATE_SLEEPING 6

Waiting in `chThdSleep()` or `chThdSleepUntil()`.

7.8.3.8 #define THD_STATE_WTEXIT 7

Waiting in `chThdWait()`.

7.8.3.9 #define THD_STATE_WTOREVT 8

Waiting for an event.

7.8.3.10 #define THD_STATE_WTANDEVT 9

Waiting for several events.

7.8.3.11 #define THD_STATE SNDMSGQ 10

Sending a message, in queue.

7.8.3.12 #define THD_STATE SNDMSG 11

Sent a message, waiting answer.

7.8.3.13 #define THD_STATE_WTMSG 12

Waiting for a message.

7.8.3.14 #define THD_STATE_WTQUEUE 13

Waiting on an I/O queue.

7.8.3.15 #define THD_STATE_FINAL 14

`Thread` terminated.

7.8.3.16 #define THD_STATE_NAMES

Value:

```
"READY", "CURRENT", "SUSPENDED", "WTSEM", "WTMTX", "WTCOND", "SLEEPING", \
"WTEXIT", "WTOREVT", "WTANDEVT", "SNDMSGQ", "SNDMSG", "WTMSG", "WTQUEUE", \
"FINAL"
```

`Thread` states as array of strings.

Each element in an array initialized with this macro can be indexed using the numeric thread state values.

7.8.3.17 #define THD_MEM_MODE_MASK 3

`Thread` memory mode mask.

7.8.3.18 #define THD_MEM_MODE_STATIC 0

Static thread.

7.8.3.19 #define THD_MEM_MODE_HEAP 1

Thread allocated from a Memory Heap.

7.8.3.20 #define THD_MEM_MODE_MEMPOOL 2

Thread allocated from a Memory Pool.

7.8.3.21 #define THD_TERMINATE 4

Termination requested flag.

7.8.3.22 #define chThdSelf() currp

Returns a pointer to the current Thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.23 #define chThdGetPriority()(currp->p_prio)

Returns the current thread priority.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.24 #define chThdGetTicks(tp)(tp)->p_time

Returns the number of ticks consumed by the specified thread.

Note

This function is only available when the CH_DBG_THREADS_PROFILING configuration option is enabled.

Parameters

in *tp* pointer to the thread

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.25 #define chThdLS()(void *)(currp + 1)

Returns the pointer to the Thread local storage area, if any.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.26 #define chThdTerminated(*tp*) ((*tp*)>p_state == THD_STATE_FINAL)

Verifies if the specified thread is in the THD_STATE_FINAL state.

Parameters

in *tp* pointer to the thread

Return values

TRUE thread terminated.

FALSE thread not terminated.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.27 #define chThdShouldTerminate() (currp->p_flags & THD_TERMINATE)

Verifies if the current thread has a termination request pending.

Return values

TRUE termination request pended.

FALSE termination request not pended.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.28 #define chThdResume(*tp*) chSchReadyI(*tp*)

Resumes a thread created with [chThdCreateI\(\)](#).

Parameters

in *tp* pointer to the thread

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.8.3.29 #define chThdSleepS(*time*) chSchGoSleepTimeoutS(THD_STATE_SLEEPING, *time*)

Suspends the invoking thread for the specified time.

Parameters

in *time* the delay in system ticks, the special values are handled as follow:

- *TIME_INFINITE* the thread enters an infinite sleep state.
- *TIME_IMMEDIATE* this value is not allowed.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.8.3.30 #define chThdSleepSeconds(*sec*) chThdSleep(S2ST(*sec*))

Delays the invoking thread for the specified number of seconds.

Note

The specified time is rounded up to a value allowed by the real system clock.
The maximum specified value is implementation dependent.

Parameters

in *sec* time in seconds, must be different from zero

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.31 #define chThdSleepMilliseconds(*msec*) chThdSleep(MS2ST(*msec*))

Delays the invoking thread for the specified number of milliseconds.

Note

The specified time is rounded up to a value allowed by the real system clock.
The maximum specified value is implementation dependent.

Parameters

in *msec* time in milliseconds, must be different from zero

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.3.32 #define chThdSleepMicroseconds(*usec*) chThdSleep(US2ST(*usec*))

Delays the invoking thread for the specified number of microseconds.

Note

The specified time is rounded up to a value allowed by the real system clock.
The maximum specified value is implementation dependent.

Parameters

in *usec* time in microseconds, must be different from zero

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.8.4 Typedef Documentation

7.8.4.1 `typedef msg_t(* tfunc_t)(void *)`

[Thread](#) function.

7.9 Time and Virtual Timers

7.9.1 Detailed Description

Time and Virtual Timers related APIs and services.

Data Structures

- struct [VirtualTimer](#)
Virtual Timer descriptor structure.
- struct [VTList](#)
Virtual timers list header.

Functions

- `void _vt_init (void)`
Virtual Timers initialization.
- `void chVTSetl (VirtualTimer *vtp, systime_t time, vfunc_t vtfunc, void *par)`
Enables a virtual timer.
- `void chVTResetl (VirtualTimer *vtp)`
Disables a Virtual Timer.
- `bool_t chTimelsWithin (systime_t start, systime_t end)`
Checks if the current system time is within the specified time window.

Variables

- [VTList vtlist](#)
Virtual timers delta list header.
- [VTList vtlist](#)
Virtual timers delta list header.

Time conversion utilities

- `#define S2ST(sec) ((systime_t)((sec) * CH_FREQUENCY))`
Seconds to system ticks.
- `#define MS2ST(msec) ((systime_t)((((msec) - 1L) * CH_FREQUENCY) / 1000L) + 1L)`
Milliseconds to system ticks.
- `#define US2ST(usec) (((systime_t)((((usec) - 1L) * CH_FREQUENCY) / 1000000L) + 1L))`
Microseconds to system ticks.

Macro Functions

- `#define chVTDoTick()`
Virtual timers ticker.
- `#define chVTIsArmed(vtp) ((vtp)->vt_func != NULL)`
Returns TRUE if the specified timer is armed.
- `#define chTimeNow() (vtlist.vt_systime)`
Current system time.

TypeDefs

- `typedef void(* vfunc_t)(void *)`
Virtual Timer callback function.
- `typedef struct VirtualTimer VirtualTimer`
Virtual Timer structure type.

7.9.2 Function Documentation

7.9.2.1 void _vt_init(void)

Virtual Timers initialization.

Note

Internal use only.

Function Class:

Not an API, this function is for internal use only.

7.9.2.2 void chVTSetl(VirtualTimer * vtp, systime_t time, vfunc_t vfunc, void * par)

Enables a virtual timer.

Note

The associated function is invoked by an interrupt handler within the I-Locked state, see [System States](#).

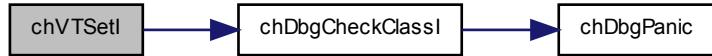
Parameters

out	<code>vtp</code>	the <code>VirtualTimer</code> structure pointer
in	<code>time</code>	the number of ticks before the operation timeouts, the special values are handled as follow: <ul style="list-style-type: none"> • <code>TIME_INFINITE</code> is allowed but interpreted as a normal time specification. • <code>TIME_IMMEDIATE</code> this value is not allowed.
in	<code>vfunc</code>	the timer callback function. After invoking the callback the timer is disabled and the structure can be disposed or reused.
in	<code>par</code>	a parameter that will be passed to the callback function

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.9.2.3 void chVTResiI (VirtualTimer * vtp)

Disables a Virtual Timer.

Note

The timer MUST be active when this function is invoked.

Parameters

in	<i>vtp</i> the <code>VirtualTimer</code> structure pointer
----	--

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.9.2.4 bool_t chTimelsWithin (systime_t start, systime_t end)

Checks if the current system time is within the specified time window.

Note

When *start*=*end* then the function returns always true because the whole time range is specified.

Parameters

in	<i>start</i> the start of the time window (inclusive)
in	<i>end</i> the end of the time window (non inclusive)

Return values

<i>TRUE</i>	current time within the specified time window.
-------------	--

<i>FALSE</i>	current time not within the specified time window.
--------------	--

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.9.3 Variable Documentation

7.9.3.1 VTLList vtlist

Virtual timers delta list header.

7.9.3.2 VTLList vtlist

Virtual timers delta list header.

7.9.4 Define Documentation

7.9.4.1 #define S2ST(sec) ((systime_t)((sec) * CH_FREQUENCY))

Seconds to system ticks.

Converts from seconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.

Parameters

in sec number of seconds

Returns

The number of ticks.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.9.4.2 #define MS2ST(msec) (((systime_t)((((msec) - 1L) * CH_FREQUENCY) / 1000L) + 1L))

Milliseconds to system ticks.

Converts from milliseconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.

Parameters

in msec number of milliseconds

Returns

The number of ticks.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.9.4.3 #define US2ST(*usec*) ((systime_t)((((usec) - 1L) * CH_FREQUENCY) / 1000000L) + 1L)

Microseconds to system ticks.

Converts from microseconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.

Parameters

in *usec* number of microseconds

Returns

The number of ticks.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.9.4.4 #define chVTDoTickl()

Value:

```
{
    vtlist.vt_systime++; \
    if (&vtlist != (VTLList *)vtlist.vt_next) { \
        VirtualTimer *vtp; \
        \
        --vtlist.vt_next->vt_time; \
        while (!(vtp = vtlist.vt_next)->vt_time) { \
            vtfunc_t fn = vtp->vt_func; \
            vtp->vt_func = (vtfunc_t)NULL; \
            vtp->vt_next->vt_prev = (void *)&vtlist; \
            (&vtlist)->vt_next = vtp->vt_next; \
            fn(vtp->vt_par); \
        } \
    } \
}
```

Virtual timers ticker.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.9.4.5 #define chVTIsArmedl(*vtp*) ((vtp)->vt_func != NULL)

Returns TRUE if the specified timer is armed.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.9.4.6 #define chTimeNow() (vtlist.vt_systime)

Current system time.

Returns the number of system ticks since the [chSysInit\(\)](#) invocation.

Note

The counter can reach its maximum and then restart from zero.
This function is designed to work with the [chThdSleepUntil\(\)](#).

Returns

The system time in ticks.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.9.5 Typedef Documentation

7.9.5.1 `typedef void(* vfunc_t)(void *)`

Virtual Timer callback function.

7.9.5.2 `typedef struct VirtualTimer VirtualTimer`

Virtual Timer structure type.

7.10 Synchronization

7.10.1 Detailed Description

Synchronization services.

Modules

- [Counting Semaphores](#)
- [Binary Semaphores](#)
- [Mutexes](#)
- [Condition Variables](#)
- [Event Flags](#)
- [Synchronous Messages](#)
- [Mailboxes](#)

7.11 Counting Semaphores

7.11.1 Detailed Description

Semaphores related APIs and services.

Operation mode

Semaphores are a flexible synchronization primitive, ChibiOS/RT implements semaphores in their "counting semaphores" variant as defined by Edsger Dijkstra plus several enhancements like:

- Wait operation with timeout.
- Reset operation.

- Atomic wait+signal operation.
- Return message from the wait operation (OK, RESET, TIMEOUT).

The binary semaphores variant can be easily implemented using counting semaphores.

Operations defined for semaphores:

- **Signal:** The semaphore counter is increased and if the result is non-positive then a waiting thread is removed from the semaphore queue and made ready for execution.
- **Wait:** The semaphore counter is decreased and if the result becomes negative the thread is queued in the semaphore and suspended.
- **Reset:** The semaphore counter is reset to a non-negative value and all the threads in the queue are released.

Semaphores can be used as guards for mutual exclusion zones (note that mutexes are recommended for this kind of use) but also have other uses, queues guards and counters for example.

Semaphores usually use a FIFO queuing strategy but it is possible to make them order threads by priority by enabling CH_USE_SEMAPHORES_PRIORITY in [chconf.h](#).

Precondition

In order to use the semaphore APIs the CH_USE_SEMAPHORES option must be enabled in [chconf.h](#).

Data Structures

- struct [Semaphore](#)
Semaphore structure.

Functions

- void [chSemInit](#) ([Semaphore](#) *sp, [cnt_t](#) n)
Initializes a semaphore with the specified counter value.
- void [chSemReset](#) ([Semaphore](#) *sp, [cnt_t](#) n)
Performs a reset operation on the semaphore.
- void [chSemReset1](#) ([Semaphore](#) *sp, [cnt_t](#) n)
Performs a reset operation on the semaphore.
- [msg_t](#) [chSemWait](#) ([Semaphore](#) *sp)
Performs a wait operation on a semaphore.
- [msg_t](#) [chSemWaitS](#) ([Semaphore](#) *sp)
Performs a wait operation on a semaphore.
- [msg_t](#) [chSemWaitTimeout](#) ([Semaphore](#) *sp, [systime_t](#) time)
Performs a wait operation on a semaphore with timeout specification.
- [msg_t](#) [chSemWaitTimeoutS](#) ([Semaphore](#) *sp, [systime_t](#) time)
Performs a wait operation on a semaphore with timeout specification.
- void [chSemSignal](#) ([Semaphore](#) *sp)
Performs a signal operation on a semaphore.
- void [chSemSignall](#) ([Semaphore](#) *sp)
Performs a signal operation on a semaphore.
- void [chSemAddCounter1](#) ([Semaphore](#) *sp, [cnt_t](#) n)
Adds the specified value to the semaphore counter.
- [msg_t](#) [chSemSignalWait](#) ([Semaphore](#) *sp, [Semaphore](#) *spw)
Performs atomic signal and wait operations on two semaphores.

Macro Functions

- `#define chSemFastWait(sp) ((sp)->s_cnt--)`
Decreases the semaphore counter.
- `#define chSemFastSignal(sp) ((sp)->s_cnt++)`
Increases the semaphore counter.
- `#define chSemGetCounter(sp) ((sp)->s_cnt)`
Returns the semaphore counter current value.

Defines

- `#define _SEMAPHORE_DATA(name, n) {_THREADSQUEUE_DATA(name.s_queue), n}`
Data part of a static semaphore initializer.
- `#define SEMAPHORE_DECL(name, n) Semaphore name = _SEMAPHORE_DATA(name, n)`
Static semaphore initializer.

TypeDefs

- `typedef struct Semaphore Semaphore`
Semaphore structure.

7.11.2 Function Documentation

7.11.2.1 void chSemInit (Semaphore * sp, cnt_t n)

Initializes a semaphore with the specified counter value.

Parameters

out	<code>sp</code>	pointer to a <code>Semaphore</code> structure
in	<code>n</code>	initial value of the semaphore counter. Must be non-negative.

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.11.2.2 void chSemReset (Semaphore * sp, cnt_t n)

Performs a reset operation on the semaphore.

Postcondition

After invoking this function all the threads waiting on the semaphore, if any, are released and the semaphore counter is set to the specified, non negative, value.

Note

The released threads can recognize they were wakened up by a reset rather than a signal because the `chSemWait()` will return `RDY_RESET` instead of `RDY_OK`.

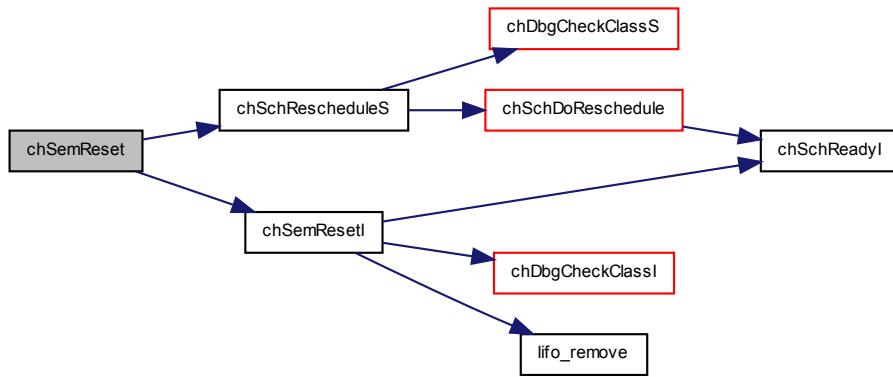
Parameters

in	<code>sp</code>	pointer to a <code>Semaphore</code> structure
in	<code>n</code>	the new value of the semaphore counter. The value must be non-negative.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.11.2.3 void chSemReset(Semaphore * sp, cnt_t n)

Performs a reset operation on the semaphore.

Postcondition

After invoking this function all the threads waiting on the semaphore, if any, are released and the semaphore counter is set to the specified, non negative, value.

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

Note

The released threads can recognize they were wakened up by a reset rather than a signal because the `chSemWait()` will return `RDY_RESET` instead of `RDY_OK`.

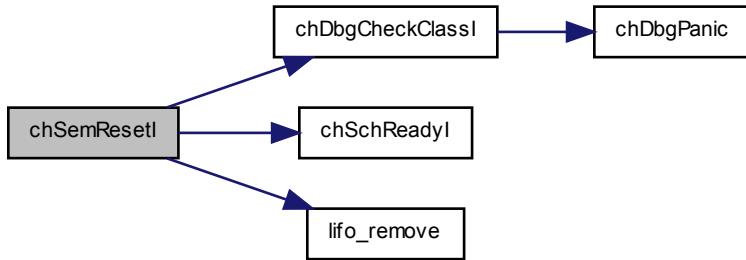
Parameters

in	<code>sp</code>	pointer to a Semaphore structure
in	<code>n</code>	the new value of the semaphore counter. The value must be non-negative.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.11.2.4 msg_t chSemWait (Semaphore * sp)

Performs a wait operation on a semaphore.

Parameters

in `sp` pointer to a `Semaphore` structure

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

<code>RDY_OK</code>	if the thread has not stopped on the semaphore or the semaphore has been signaled.
<code>RDY_RESET</code>	if the semaphore has been reset using <code>chSemReset ()</code> .

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.11.2.5 msg_t chSemWaitS (Semaphore * sp)

Performs a wait operation on a semaphore.

Parameters

in *sp* pointer to a [Semaphore](#) structure

Returns

A message specifying how the invoking thread has been released from the semaphore.

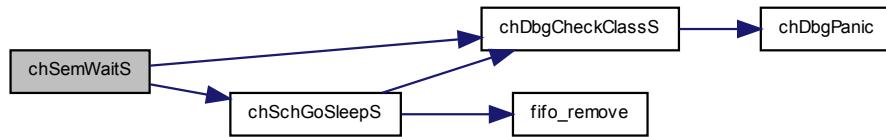
Return values

RDY_OK if the thread has not stopped on the semaphore or the semaphore has been signaled.
RDY_RESET if the semaphore has been reset using [chSemReset \(\)](#).

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.11.2.6 msg_t chSemWaitTimeout (Semaphore * sp, systime_t time)

Performs a wait operation on a semaphore with timeout specification.

Parameters

in	<i>sp</i>	pointer to a Semaphore structure
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

RDY_OK if the thread has not stopped on the semaphore or the semaphore has been signaled.
RDY_RESET if the semaphore has been reset using [chSemReset \(\)](#).
RDY_TIMEOUT if the semaphore has not been signaled or reset within the specified timeout.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.11.2.7 msg_t chSemWaitTimeoutS (Semaphore * sp, systime_t time)

Performs a wait operation on a semaphore with timeout specification.

Parameters

in	<i>sp</i>	pointer to a Semaphore structure
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

A message specifying how the invoking thread has been released from the semaphore.

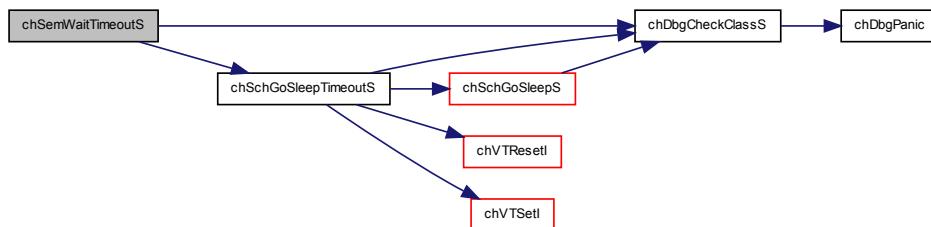
Return values

<i>RDY_OK</i>	if the thread has not stopped on the semaphore or the semaphore has been signaled.
<i>RDY_RESET</i>	if the semaphore has been reset using chSemReset () .
<i>RDY_TIMEOUT</i>	if the semaphore has not been signaled or reset within the specified timeout.

Function Class:

This is an **S-Class API**, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.11.2.8 void chSemSignal (Semaphore * sp)

Performs a signal operation on a semaphore.

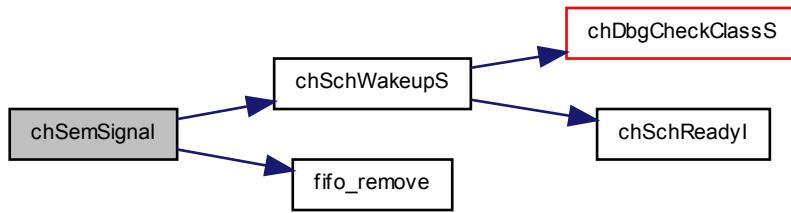
Parameters

in *sp* pointer to a [Semaphore](#) structure

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

**7.11.2.9 void chSemSignall (Semaphore * sp)**

Performs a signal operation on a semaphore.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

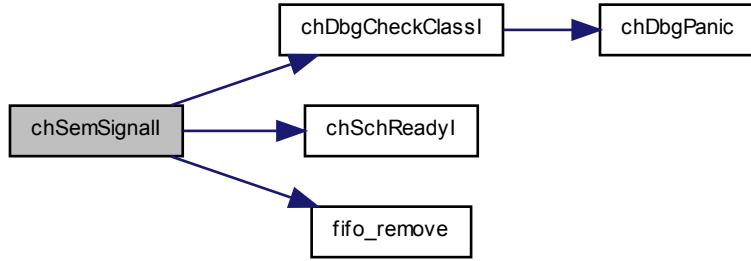
Parameters

in *sp* pointer to a [Semaphore](#) structure

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.11.2.10 void chSemAddCounterI (Semaphore * sp, cnt_t n)

Adds the specified value to the semaphore counter.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

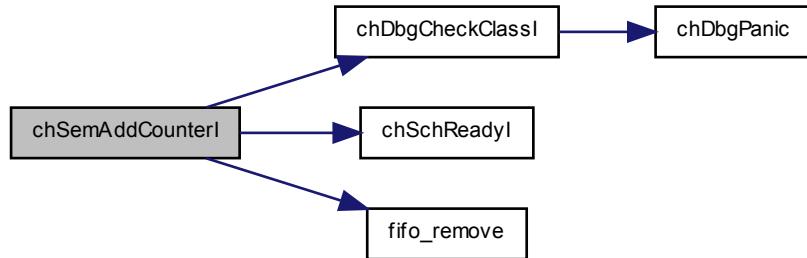
Parameters

in	<code>sp</code>	pointer to a Semaphore structure
in	<code>n</code>	value to be added to the semaphore counter. The value must be positive.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.11.2.11 `msg_t chSemSignalWait (Semaphore * sps, Semaphore * spw)`

Performs atomic signal and wait operations on two semaphores.

Precondition

The configuration option `CH_USE_SEMSW` must be enabled in order to use this function.

Parameters

in	<code>sps</code>	pointer to a <code>Semaphore</code> structure to be signaled
in	<code>spw</code>	pointer to a <code>Semaphore</code> structure to be wait on

Returns

A message specifying how the invoking thread has been released from the semaphore.

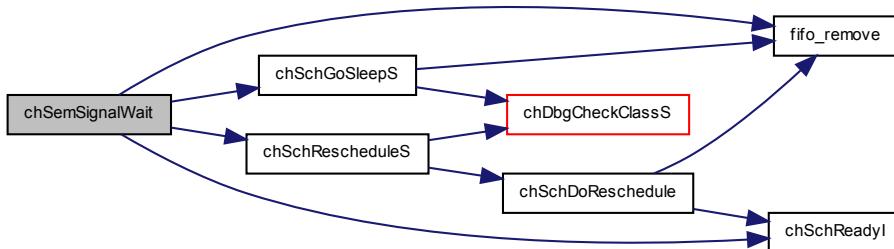
Return values

<code>RDY_OK</code>	if the thread has not stopped on the semaphore or the semaphore has been signaled.
<code>RDY_RESET</code>	if the semaphore has been reset using <code>chSemReset ()</code> .

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.11.3 Define Documentation

7.11.3.1 `#define _SEMAPHORE_DATA(name, n) { _THREADSQUEUE_DATA(name.s.queue), n }`

Data part of a static semaphore initializer.

This macro should be used when statically initializing a semaphore that is part of a bigger structure.

Parameters

in	<code>name</code>	the name of the semaphore variable
in	<code>n</code>	the counter initial value, this value must be non-negative

7.11.3.2 `#define SEMAPHORE_DECL(name, n) Semaphore name = _SEMAPHORE_DATA(name, n)`

Static semaphore initializer.

Statically initialized semaphores require no explicit initialization using `chSemInit()`.

Parameters

in	<i>name</i>	the name of the semaphore variable
in	<i>n</i>	the counter initial value, this value must be non-negative

7.11.3.3 `#define chSemFastWait(sp) ((sp)->s_cnt--)`

Decreases the semaphore counter.

This macro can be used when the counter is known to be positive.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.11.3.4 `#define chSemFastSignal(sp) ((sp)->s_cnt++)`

Increases the semaphore counter.

This macro can be used when the counter is known to be not negative.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.11.3.5 `#define chSemGetCounter(sp) ((sp)->s_cnt)`

Returns the semaphore counter current value.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.11.4 Typedef Documentation

7.11.4.1 `typedef struct Semaphore Semaphore`

`Semaphore` structure.

7.12 Binary Semaphores

7.12.1 Detailed Description

Binary semaphores related APIs and services.

Operation mode

Binary semaphores are implemented as a set of macros that use the existing counting semaphores primitives. The difference between counting and binary semaphores is that the counter of binary semaphores is not allowed to grow above the value 1. Repeated signal operation are ignored. A binary semaphore can thus have only two defined states:

- **Taken**, when its counter has a value of zero or lower than zero. A negative number represent the number of threads queued on the binary semaphore.
- **Not taken**, when its counter has a value of one.

Binary semaphores are different from mutexes because there is no the concept of ownership, a binary semaphore can be taken by a thread and signaled by another thread or an interrupt handler, mutexes can only be taken and released by the same thread. Another difference is that binary semaphores, unlike mutexes, do not implement the priority inheritance protocol.

In order to use the binary semaphores APIs the `CH_USE_SEMAPHORES` option must be enabled in [chconf.h](#).

Data Structures

- struct [BinarySemaphore](#)
Binary semaphore type.

Macro Functions

- #define [chBSemInit](#)(bsp, taken) chSemInit(&(bsp)->bs_sem, (taken) ? 0 : 1)
Initializes a binary semaphore.
- #define [chBSemWait](#)(bsp) chSemWait(&(bsp)->bs_sem)
Wait operation on the binary semaphore.
- #define [chBSemWaitS](#)(bsp) chSemWaitS(&(bsp)->bs_sem)
Wait operation on the binary semaphore.
- #define [chBSemWaitTimeout](#)(bsp, time) chSemWaitTimeout(&(bsp)->bs_sem, (time))
Wait operation on the binary semaphore.
- #define [chBSemWaitTimeoutS](#)(bsp, time) chSemWaitTimeoutS(&(bsp)->bs_sem, (time))
Wait operation on the binary semaphore.
- #define [chBSemReset](#)(bsp, taken) chSemReset(&(bsp)->bs_sem, (taken) ? 0 : 1)
Reset operation on the binary semaphore.
- #define [chBSemResetI](#)(bsp, taken) chSemResetI(&(bsp)->bs_sem, (taken) ? 0 : 1)
Reset operation on the binary semaphore.
- #define [chBSemSignal](#)(bsp)
Performs a signal operation on a binary semaphore.
- #define [chBSemSignall](#)(bsp)
Performs a signal operation on a binary semaphore.
- #define [chBSemGetState](#)(bsp) ((bsp)->bs_sem.s_cnt > 0 ? FALSE : TRUE)
Returns the binary semaphore current state.

Defines

- #define [_BSEMAPHORE_DATA](#)(name, taken) {_SEMAPHORE_DATA(name.bs_sem, ((taken) ? 0 : 1))}
Data part of a static semaphore initializer.
- #define [BSEMAPHORE_DECL](#)(name, taken) [BinarySemaphore](#) name = [_BSEMAPHORE_DATA](#)(name, taken)
Static semaphore initializer.

7.12.2 Define Documentation

7.12.2.1 `#define _BSEMAPHORE_DATA(name, taken) { _SEMAPHORE_DATA(name.bs_sem, ((taken) ? 0 : 1)) }`

Data part of a static semaphore initializer.

This macro should be used when statically initializing a semaphore that is part of a bigger structure.

Parameters

in	<i>name</i>	the name of the semaphore variable
in	<i>taken</i>	the semaphore initial state

7.12.2.2 `#define BSEMAPHORE_DECL(name, taken) BinarySemaphore name = _BSEMAPHORE_DATA(name, taken)`

Static semaphore initializer.

Statically initialized semaphores require no explicit initialization using `chSemInit()`.

Parameters

in	<i>name</i>	the name of the semaphore variable
in	<i>taken</i>	the semaphore initial state

7.12.2.3 `#define chBSemInit(bsp, taken) chSemInit(&(bsp)->bs_sem, (taken) ? 0 : 1)`

Initializes a binary semaphore.

Parameters

out	<i>bsp</i>	pointer to a <code>BinarySemaphore</code> structure
in	<i>taken</i>	initial state of the binary semaphore: <ul style="list-style-type: none">• <i>FALSE</i>, the initial state is not taken.• <i>TRUE</i>, the initial state is taken.

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.12.2.4 `#define chBSemWait(bsp) chSemWait(&(bsp)->bs_sem)`

Wait operation on the binary semaphore.

Parameters

in	<i>bsp</i>	pointer to a <code>BinarySemaphore</code> structure
----	------------	---

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

<code>RDY_OK</code>	if the binary semaphore has been successfully taken.
<code>RDY_RESET</code>	if the binary semaphore has been reset using <code>bsemReset()</code> .

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.12.2.5 #define chBSemWaitS(*bsp*) chSemWaitS(&(*bsp*)->bs_sem)

Wait operation on the binary semaphore.

Parameters

in	<i>bsp</i> pointer to a BinarySemaphore structure
----	---

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

RDY_OK if the binary semaphore has been successfully taken.

RDY_RESET if the binary semaphore has been reset using `bsemReset()`.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.12.2.6 #define chBSemWaitTimeout(*bsp*, *time*) chSemWaitTimeout(&(*bsp*)->bs_sem, (*time*))

Wait operation on the binary semaphore.

Parameters

in	<i>bsp</i> pointer to a BinarySemaphore structure
----	---

in	<i>time</i> the number of ticks before the operation timeouts, the following special values are allowed:
----	--

- *TIME_IMMEDIATE* immediate timeout.
- *TIME_INFINITE* no timeout.

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

RDY_OK if the binary semaphore has been successfully taken.

RDY_RESET if the binary semaphore has been reset using `bsemReset()`.

RDY_TIMEOUT if the binary semaphore has not been signaled or reset within the specified timeout.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.12.2.7 #define chBSemWaitTimeoutS(*bsp*, *time*) chSemWaitTimeoutS(&(*bsp*)->bs_sem, (*time*))

Wait operation on the binary semaphore.

Parameters

in	<i>bsp</i> pointer to a BinarySemaphore structure
----	---

in *time* the number of ticks before the operation timeouts, the following special values are allowed:

- *TIME_IMMEDIATE* immediate timeout.
- *TIME_INFINITE* no timeout.

Returns

A message specifying how the invoking thread has been released from the semaphore.

Return values

<i>RDY_OK</i>	if the binary semaphore has been successfully taken.
<i>RDY_RESET</i>	if the binary semaphore has been reset using <code>bsemReset()</code> .
<i>RDY_TIMEOUT</i>	if the binary semaphore has not been signaled or reset within the specified timeout.

Function Class:

This is an **S-Class API**, this function can be invoked from within a system lock zone by threads only.

7.12.2.8 `#define chBSemReset(bsp, taken) chSemReset(&(bsp)->bs_sem, (taken) ? 0 : 1)`

Reset operation on the binary semaphore.

Note

The released threads can recognize they were waked up by a reset rather than a signal because the `bsemWait()` will return *RDY_RESET* instead of *RDY_OK*.

Parameters

in	<i>bsp</i>	pointer to a <code>BinarySemaphore</code> structure
in	<i>taken</i>	new state of the binary semaphore <ul style="list-style-type: none"> • <i>FALSE</i>, the new state is not taken. • <i>TRUE</i>, the new state is taken.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.12.2.9 `#define chBSemResetI(bsp, taken) chSemResetI(&(bsp)->bs_sem, (taken) ? 0 : 1)`

Reset operation on the binary semaphore.

Note

The released threads can recognize they were waked up by a reset rather than a signal because the `bsemWait()` will return *RDY_RESET* instead of *RDY_OK*.

This function does not reschedule.

Parameters

in	<i>bsp</i>	pointer to a <code>BinarySemaphore</code> structure
in	<i>taken</i>	new state of the binary semaphore <ul style="list-style-type: none"> • <i>FALSE</i>, the new state is not taken. • <i>TRUE</i>, the new state is taken.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.12.2.10 #define chBSemSignal(*bsp*)

Value:

```
{
    chSysLock();
    chBSemSignalI((bsp));
    chSchRescheduleS();
    chSysUnlock();
}
```

Performs a signal operation on a binary semaphore.

Parameters

in *bsp* pointer to a [BinarySemaphore](#) structure

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.12.2.11 #define chBSemSignall(*bsp*)

Value:

```
{
    if ((bsp)->bs_sem.s_cnt < 1)
        chSemSignalI(&(bsp)->bs_sem);
}
```

Performs a signal operation on a binary semaphore.

Note

This function does not reschedule.

Parameters

in *bsp* pointer to a [BinarySemaphore](#) structure

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.12.2.12 #define chBSemGetState(*bsp*) ((bsp)->bs_sem.s_cnt > 0 ? FALSE : TRUE)

Returns the binary semaphore current state.

Parameters

in *bsp* pointer to a [BinarySemaphore](#) structure

Returns

The binary semaphore current state.

Return values

FALSE if the binary semaphore is not taken.

TRUE if the binary semaphore is taken.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.13 Mutexes

7.13.1 Detailed Description

Mutexes related APIs and services.

Operation mode

A mutex is a threads synchronization object that can be in two distinct states:

- Not owned (unlocked).
- Owned by a thread (locked).

Operations defined for mutexes:

- **Lock:** The mutex is checked, if the mutex is not owned by some other thread then it is associated to the locking thread else the thread is queued on the mutex in a list ordered by priority.
- **Unlock:** The mutex is released by the owner and the highest priority thread waiting in the queue, if any, is resumed and made owner of the mutex.

Constraints

In ChibiOS/RT the Unlock operations are always performed in lock-reverse order. The unlock API does not even have a parameter, the mutex to unlock is selected from an internal, per-thread, stack of owned mutexes. This both improves the performance and is required for an efficient implementation of the priority inheritance mechanism.

The priority inversion problem

The mutexes in ChibiOS/RT implements the **full** priority inheritance mechanism in order handle the priority inversion problem.

When a thread is queued on a mutex, any thread, directly or indirectly, holding the mutex gains the same priority of the waiting thread (if their priority was not already equal or higher). The mechanism works with any number of nested mutexes and any number of involved threads. The algorithm complexity (worst case) is N with N equal to the number of nested mutexes.

Precondition

In order to use the mutex APIs the `CH_USE_MUTEXES` option must be enabled in `chconf.h`.

Postcondition

Enabling mutexes requires 5-12 (depending on the architecture) extra bytes in the `Thread` structure.

Data Structures

- struct **Mutex**

Mutex structure.

Functions

- void **chMtxInit** (**Mutex** *mp)
*Initializes s **Mutex** structure.*
- void **chMtxLock** (**Mutex** *mp)
Locks the specified mutex.
- void **chMtxLockS** (**Mutex** *mp)
Locks the specified mutex.
- **bool_t** **chMtxTryLock** (**Mutex** *mp)
Tries to lock a mutex.
- **bool_t** **chMtxTryLockS** (**Mutex** *mp)
Tries to lock a mutex.
- **Mutex** * **chMtxUnlock** (void)
Unlocks the next owned mutex in reverse lock order.
- **Mutex** * **chMtxUnlockS** (void)
Unlocks the next owned mutex in reverse lock order.
- void **chMtxUnlockAll** (void)
Unlocks all the mutexes owned by the invoking thread.

Macro Functions

- #define **chMtxQueueNotEmptyS**(mp) notempty(&(mp)->m_queue)
Returns TRUE if the mutex queue contains at least a waiting thread.

Defines

- #define **_MUTEX_DATA**(name) {_THREADSQUEUE_DATA(name.m_queue), NULL, NULL}
Data part of a static mutex initializer.
- #define **MUTEX_DECL**(name) **Mutex** name = **_MUTEX_DATA**(name)
Static mutex initializer.

Typedefs

- typedef struct **Mutex** **Mutex**
Mutex structure.

7.13.2 Function Documentation

7.13.2.1 void chMtxInit (**Mutex** * mp)

Initializes s **Mutex** structure.

Parameters

out	<i>mp</i> pointer to a Mutex structure
-----	---

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.13.2.2 void chMtxLock (**Mutex** * *mp*)

Locks the specified mutex.

Postcondition

The mutex is locked and inserted in the per-thread stack of owned mutexes.

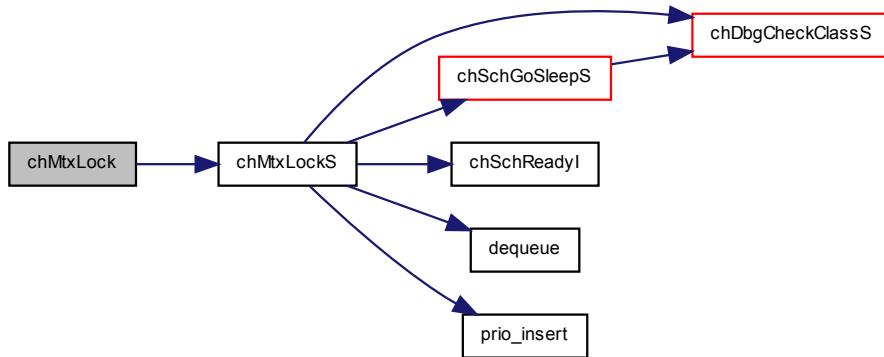
Parameters

in *mp* pointer to the [Mutex](#) structure

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

7.13.2.3 void chMtxLockS (**Mutex** * *mp*)

Locks the specified mutex.

Postcondition

The mutex is locked and inserted in the per-thread stack of owned mutexes.

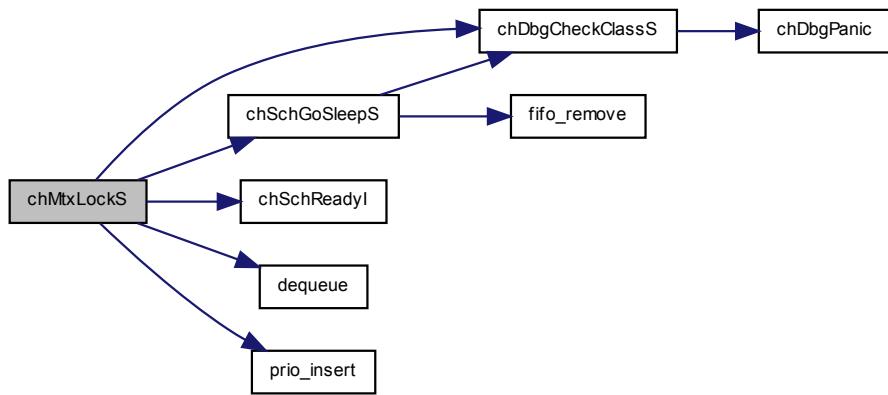
Parameters

in *mp* pointer to the [Mutex](#) structure

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.13.2.4 `bool_t chMtxTryLock (Mutex * mp)`

Tries to lock a mutex.

This function attempts to lock a mutex, if the mutex is already locked by another thread then the function exits without waiting.

Postcondition

The mutex is locked and inserted in the per-thread stack of owned mutexes.

Note

This function does not have any overhead related to the priority inheritance mechanism because it does not try to enter a sleep state.

Parameters

`in` `mp` pointer to the `Mutex` structure

Returns

The operation status.

Return values

`TRUE` if the mutex has been successfully acquired

`FALSE` if the lock attempt failed.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.13.2.5 `bool_t chMtxTryLockS (Mutex * mp)`

Tries to lock a mutex.

This function attempts to lock a mutex, if the mutex is already taken by another thread then the function exits without waiting.

Postcondition

The mutex is locked and inserted in the per-thread stack of owned mutexes.

Note

This function does not have any overhead related to the priority inheritance mechanism because it does not try to enter a sleep state.

Parameters

in *mp* pointer to the [Mutex](#) structure

Returns

The operation status.

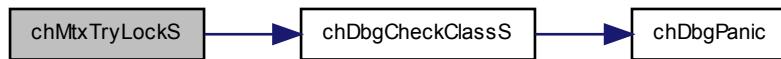
Return values

<i>TRUE</i>	if the mutex has been successfully acquired
<i>FALSE</i>	if the lock attempt failed.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.13.2.6 `Mutex * chMtxUnlock(void)`

Unlocks the next owned mutex in reverse lock order.

Precondition

The invoking thread **must** have at least one owned mutex.

Postcondition

The mutex is unlocked and removed from the per-thread stack of owned mutexes.

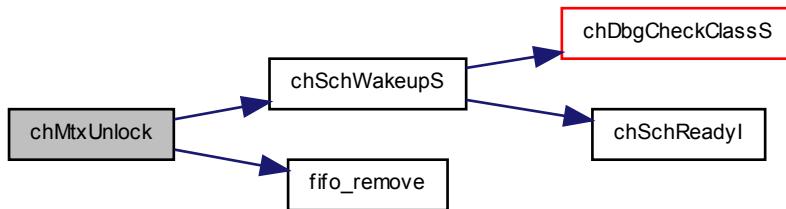
Returns

A pointer to the unlocked mutex.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.13.2.7 `Mutex * chMtxUnlockS(void)`

Unlocks the next owned mutex in reverse lock order.

Precondition

The invoking thread **must** have at least one owned mutex.

Postcondition

The mutex is unlocked and removed from the per-thread stack of owned mutexes.
This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel.

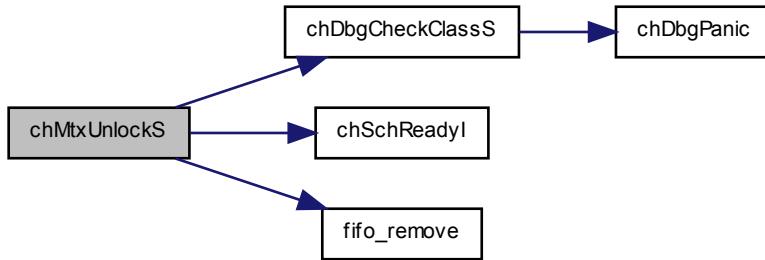
Returns

A pointer to the unlocked mutex.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.13.2.8 void chMtxUnlockAll (void)

Unlocks all the mutexes owned by the invoking thread.

Postcondition

The stack of owned mutexes is emptied and all the found mutexes are unlocked.

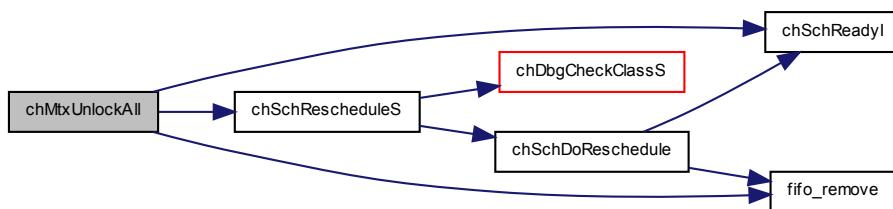
Note

This function is **MUCH MORE** efficient than releasing the mutexes one by one and not just because the call overhead, this function does not have any overhead related to the priority inheritance mechanism.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.13.3 Define Documentation

7.13.3.1 #define _MUTEX_DATA(name) { _THREADSQUEUE_DATA(name.m_queue), NULL, NULL }

Data part of a static mutex initializer.

This macro should be used when statically initializing a mutex that is part of a bigger structure.

Parameters

in *name* the name of the mutex variable

7.13.3.2 #define MUTEX_DECL(*name*) Mutex name = _MUTEX_DATA(*name*)

Static mutex initializer.

Statically initialized mutexes require no explicit initialization using [chMtxInit\(\)](#).

Parameters

in *name* the name of the mutex variable

7.13.3.3 #define chMtxQueueNotEmptyS(*mp*) notempty(&(*mp*)->m_queue)

Returns TRUE if the mutex queue contains at least a waiting thread.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.13.4 Typedef Documentation

7.13.4.1 typedef struct Mutex Mutex

[Mutex](#) structure.

7.14 Condition Variables

7.14.1 Detailed Description

This module implements the Condition Variables mechanism. Condition variables are an extensions to the [Mutex](#) subsystem and cannot work alone.

Operation mode

The condition variable is a synchronization object meant to be used inside a zone protected by a [Mutex](#). Mutexes and CondVars together can implement a Monitor construct.

Precondition

In order to use the condition variable APIs the CH_USE_CONDVAR option must be enabled in [chconf.h](#).

Data Structures

- struct [CondVar](#)
CondVar structure.

Functions

- void [chCondInit](#) ([CondVar](#) *cp)
Initializes a [CondVar](#) structure.

- void `chCondSignal (CondVar *cp)`
Signals one thread that is waiting on the condition variable.
- void `chCondSignall (CondVar *cp)`
Signals one thread that is waiting on the condition variable.
- void `chCondBroadcast (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- void `chCondBroadcastl (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- `msg_t chCondWait (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitS (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeout (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeoutS (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.

Defines

- `#define _CONDVAR_DATA(name) {_THREADSQUEUE_DATA(name.c_queue)}`
Data part of a static condition variable initializer.
- `#define CONDVAR_DECL(name) CondVar name = _CONDVAR_DATA(name)`
Static condition variable initializer.

TypeDefs

- `typedef struct CondVar CondVar`
CondVar structure.

7.14.2 Function Documentation

7.14.2.1 void chCondInit (CondVar * cp)

Initializes a `CondVar` structure.

Parameters

out	<code>cp</code> pointer to a <code>CondVar</code> structure
-----	---

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.14.2.2 void chCondSignal (CondVar * cp)

Signals one thread that is waiting on the condition variable.

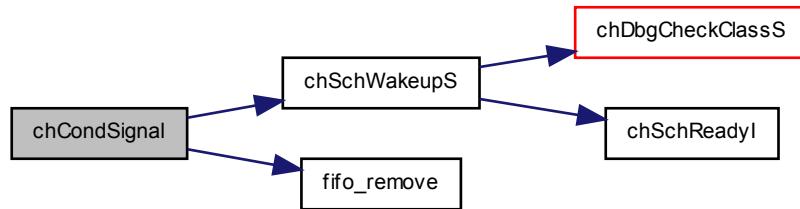
Parameters

in	<code>cp</code> pointer to the <code>CondVar</code> structure
----	---

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

7.14.2.3 `void chCondSignal(CondVar * cp)`

Signals one thread that is waiting on the condition variable.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

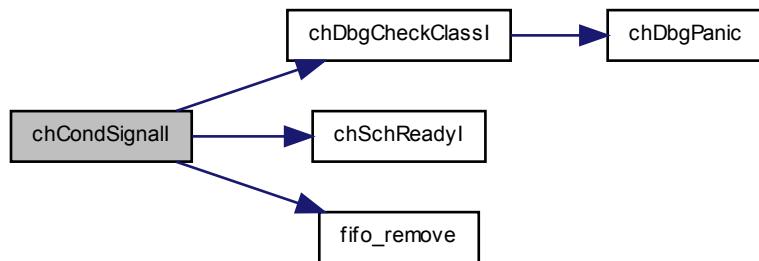
Parameters

in `cp` pointer to the `CondVar` structure

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.14.2.4 void chCondBroadcast (*CondVar* * *cp*)

Signals all threads that are waiting on the condition variable.

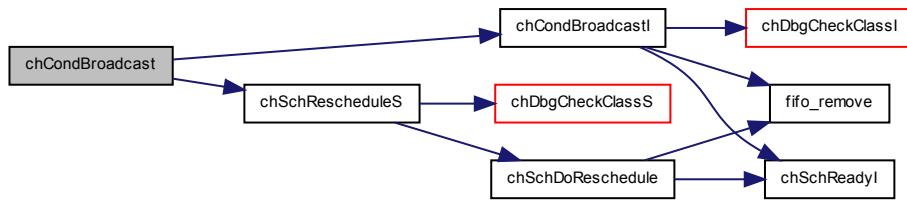
Parameters

in *cp* pointer to the *CondVar* structure

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.14.2.5 void chCondBroadcastI (*CondVar* * *cp*)

Signals all threads that are waiting on the condition variable.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

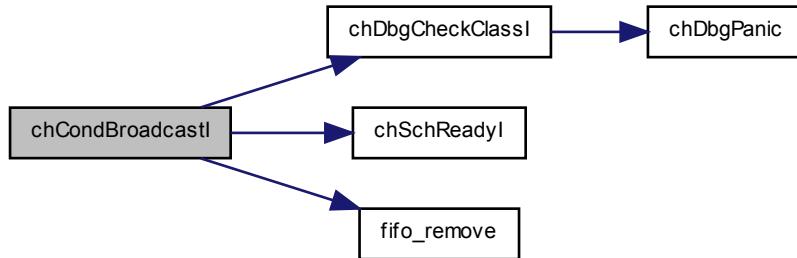
Parameters

in *cp* pointer to the *CondVar* structure

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.14.2.6 `msg_t chCondWait(CondVar * cp)`

Waits on the condition variable releasing the mutex lock.

Releases the currently owned mutex, waits on the condition variable, and finally acquires the mutex again. All the sequence is performed atomically.

Precondition

The invoking thread **must** have at least one owned mutex.

Parameters

in	<code>cp</code> pointer to the <code>CondVar</code> structure
----	---

Returns

A message specifying how the invoking thread has been released from the condition variable.

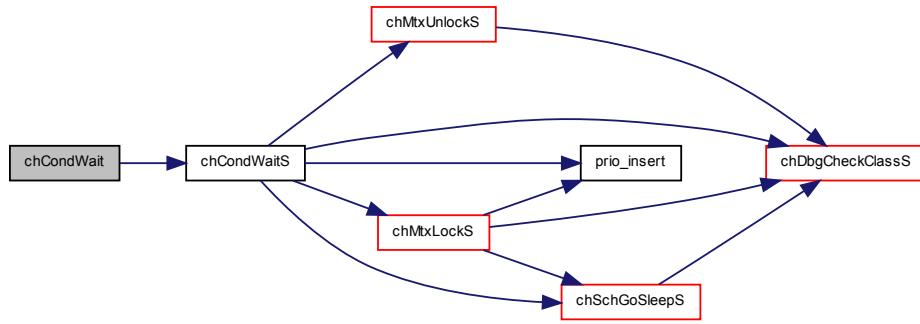
Return values

<code>RDY_OK</code>	if the condvar has been signaled using <code>chCondSignal()</code> .
<code>RDY_RESET</code>	if the condvar has been signaled using <code>chCondBroadcast()</code> .

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.14.2.7 msg_t chCondWaitS (CondVar * cp)

Waits on the condition variable releasing the mutex lock.

Releases the currently owned mutex, waits on the condition variable, and finally acquires the mutex again. All the sequence is performed atomically.

Precondition

The invoking thread **must** have at least one owned mutex.

Parameters

in	<code>cp</code> pointer to the <code>CondVar</code> structure
----	---

Returns

A message specifying how the invoking thread has been released from the condition variable.

Return values

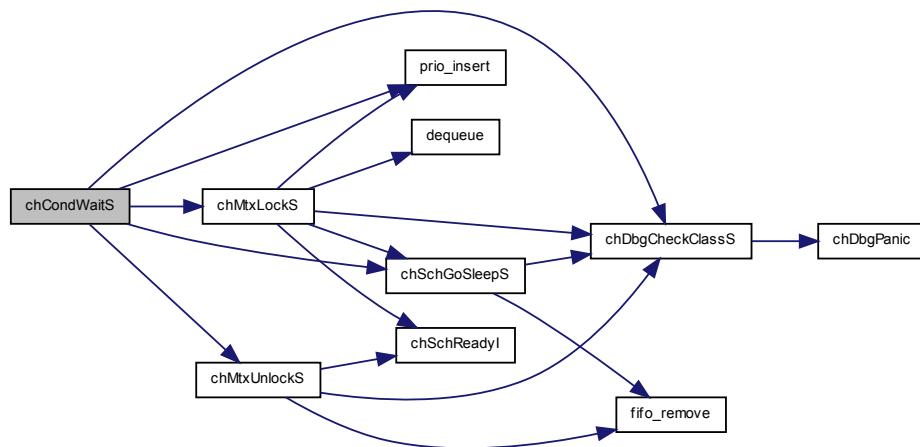
`RDY_OK` if the condvar has been signaled using `chCondSignal()`.

`RDY_RESET` if the condvar has been signaled using `chCondBroadcast()`.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.14.2.8 msg_t chCondWaitTimeout (CondVar * cp, systime_t time)

Waits on the condition variable releasing the mutex lock.

Releases the currently owned mutex, waits on the condition variable, and finally acquires the mutex again. All the sequence is performed atomically.

Precondition

The invoking thread **must** have at least one owned mutex.

The configuration option `CH_USE_CONDVAR_TIMEOUT` must be enabled in order to use this function.

Postcondition

Exiting the function because a timeout does not re-acquire the mutex, the mutex ownership is lost.

Parameters

in	<code>cp</code>	pointer to the <code>CondVar</code> structure
in	<code>time</code>	the number of ticks before the operation timeouts, the special values are handled as follow: <ul style="list-style-type: none"> • <code>TIME_INFINITE</code> no timeout. • <code>TIME_IMMEDIATE</code> this value is not allowed.

Returns

A message specifying how the invoking thread has been released from the condition variable.

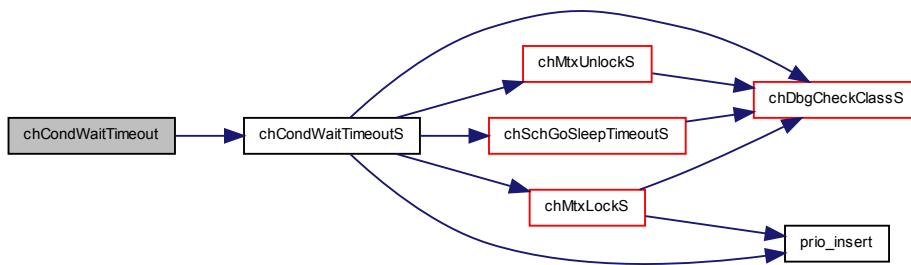
Return values

<code>RDY_OK</code>	if the condvar has been signaled using <code>chCondSignal()</code> .
<code>RDY_RESET</code>	if the condvar has been signaled using <code>chCondBroadcast()</code> .
<code>RDY_TIMEOUT</code>	if the condvar has not been signaled within the specified timeout.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.14.2.9 `msg_t chCondWaitTimeoutS(CondVar * cp, systime_t time)`

Waits on the condition variable releasing the mutex lock.

Releases the currently owned mutex, waits on the condition variable, and finally acquires the mutex again. All the sequence is performed atomically.

Precondition

The invoking thread **must** have at least one owned mutex.

The configuration option `CH_USE_CONDVAR_TIMEOUT` must be enabled in order to use this function.

Postcondition

Exiting the function because a timeout does not re-acquire the mutex, the mutex ownership is lost.

Parameters

in	<code>cp</code>	pointer to the <code>CondVar</code> structure
in	<code>time</code>	the number of ticks before the operation timeouts, the special values are handled as follow: <ul style="list-style-type: none"> • <code>TIME_INFINITE</code> no timeout. • <code>TIME_IMMEDIATE</code> this value is not allowed.

Returns

A message specifying how the invoking thread has been released from the condition variable.

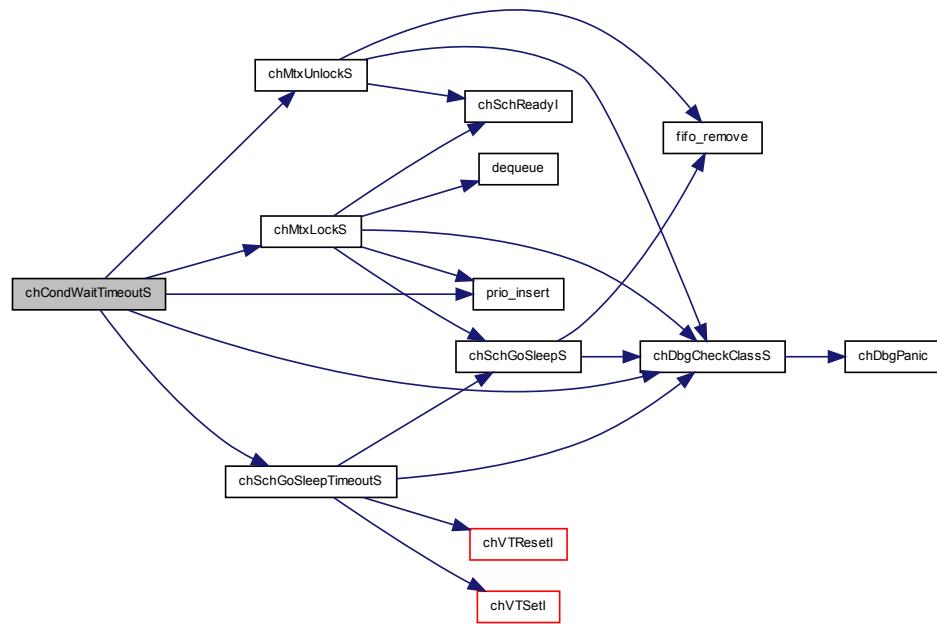
Return values

<code>RDY_OK</code>	if the condvar has been signaled using <code>chCondSignal()</code> .
<code>RDY_RESET</code>	if the condvar has been signaled using <code>chCondBroadcast()</code> .
<code>RDY_TIMEOUT</code>	if the condvar has not been signaled within the specified timeout.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.14.3 Define Documentation

7.14.3.1 `#define _CONDVAR_DATA(name) { _THREADSQUEUE_DATA(name.c_queue) }`

Data part of a static condition variable initializer.

This macro should be used when statically initializing a condition variable that is part of a bigger structure.

Parameters

`in` `name` the name of the condition variable

7.14.3.2 `#define CONDVAR_DECL(name) CondVar name = _CONDVAR_DATA(name)`

Static condition variable initializer.

Statically initialized condition variables require no explicit initialization using `chCondInit()`.

Parameters

`in` `name` the name of the condition variable

7.14.4 Typedef Documentation

7.14.4.1 `typedef struct CondVar CondVar`

`CondVar` structure.

7.15 Event Flags

7.15.1 Detailed Description

Event Flags, Event Sources and Event Listeners.

Operation mode

Each thread has a mask of pending event flags inside its [Thread](#) structure. Operations defined for event flags:

- **Wait**, the invoking thread goes to sleep until a certain AND/OR combination of event flags becomes pending.
- **Clear**, a mask of event flags is cleared from the pending events mask, the cleared event flags mask is returned (only the flags that were actually pending and then cleared).
- **Signal**, an event mask is directly ORed to the mask of the signaled thread.
- **Broadcast**, each thread registered on an Event Source is signaled with the event flags specified in its Event Listener.
- **Dispatch**, an events mask is scanned and for each bit set to one an associated handler function is invoked. Bit masks are scanned from bit zero upward.

An Event Source is a special object that can be "broadcasted" by a thread or an interrupt service routine. Broadcasting an Event Source has the effect that all the threads registered on the Event Source will be signaled with an events mask.

An unlimited number of Event Sources can exists in a system and each thread can be listening on an unlimited number of them.

Precondition

In order to use the Events APIs the `CH_USE_EVENTS` option must be enabled in [chconf.h](#).

Postcondition

Enabling events requires 1-4 (depending on the architecture) extra bytes in the [Thread](#) structure.

Data Structures

- struct [EventListener](#)
Event Listener structure.
- struct [EventSource](#)
Event Source structure.

Functions

- void [chEvtRegisterMask](#) ([EventSource](#) *esp, [EventListener](#) *elp, [eventmask_t](#) mask)
Registers an Event Listener on an Event Source.
- void [chEvtUnregister](#) ([EventSource](#) *esp, [EventListener](#) *elp)
Unregisters an Event Listener from its Event Source.
- [eventmask_t](#) [chEvtClearFlags](#) ([eventmask_t](#) mask)
Clears the pending events specified in the mask.
- [eventmask_t](#) [chEvtAddFlags](#) ([eventmask_t](#) mask)
*Adds (OR) a set of event flags on the current thread, this is **much** faster than using [chEvtBroadcast\(\)](#) or [chEvtSignal\(\)](#).*
- void [chEvtSignalFlags](#) ([Thread](#) *tp, [eventmask_t](#) mask)

- Adds (OR) a set of event flags on the specified *Thread*.
 - void **chEvtSignalFlagsI** (*Thread* *tp, *eventmask_t* mask)
 - Adds (OR) a set of event flags on the specified *Thread*.
 - void **chEvtBroadcastFlags** (*EventSource* *esp, *eventmask_t* mask)
 - Signals all the Event Listeners registered on the specified Event Source.
 - void **chEvtBroadcastFlagsI** (*EventSource* *esp, *eventmask_t* mask)
 - Signals all the Event Listeners registered on the specified Event Source.
- void **chEvtDispatch** (const *evhandler_t* *handlers, *eventmask_t* mask)
 - Invokes the event handlers associated to an event flags mask.
- *eventmask_t* **chEvtWaitOneTimeout** (*eventmask_t* mask, *systime_t* time)
 - Waits for exactly one of the specified events.
- *eventmask_t* **chEvtWaitAnyTimeout** (*eventmask_t* mask, *systime_t* time)
 - Waits for any of the specified events.
- *eventmask_t* **chEvtWaitAllTimeout** (*eventmask_t* mask, *systime_t* time)
 - Waits for all the specified events.
- *eventmask_t* **chEvtWaitOne** (*eventmask_t* mask)
 - Waits for exactly one of the specified events.
- *eventmask_t* **chEvtWaitAny** (*eventmask_t* mask)
 - Waits for any of the specified events.
- *eventmask_t* **chEvtWaitAll** (*eventmask_t* mask)
 - Waits for all the specified events.

Macro Functions

- #define **chEvtRegister**(esp, elp, eid) **chEvtRegisterMask**(esp, elp, EVENT_MASK(eid))
 - Registers an Event Listener on an Event Source.
- #define **chEvtInit**(esp) ((esp)->es_next = (*EventListener* *)(void *)(esp))
 - Initializes an Event Source.
- #define **chEvtIsListeningI**(esp) ((void *)(esp) != (void *)(esp)->es_next)
 - Verifies if there is at least one *EventListener* registered.
- #define **chEvtBroadcast**(esp) **chEvtBroadcastFlagsI**(esp, 0)
 - Signals all the Event Listeners registered on the specified Event Source.
- #define **chEvtBroadcastI**(esp) **chEvtBroadcastFlagsI**(esp, 0)
 - Signals all the Event Listeners registered on the specified Event Source.

Defines

- #define **_EVENTSOURCE_DATA**(name) {(void *)(&name)}
 - Data part of a static event source initializer.
- #define **EVENTSOURCE_DECL**(name) **EventSource** name = **_EVENTSOURCE_DATA**(name)
 - Static event source initializer.
- #define **ALL_EVENTS** ((*eventmask_t*)-1)
 - All events allowed mask.
- #define **EVENT_MASK**(eid) ((*eventmask_t*)(1 << (eid)))
 - Returns an event mask from an event identifier.

Typedefs

- typedef struct **EventSource** **EventSource**
 - Event Source structure.
- typedef void(* **evhandler_t**)(*eventid_t*)
 - Event Handler callback function.

7.15.2 Function Documentation

7.15.2.1 void chEvtRegisterMask (`EventSource * esp, EventListener * elp, eventmask_t mask`)

Registers an Event Listener on an Event Source.

Once a thread has registered as listener on an event source it will be notified of all events broadcasted there.

Note

Multiple Event Listeners can specify the same bits to be ORed to different threads.

Parameters

in	<code>esp</code>	pointer to the <code>EventSource</code> structure
in	<code>elp</code>	pointer to the <code>EventListener</code> structure
in	<code>mask</code>	the mask of event flags to be ORed to the thread when the event source is broadcasted

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.2.2 void chEvtUnregister (`EventSource * esp, EventListener * elp`)

Unregisters an Event Listener from its Event Source.

Note

If the event listener is not registered on the specified event source then the function does nothing.

For optimal performance it is better to perform the unregister operations in inverse order of the register operations (elements are found on top of the list).

Parameters

in	<code>esp</code>	pointer to the <code>EventSource</code> structure
in	<code>elp</code>	pointer to the <code>EventListener</code> structure

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.2.3 eventmask_t chEvtClearFlags (`eventmask_t mask`)

Clears the pending events specified in the mask.

Parameters

in	<code>mask</code>	the events to be cleared
----	-------------------	--------------------------

Returns

The pending events that were cleared.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.2.4 eventmask_t chEvtAddFlags (eventmask_t mask)

Adds (OR) a set of event flags on the current thread, this is **much** faster than using [chEvtBroadcast\(\)](#) or [chEvtSignal\(\)](#).

Parameters

in *mask* the event flags to be ORed

Returns

The current pending events mask.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.2.5 void chEvtSignalFlags (Thread * tp, eventmask_t mask)

Adds (OR) a set of event flags on the specified [Thread](#).

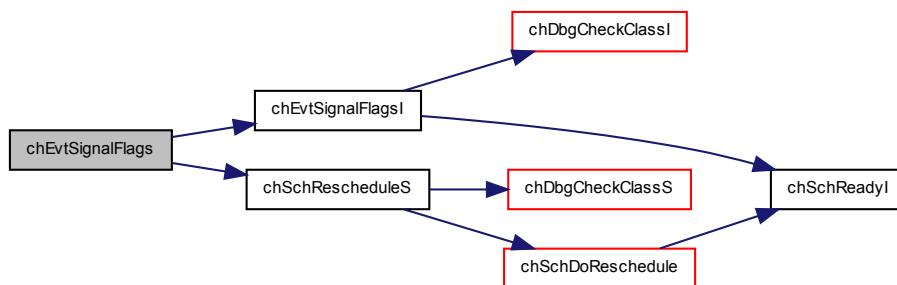
Parameters

in	<i>tp</i>	the thread to be signaled
in	<i>mask</i>	the event flags set to be ORed

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.6 void chEvtSignalFlagsI (Thread * tp, eventmask_t mask)

Adds (OR) a set of event flags on the specified [Thread](#).

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

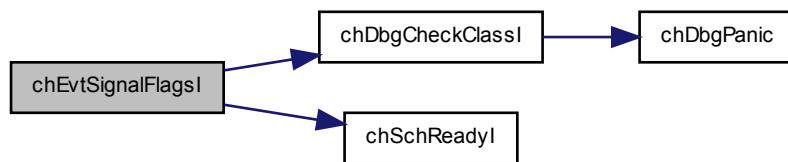
Parameters

in	<i>tp</i> the thread to be signaled
in	<i>mask</i> the event flags set to be ORed

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:

**7.15.2.7 void chEvtBroadcastFlags (EventSource * esp, eventmask_t mask)**

Signals all the Event Listeners registered on the specified Event Source.

This function variants ORs the specified event flags to all the threads registered on the `EventSource` in addition to the event flags specified by the threads themselves in the `EventListener` objects.

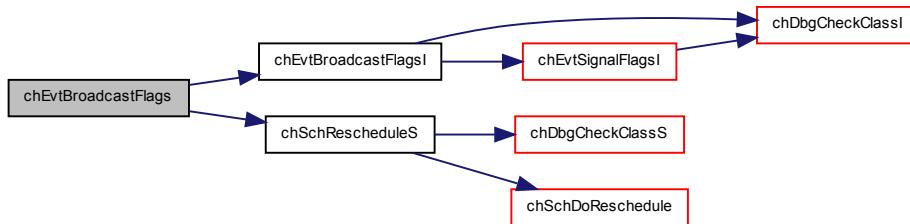
Parameters

in	<i>esp</i> pointer to the <code>EventSource</code> structure
in	<i>mask</i> the event flags set to be ORed

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.8 void chEvtBroadcastFlagsI (EventSource * esp, eventmask_t mask)

Signals all the Event Listeners registered on the specified Event Source.

This function variants ORs the specified event flags to all the threads registered on the `EventSource` in addition to the event flags specified by the threads themselves in the `EventListener` objects.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

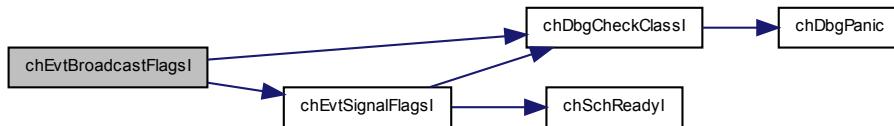
Parameters

in	<code>esp</code>	pointer to the <code>EventSource</code> structure
in	<code>mask</code>	the event flags set to be ORed

Function Class:

This is an **I-Class API**, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.15.2.9 void chEvtDispatch (const evhandler_t * handlers, eventmask_t mask)

Invokes the event handlers associated to an event flags mask.

Parameters

in	<code>mask</code>	mask of the event flags to be dispatched
in	<code>handlers</code>	an array of <code>evhandler_t</code> . The array must have size equal to the number of bits in <code>eventmask_t</code> .

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.2.10 eventmask_t chEvtWaitOneTimeout (eventmask_t mask, systime_t time)

Waits for exactly one of the specified events.

The function waits for one event among those specified in `mask` to become pending then the event is cleared and returned.

Note

One and only one event is served in the function, the one with the lowest event id. The function is meant to be

invoked into a loop in order to serve all the pending events.

This means that Event Listeners with a lower event identifier have an higher priority.

Parameters

in	<i>mask</i>	mask of the event flags that the function should wait for, ALL_EVENTS enables all the events
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

The mask of the lowest id served and cleared event.

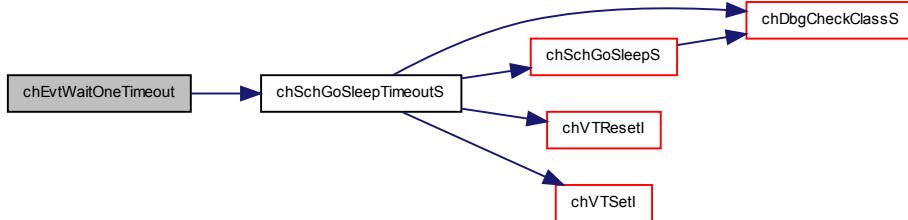
Return values

0 if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.11 eventmask_t chEvtWaitAnyTimeout (eventmask_t *mask*, systime_t *time*)

Waits for any of the specified events.

The function waits for any event among those specified in *mask* to become pending then the events are cleared and returned.

Parameters

in	<i>mask</i>	mask of the event flags that the function should wait for, ALL_EVENTS enables all the events
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

The mask of the served and cleared events.

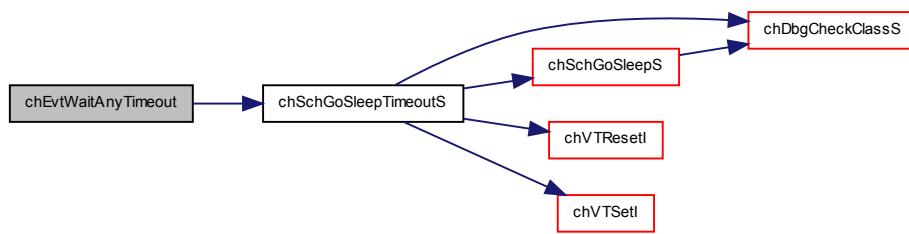
Return values

0 if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

**7.15.2.12 eventmask_t chEvtWaitAllTimeout(eventmask_t mask, systime_t time)**

Waits for all the specified events.

The function waits for all the events specified in `mask` to become pending then the events are cleared and returned.

Parameters

in	<code>mask</code>	mask of the event flags that the function should wait for, <code>ALL_EVENTS</code> requires all the events
in	<code>time</code>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <code>TIME_IMMEDIATE</code> immediate timeout. • <code>TIME_INFINITE</code> no timeout.

Returns

The mask of the served and cleared events.

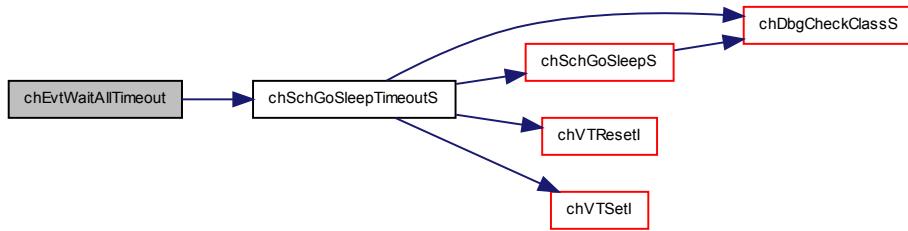
Return values

0 if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.13 eventmask_t chEvtWaitOne(eventmask_t mask)

Waits for exactly one of the specified events.

The function waits for one event among those specified in `mask` to become pending then the event is cleared and returned.

Note

One and only one event is served in the function, the one with the lowest event id. The function is meant to be invoked into a loop in order to serve all the pending events.

This means that Event Listeners with a lower event identifier have an higher priority.

Parameters

in	<code>mask</code> mask of the event flags that the function should wait for, <code>ALL_EVENTS</code> enables all the events
----	---

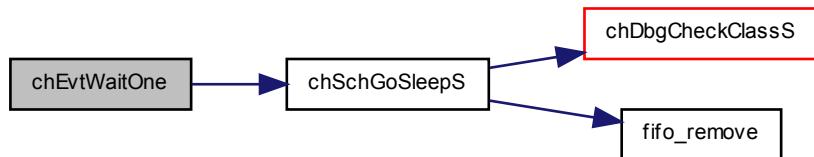
Returns

The mask of the lowest id served and cleared event.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.14 eventmask_t chEvtWaitAny(eventmask_t mask)

Waits for any of the specified events.

The function waits for any event among those specified in `mask` to become pending then the events are cleared and returned.

Parameters

in `mask` mask of the event flags that the function should wait for, `ALL_EVENTS` enables all the events

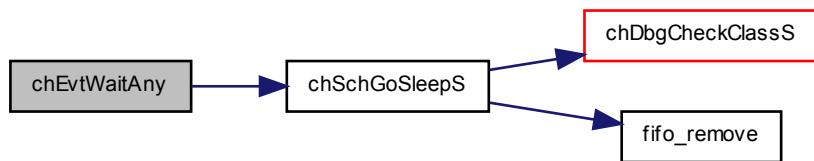
Returns

The mask of the served and cleared events.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.2.15 eventmask_t chEvtWaitAll(eventmask_t mask)

Waits for all the specified events.

The function waits for all the events specified in `mask` to become pending then the events are cleared and returned.

Parameters

in `mask` mask of the event flags that the function should wait for, `ALL_EVENTS` requires all the events

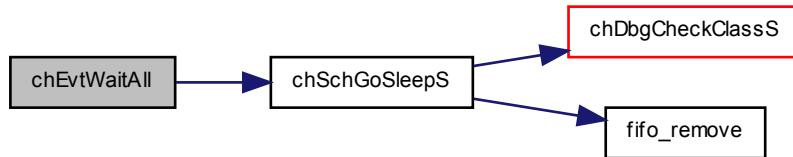
Returns

The mask of the served and cleared events.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.15.3 Define Documentation

7.15.3.1 `#define _EVENTSOURCE_DATA(name) {(void *)(&name)}`

Data part of a static event source initializer.

This macro should be used when statically initializing an event source that is part of a bigger structure.

Parameters

name the name of the event source variable

7.15.3.2 `#define EVENTSOURCE_DECL(name) EventSource name = _EVENTSOURCE_DATA(name)`

Static event source initializer.

Statically initialized event sources require no explicit initialization using `chEvtInit()`.

Parameters

name the name of the event source variable

7.15.3.3 `#define ALL_EVENTS ((eventmask_t)-1)`

All events allowed mask.

7.15.3.4 `#define EVENT_MASK(eid) ((eventmask_t)(1 << (eid)))`

Returns an event mask from an event identifier.

7.15.3.5 `#define chEvtRegister(esp, elp, eid) chEvtRegisterMask(esp, elp, EVENT_MASK(eid))`

Registers an Event Listener on an Event Source.

Note

Multiple Event Listeners can use the same event identifier, the listener will share the callback function.

Parameters

in *esp* pointer to the `EventSource` structure

out in	<p><i>elp</i> pointer to the EventListener structure</p> <p><i>eid</i> numeric identifier assigned to the Event Listener. The identifier is used as index for the event callback function. The value must range between zero and the size, in bit, of the <code>eventid_t</code> type minus one.</p>
-----------	--

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.3.6 `#define chEvtInit(esp) ((esp)->es_next = (EventListener *)(void *)(esp))`

Initializes an Event Source.

Note

This function can be invoked before the kernel is initialized because it just prepares a [EventSource](#) structure.

Parameters

in	<i>esp</i> pointer to the EventSource structure
----	---

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.15.3.7 `#define chEvtIsListening(esp) ((void *)(esp) != (void *)(esp)->es_next)`

Verifies if there is at least one [EventListener](#) registered.

Parameters

in	<i>esp</i> pointer to the EventSource structure
----	---

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.15.3.8 `#define chEvtBroadcast(esp) chEvtBroadcastFlags(esp, 0)`

Signals all the Event Listeners registered on the specified Event Source.

Parameters

in	<i>esp</i> pointer to the EventSource structure
----	---

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.15.3.9 `#define chEvtBroadcastI(esp) chEvtBroadcastFlagsI(esp, 0)`

Signals all the Event Listeners registered on the specified Event Source.

Postcondition

This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel. Note that interrupt handlers always reschedule on exit so an explicit reschedule must not be performed in ISRs.

Parameters

in `esp` pointer to the `EventSource` structure

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.15.4 Typedef Documentation

7.15.4.1 `typedef struct EventSource EventSource`

Event Source structure.

7.15.4.2 `typedef void(* evhandler_t)(eventid_t)`

Event Handler callback function.

7.16 Synchronous Messages

7.16.1 Detailed Description

Synchronous inter-thread messages APIs and services.

Operation Mode

Synchronous messages are an easy to use and fast IPC mechanism, threads can both act as message servers and/or message clients, the mechanism allows data to be carried in both directions. Note that messages are not copied between the client and server threads but just a pointer passed so the exchange is very time efficient.

Messages are scalar data types of type `msg_t` that are guaranteed to be size compatible with data pointers. Note that on some architectures function pointers can be larger than `msg_t`.

Messages are usually processed in FIFO order but it is possible to process them in priority order by enabling the `CH_USE_MESSAGES_PRIORITY` option in `chconf.h`.

Precondition

In order to use the message APIs the `CH_USE_MESSAGES` option must be enabled in `chconf.h`.

Postcondition

Enabling messages requires 6-12 (depending on the architecture) extra bytes in the `Thread` structure.

Functions

- `msg_t chMsgSend (Thread *tp, msg_t msg)`
Sends a message to the specified thread.
- `Thread * chMsgWait (void)`
Suspends the thread and waits for an incoming message.

- void [chMsgRelease](#) (Thread *tp, msg_t msg)
Releases a sender thread specifying a response message.

Macro Functions

- #define [chMsgIsPendingI](#)(tp) ((tp)->p_msgqueue.p_next != (Thread *)&(tp)->p_msgqueue)
Evaluates to TRUE if the thread has pending messages.
- #define [chMsgGet](#)(tp) ((tp)->p_msg)
Returns the message carried by the specified thread.
- #define [chMsgGetS](#)(tp) ((tp)->p_msg)
Returns the message carried by the specified thread.
- #define [chMsgReleaseS](#)(tp, msg) chSchWakeupS(tp, msg)
Releases the thread waiting on top of the messages queue.

7.16.2 Function Documentation

7.16.2.1 msg_t [chMsgSend](#) (Thread * tp, msg_t msg)

Sends a message to the specified thread.

The sender is stopped until the receiver executes a [chMsgRelease\(\)](#) after receiving the message.

Parameters

in	<i>tp</i>	the pointer to the thread
in	<i>msg</i>	the message

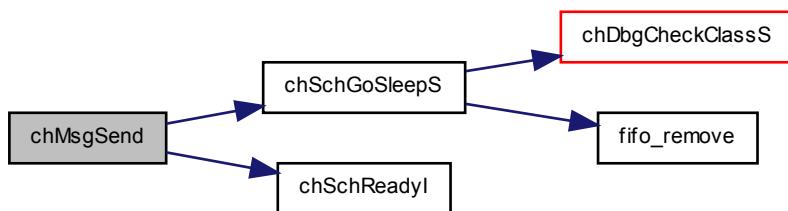
Returns

The answer message from [chMsgRelease\(\)](#).

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.16.2.2 Thread * [chMsgWait](#) (void)

Suspends the thread and waits for an incoming message.

Postcondition

After receiving a message the function `chMsgGet()` must be called in order to retrieve the message and then `chMsgRelease()` must be invoked in order to acknowledge the reception and send the answer.

Note

If the message is a pointer then you can assume that the data pointed by the message is stable until you invoke `chMsgRelease()` because the sending thread is suspended until then.

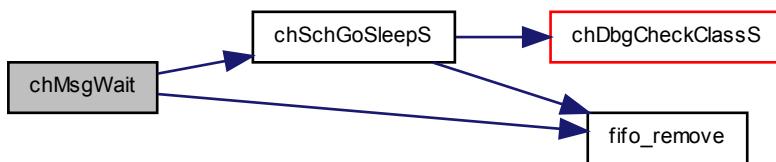
Returns

A reference to the thread carrying the message.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.16.2.3 void chMsgRelease (Thread * tp, msg_t msg)

Releases a sender thread specifying a response message.

Precondition

Invoke this function only after a message has been received using `chMsgWait()`.

Parameters

in	<i>tp</i>	pointer to the thread
in	<i>msg</i>	message to be returned to the sender

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.16.3 Define Documentation

7.16.3.1 #define chMsgIsPending(tp) ((tp)->p_msgqueue.p_next != (Thread *)(&(tp)->p_msgqueue))

Evaluates to TRUE if the thread has pending messages.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt

handlers.

7.16.3.2 #define chMsgGet(*tp*) ((*tp*)->p_msg)

Returns the message carried by the specified thread.

Precondition

This function must be invoked immediately after exiting a call to [chMsgWait\(\)](#).

Parameters

in *tp* pointer to the thread

Returns

The message carried by the sender.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.16.3.3 #define chMsgGetS(*tp*) ((*tp*)->p_msg)

Returns the message carried by the specified thread.

Precondition

This function must be invoked immediately after exiting a call to [chMsgWait\(\)](#).

Parameters

in *tp* pointer to the thread

Returns

The message carried by the sender.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.16.3.4 #define chMsgReleaseS(*tp*, *msg*) chSchWakeupS(*tp*, *msg*)

Releases the thread waiting on top of the messages queue.

Precondition

Invoke this function only after a message has been received using [chMsgWait\(\)](#).

Parameters

in *tp* pointer to the thread

in *msg* message to be returned to the sender

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

7.17 Mailboxes

7.17.1 Detailed Description

Asynchronous messages.

Operation mode

A mailbox is an asynchronous communication mechanism.

Operations defined for mailboxes:

- **Post**: Posts a message on the mailbox in FIFO order.
- **Post Ahead**: Posts a message on the mailbox with urgent priority.
- **Fetch**: A message is fetched from the mailbox and removed from the queue.
- **Reset**: The mailbox is emptied and all the stored messages are lost.

A message is a variable of type msg_t that is guaranteed to have the same size of and be compatible with (data) pointers (anyway an explicit cast is needed). If larger messages need to be exchanged then a pointer to a structure can be posted in the mailbox but the posting side has no predefined way to know when the message has been processed. A possible approach is to allocate memory (from a memory pool as example) from the posting side and free it on the fetching side. Another approach is to set a "done" flag into the structure pointed by the message.

Precondition

In order to use the mailboxes APIs the CH_USE_MAILBOXES option must be enabled in [chconf.h](#).

Data Structures

- struct [Mailbox](#)

Structure representing a mailbox object.

Functions

- void [chMBInit](#) ([Mailbox](#) *mbp, [msg_t](#) *buf, [cnt_t](#) n)
Initializes a [Mailbox](#) object.
- void [chMBReset](#) ([Mailbox](#) *mbp)
Resets a [Mailbox](#) object.
- [msg_t](#) [chMBPost](#) ([Mailbox](#) *mbp, [msg_t](#) msg, [systime_t](#) time)
Posts a message into a mailbox.
- [msg_t](#) [chMBPostS](#) ([Mailbox](#) *mbp, [msg_t](#) msg, [systime_t](#) time)
Posts a message into a mailbox.
- [msg_t](#) [chMBPostI](#) ([Mailbox](#) *mbp, [msg_t](#) msg)
Posts a message into a mailbox.
- [msg_t](#) [chMBPostAhead](#) ([Mailbox](#) *mbp, [msg_t](#) msg, [systime_t](#) time)
Posts an high priority message into a mailbox.
- [msg_t](#) [chMBPostAheadS](#) ([Mailbox](#) *mbp, [msg_t](#) msg, [systime_t](#) time)
Posts an high priority message into a mailbox.
- [msg_t](#) [chMBPostAheadI](#) ([Mailbox](#) *mbp, [msg_t](#) msg)
Posts an high priority message into a mailbox.
- [msg_t](#) [chMBFetch](#) ([Mailbox](#) *mbp, [msg_t](#) *msgp, [systime_t](#) time)
Retrieves a message from a mailbox.

- `msg_t chMBFetchS (Mailbox *mbp, msg_t *msgp, systime_t time)`
Retrieves a message from a mailbox.
- `msg_t chMBFetchl (Mailbox *mbp, msg_t *msgp)`
Retrieves a message from a mailbox.

Macro Functions

- `#define chMBSizel(mbp) ((mbp)->mb_top - (mbp)->mb_buffer)`
Returns the mailbox buffer size.
- `#define chMBGetFreeCountl(mbp) chSemGetCounterl(&(mbp)->mb_emptysem)`
Returns the number of free message slots into a mailbox.
- `#define chMBGetUsedCountl(mbp) chSemGetCounterl(&(mbp)->mb_fullsem)`
Returns the number of used message slots into a mailbox.
- `#define chMBPeekl(mbp) (*(mbp)->mb_rdptr)`
Returns the next message in the queue without removing it.

Defines

- `#define _MAILBOX_DATA(name, buffer, size)`
Data part of a static mailbox initializer.
- `#define MAILBOX_DECL(name, buffer, size) Mailbox name = _MAILBOX_DATA(name, buffer, size)`
Static mailbox initializer.

7.17.2 Function Documentation

7.17.2.1 void chMBInit (Mailbox * mbp, msg_t * buf, cnt_t n)

Initializes a `Mailbox` object.

Parameters

out	<code>mbp</code>	the pointer to the <code>Mailbox</code> structure to be initialized
in	<code>buf</code>	the circular messages buffer
in	<code>n</code>	the buffer size as number of <code>msg_t</code>

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

Here is the call graph for this function:



7.17.2.2 void chMBReset (Mailbox * mbp)

Resets a [Mailbox](#) object.

All the waiting threads are resumed with status `RDY_RESET` and the queued messages are lost.

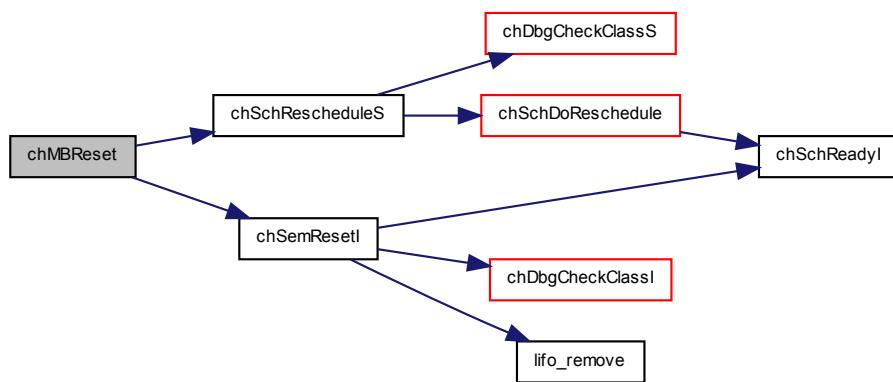
Parameters

`in` `mbp` the pointer to an initialized [Mailbox](#) object

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.17.2.3 msg_t chMBPost (Mailbox * mbp, msg_t msg, systime_t time)

Posts a message into a mailbox.

The invoking thread waits until a empty slot in the mailbox becomes available or the specified time runs out.

Parameters

`in` `mbp` the pointer to an initialized [Mailbox](#) object

`in` `msg` the message to be posted on the mailbox

`in` `time` the number of ticks before the operation timeouts, the following special values are allowed:

- `TIME_IMMEDIATE` immediate timeout.
- `TIME_INFINITE` no timeout.

Returns

The operation status.

Return values

`RDY_OK` if a message has been correctly posted.

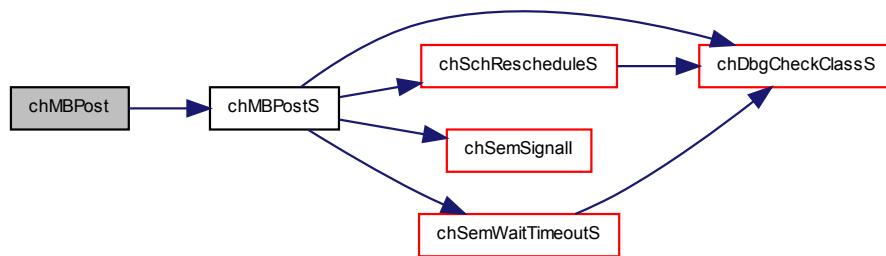
`RDY_RESET` if the mailbox has been reset while waiting.

`RDY_TIMEOUT` if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.17.2.4 msg_t chMBPostS (Mailbox * mbp, msg_t msg, systime_t time)

Posts a message into a mailbox.

The invoking thread waits until a empty slot in the mailbox becomes available or the specified time runs out.

Parameters

in	<i>mbp</i>	the pointer to an initialized Mailbox object
in	<i>msg</i>	the message to be posted on the mailbox
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

The operation status.

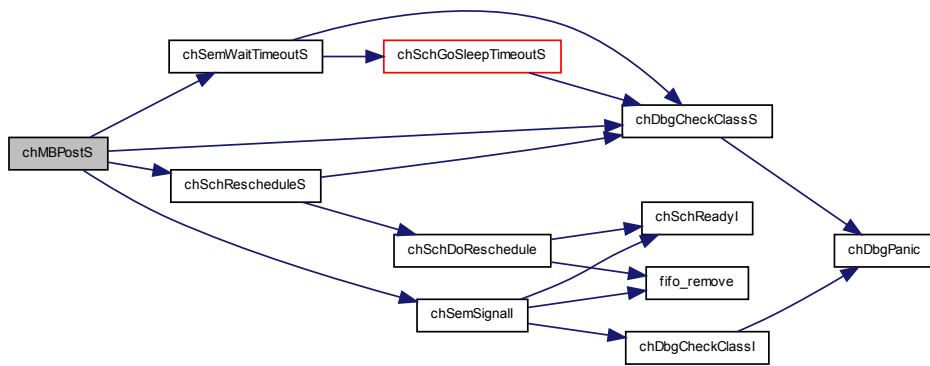
Return values

<i>RDY_OK</i>	if a message has been correctly posted.
<i>RDY_RESET</i>	if the mailbox has been reset while waiting.
<i>RDY_TIMEOUT</i>	if the operation has timed out.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.17.2.5 `msg_t chMBPostI(Mailbox * mbp, msg_t msg)`

Posts a message into a mailbox.

This variant is non-blocking, the function returns a timeout condition if the queue is full.

Parameters

<code>in</code>	<code>mbp</code>	the pointer to an initialized Mailbox object
<code>in</code>	<code>msg</code>	the message to be posted on the mailbox

Returns

The operation status.

Return values

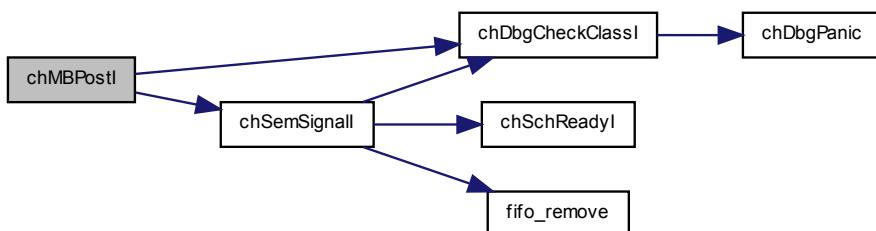
`RDY_OK` if a message has been correctly posted.

`RDY_TIMEOUT` if the mailbox is full and the message cannot be posted.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.17.2.6 `msg_t chMBPostAhead (Mailbox * mbp, msg_t msg, systime_t time)`

Posts an high priority message into a mailbox.

The invoking thread waits until a empty slot in the mailbox becomes available or the specified time runs out.

Parameters

in	<i>mbp</i>	the pointer to an initialized Mailbox object
in	<i>msg</i>	the message to be posted on the mailbox
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none">• <i>TIME_IMMEDIATE</i> immediate timeout.• <i>TIME_INFINITE</i> no timeout.

Returns

The operation status.

Return values

RDY_OK if a message has been correctly posted.

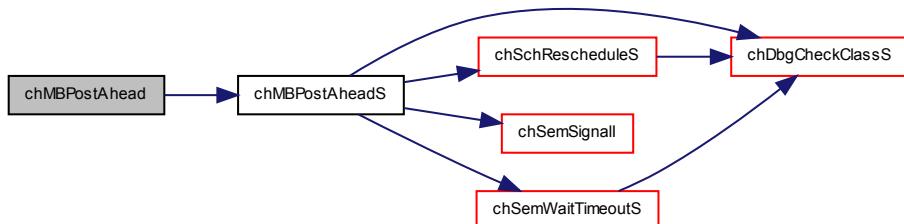
RDY_RESET if the mailbox has been reset while waiting.

RDY_TIMEOUT if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.17.2.7 `msg_t chMBPostAheadS (Mailbox * mbp, msg_t msg, systime_t time)`

Posts an high priority message into a mailbox.

The invoking thread waits until a empty slot in the mailbox becomes available or the specified time runs out.

Parameters

in	<i>mbp</i>	the pointer to an initialized Mailbox object
in	<i>msg</i>	the message to be posted on the mailbox
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none">• <i>TIME_IMMEDIATE</i> immediate timeout.• <i>TIME_INFINITE</i> no timeout.

Returns

The operation status.

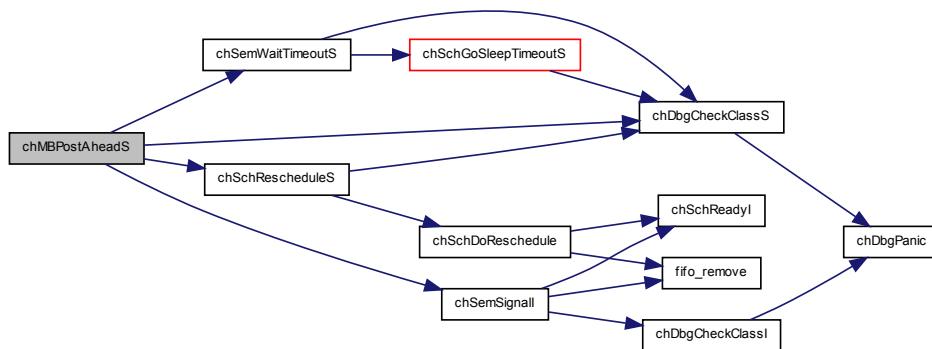
Return values

- RDY_OK* if a message has been correctly posted.
- RDY_RESET* if the mailbox has been reset while waiting.
- RDY_TIMEOUT* if the operation has timed out.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:

**7.17.2.8 msg_t chMBPostAheadI (Mailbox * mbp, msg_t msg)**

Posts an high priority message into a mailbox.

This variant is non-blocking, the function returns a timeout condition if the queue is full.

Parameters

- | | |
|----|---|
| in | <i>mbp</i> the pointer to an initialized Mailbox object |
| in | <i>msg</i> the message to be posted on the mailbox |

Returns

The operation status.

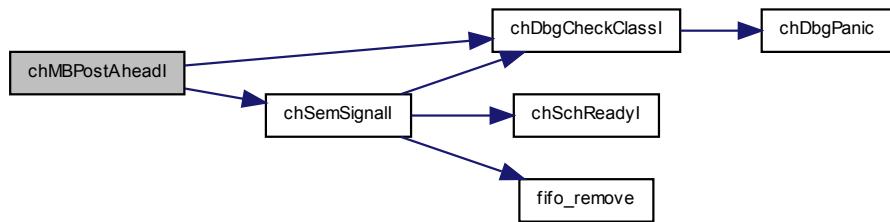
Return values

- RDY_OK* if a message has been correctly posted.
- RDY_TIMEOUT* if the mailbox is full and the message cannot be posted.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.17.2.9 `msg_t chMBFetch (Mailbox * mbp, msg_t * msgp, systime_t time)`

Retrieves a message from a mailbox.

The invoking thread waits until a message is posted in the mailbox or the specified time runs out.

Parameters

<code>in</code>	<code>mbp</code>	the pointer to an initialized Mailbox object
<code>out</code>	<code>msgp</code>	pointer to a message variable for the received message
<code>in</code>	<code>time</code>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <code>TIME_IMMEDIATE</code> immediate timeout. • <code>TIME_INFINITE</code> no timeout.

Returns

The operation status.

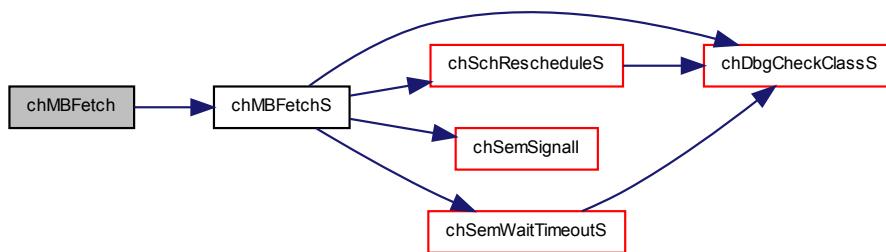
Return values

<code>RDY_OK</code>	if a message has been correctly fetched.
<code>RDY_RESET</code>	if the mailbox has been reset while waiting.
<code>RDY_TIMEOUT</code>	if the operation has timed out.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.17.2.10 `msg_t chMBFetchS (Mailbox * mbp, msg_t * msgp, systime_t time)`

Retrieves a message from a mailbox.

The invoking thread waits until a message is posted in the mailbox or the specified time runs out.

Parameters

in	<code>mbp</code>	the pointer to an initialized <code>Mailbox</code> object
out	<code>msgp</code>	pointer to a message variable for the received message
in	<code>time</code>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <code>TIME_IMMEDIATE</code> immediate timeout. • <code>TIME_INFINITE</code> no timeout.

Returns

The operation status.

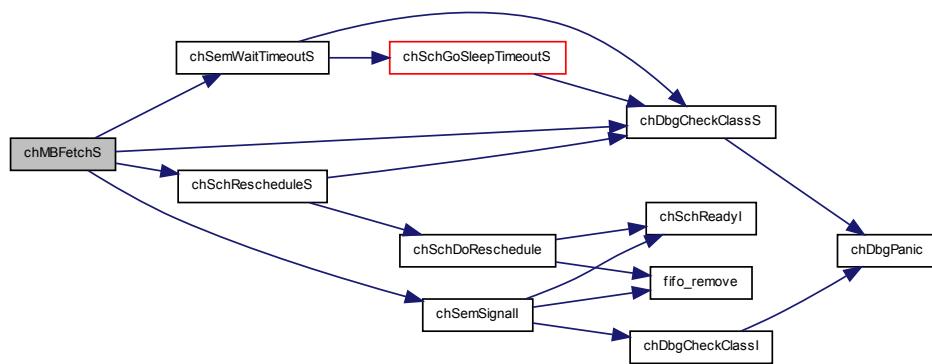
Return values

<code>RDY_OK</code>	if a message has been correctly fetched.
<code>RDY_RESET</code>	if the mailbox has been reset while waiting.
<code>RDY_TIMEOUT</code>	if the operation has timed out.

Function Class:

This is an **S-Class** API, this function can be invoked from within a system lock zone by threads only.

Here is the call graph for this function:



7.17.2.11 `msg_t chMBFetchl (Mailbox *mbp, msg_t *msgp)`

Retrieves a message from a mailbox.

This variant is non-blocking, the function returns a timeout condition if the queue is full.

Parameters

<code>in</code>	<code>mbp</code>	the pointer to an initialized <code>Mailbox</code> object
<code>out</code>	<code>msgp</code>	pointer to a message variable for the received message

Returns

The operation status.

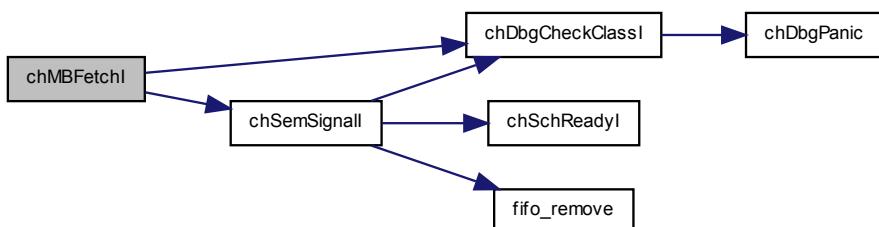
Return values

<code>RDY_OK</code>	if a message has been correctly fetched.
<code>RDY_TIMEOUT</code>	if the mailbox is empty and a message cannot be fetched.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.17.3 Define Documentation

7.17.3.1 #define chMBSizel(*mbp*) ((*mbp*)>mb_top - (*mbp*)>mb_buffer)

Returns the mailbox buffer size.

Parameters

in *mbp* the pointer to an initialized [Mailbox](#) object

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.17.3.2 #define chMBGetFreeCountl(*mbp*) chSemGetCounterl(&(*mbp*)>mb_emptysem)

Returns the number of free message slots into a mailbox.

Note

Can be invoked in any system state but if invoked out of a locked state then the returned value may change after reading.

The returned value can be less than zero when there are waiting threads on the internal semaphore.

Parameters

in *mbp* the pointer to an initialized [Mailbox](#) object

Returns

The number of empty message slots.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.17.3.3 #define chMBGetUsedCountl(*mbp*) chSemGetCounterl(&(*mbp*)>mb_fullsem)

Returns the number of used message slots into a mailbox.

Note

Can be invoked in any system state but if invoked out of a locked state then the returned value may change after reading.

The returned value can be less than zero when there are waiting threads on the internal semaphore.

Parameters

in *mbp* the pointer to an initialized [Mailbox](#) object

Returns

The number of queued messages.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.17.3.4 #define chMBPeekl(*mbp*) (*(mbp)->mb_rdptr)

Returns the next message in the queue without removing it.

Precondition

A message must be waiting in the queue for this function to work or it would return garbage. The correct way to use this macro is to use `chMBGetFullCount()` and then use this macro, all within a lock state.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.17.3.5 #define _MAILBOX_DATA(*name*, *buffer*, *size*)

Value:

```
{
    (msg_t *) (buffer), \
    (msg_t *) (buffer) + size, \
    (msg_t *) (buffer), \
    (msg_t *) (buffer), \
    _SEMAPHORE_DATA(name.mb_fullsem, 0), \
    _SEMAPHORE_DATA(name.mb_emptysem, size),
}
```

Data part of a static mailbox initializer.

This macro should be used when statically initializing a mailbox that is part of a bigger structure.

Parameters

in	<i>name</i>	the name of the mailbox variable
in	<i>buffer</i>	pointer to the mailbox buffer area
in	<i>size</i>	size of the mailbox buffer area

7.17.3.6 #define MAILBOX_DECL(*name*, *buffer*, *size*) Mailbox *name* = _MAILBOX_DATA(*name*, *buffer*, *size*)

Static mailbox initializer.

Statically initialized mailboxes require no explicit initialization using `chMBInit()`.

Parameters

in	<i>name</i>	the name of the mailbox variable
in	<i>buffer</i>	pointer to the mailbox buffer area
in	<i>size</i>	size of the mailbox buffer area

7.18 Memory Management

7.18.1 Detailed Description

Memory Management services.

Modules

- Core Memory Manager
- Heaps
- Memory Pools
- Dynamic Threads

7.19 Core Memory Manager

7.19.1 Detailed Description

Core Memory Manager related APIs and services.

Operation mode

The core memory manager is a simplified allocator that only allows to allocate memory blocks without the possibility to free them.

This allocator is meant as a memory blocks provider for the other allocators such as:

- C-Runtime allocator (through a compiler specific adapter module).
- Heap allocator (see [Heaps](#)).
- Memory pools allocator (see [Memory Pools](#)).

By having a centralized memory provider the various allocators can coexist and share the main memory.

This allocator, alone, is also useful for very simple applications that just require a simple way to get memory blocks.

Precondition

In order to use the core memory manager APIs the CH_USE_MEMCORE option must be enabled in [chconf.h](#).

Functions

- void [_core_init](#) (void)
Low level memory manager initialization.
- void * [chCoreAlloc](#) (size_t size)
Allocates a memory block.
- void * [chCoreAlloc1](#) (size_t size)
Allocates a memory block.
- size_t [chCoreStatus](#) (void)
Core memory status.

Alignment support macros

- #define [MEM_ALIGN_SIZE](#) sizeof([stkalign_t](#))
Alignment size constant.
- #define [MEM_ALIGN_MASK](#) ([MEM_ALIGN_SIZE](#) - 1)
Alignment mask constant.
- #define [MEM_ALIGN_PREV](#)(p) ((size_t)(p) & ~[MEM_ALIGN_MASK](#))
Alignment helper macro.
- #define [MEM_ALIGN_NEXT](#)(p) [MEM_ALIGN_PREV](#)((size_t)(p) + [MEM_ALIGN_MASK](#))

Alignment helper macro.

- `#define MEM_IS_ALIGNED(p) (((size_t)(p) & MEM_ALIGN_MASK) == 0)`

Returns whatever a pointer or memory size is aligned to the type align_t.

Typedefs

- `typedef void *(* memgetfunc_t)(size_t size)`

Memory get function.

7.19.2 Function Documentation

7.19.2.1 void _core_init(void)

Low level memory manager initialization.

Function Class:

Not an API, this function is for internal use only.

7.19.2.2 void * chCoreAlloc(size_t size)

Allocates a memory block.

The size of the returned block is aligned to the alignment type so it is not possible to allocate less than `MEM_ALIGN_SIZE`.

Parameters

in `size` the size of the block to be allocated

Returns

A pointer to the allocated memory block.

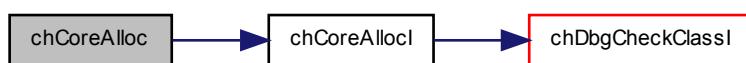
Return values

`NULL` allocation failed, core memory exhausted.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.19.2.3 void * chCoreAlloc(size_t size)

Allocates a memory block.

The size of the returned block is aligned to the alignment type so it is not possible to allocate less than `MEM_ALIGN_SIZE`.

Parameters

`in` `size` the size of the block to be allocated.

Returns

A pointer to the allocated memory block.

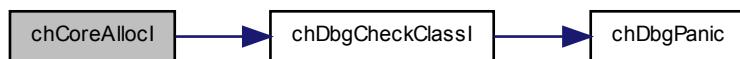
Return values

`NULL` allocation failed, core memory exhausted.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.19.2.4 size_t chCoreStatus(void)

Core memory status.

Returns

The size, in bytes, of the free core memory.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.19.3 Define Documentation

7.19.3.1 #define MEM_ALIGN_SIZE sizeof(stkalign_t)

Alignment size constant.

7.19.3.2 #define MEM_ALIGN_MASK (MEM_ALIGN_SIZE - 1)

Alignment mask constant.

7.19.3.3 `#define MEM_ALIGN_PREV(p) ((size_t)(p) & ~MEM_ALIGN_MASK)`

Alignment helper macro.

7.19.3.4 `#define MEM_ALIGN_NEXT(p) MEM_ALIGN_PREV((size_t)(p) + MEM_ALIGN_MASK)`

Alignment helper macro.

7.19.3.5 `#define MEM_IS_ALIGNED(p) (((size_t)(p) & MEM_ALIGN_MASK) == 0)`

Returns whatever a pointer or memory size is aligned to the type `align_t`.

7.19.4 Typedef Documentation

7.19.4.1 `typedef void*(* memgetfunc_t)(size_t size)`

Memory get function.

Note

This type must be assignment compatible with the `chMemAlloc()` function.

7.20 Heaps

7.20.1 Detailed Description

Heap Allocator related APIs.

Operation mode

The heap allocator implements a first-fit strategy and its APIs are functionally equivalent to the usual `malloc()` and `free()` library functions. The main difference is that the OS heap APIs are guaranteed to be thread safe.

By enabling the `CH_USE_MALLOC_HEAP` option the heap manager will use the runtime-provided `malloc()` and `free()` as backend for the heap APIs instead of the system provided allocator.

Precondition

In order to use the heap APIs the `CH_USE_HEAP` option must be enabled in `chconf.h`.

Data Structures

- union `heap_header`
Memory heap block header.
- struct `memory_heap`
Structure describing a memory heap.

Functions

- void `_heap_init` (void)
Initializes the default heap.
- void `chHeapInit` (`MemoryHeap` *heapp, void *buf, `size_t` size)

- `void * chHeapAlloc (MemoryHeap *heapp, size_t size)`

Initializes a memory heap from a static memory area.
- `Allocates a block of memory from the heap by using the first-fit algorithm.`
- `void chHeapFree (void *p)`

Frees a previously allocated memory block.
- `size_t chHeapStatus (MemoryHeap *heapp, size_t *sizep)`

Reports the heap status.

7.20.2 Function Documentation

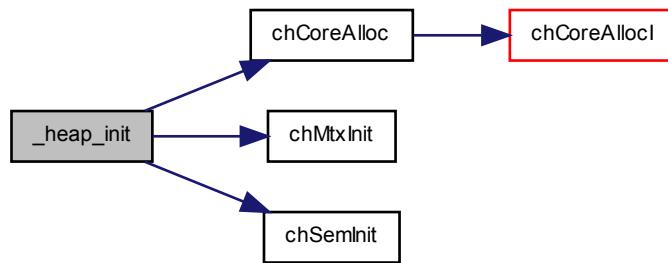
7.20.2.1 void _heap_init (void)

Initializes the default heap.

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.20.2.2 void chHeapInit (MemoryHeap * heapp, void * buf, size_t size)

Initializes a memory heap from a static memory area.

Precondition

Both the heap buffer base and the heap size must be aligned to the `stkalign_t` type size.
In order to use this function the option `CH_USE_MALLOC_HEAP` must be disabled.

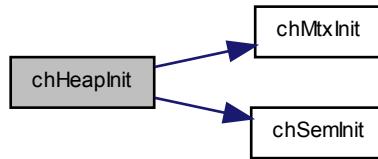
Parameters

<code>out</code>	<code>heapp</code>	pointer to the memory heap descriptor to be initialized
<code>in</code>	<code>buf</code>	heap buffer base
<code>in</code>	<code>size</code>	heap size

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

Here is the call graph for this function:



7.20.2.3 void * chHeapAlloc (MemoryHeap * heapp, size_t size)

Allocates a block of memory from the heap by using the first-fit algorithm.

The allocated block is guaranteed to be properly aligned for a pointer data type (stkalign_t).

Parameters

in	<i>heapp</i>	pointer to a heap descriptor or NULL in order to access the default heap.
in	<i>size</i>	the size of the block to be allocated. Note that the allocated block may be a bit bigger than the requested size for alignment and fragmentation reasons.

Returns

A pointer to the allocated block.

Return values

NULL if the block cannot be allocated.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.20.2.4 void chHeapFree (void * p)

Frees a previously allocated memory block.

Parameters

in	<i>p</i>	pointer to the memory block to be freed
----	----------	---

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.20.2.5 size_t chHeapStatus (MemoryHeap * heapp, size_t * sizep)

Reports the heap status.

Note

This function is meant to be used in the test suite, it should not be really useful for the application code.

This function is not implemented when the CH_USE_MALLOC_HEAP configuration option is used (it always returns zero).

Parameters

in	<i>heapp</i>	pointer to a heap descriptor or NULL in order to access the default heap.
in	<i>sizep</i>	pointer to a variable that will receive the total fragmented free space

Returns

The number of fragments in the heap.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.21 Memory Pools

7.21.1 Detailed Description

Memory Pools related APIs and services.

Operation mode

The Memory Pools APIs allow to allocate/free fixed size objects in **constant time** and reliably without memory fragmentation problems.

Precondition

In order to use the memory pools APIs the CH_USE_MEMPOOLS option must be enabled in `chconf.h`.

Data Structures

- struct `pool_header`
Memory pool tree object header.
- struct `MemoryPool`
Memory pool descriptor.

Functions

- void `chPoolInit (MemoryPool *mp, size_t size, memgetfunc_t provider)`
Initializes an empty memory pool.
- void * `chPoolAlloc (MemoryPool *mp)`
Allocates an object from a memory pool.
- void * `chPoolAlloc (MemoryPool *mp)`
Allocates an object from a memory pool.
- void `chPoolFree (MemoryPool *mp, void *objp)`
Releases (or adds) an object into (to) a memory pool.
- void `chPoolFree (MemoryPool *mp, void *objp)`
Releases (or adds) an object into (to) a memory pool.

Defines

- `#define _MEMORYPOOL_DATA(name, size, provider) {NULL, MEM_ALIGN_NEXT(size), provider}`
Data part of a static memory pool initializer.
- `#define MEMORYPOOL_DECL(name, size, provider) MemoryPool name = _MEMORYPOOL_DATA(name, size, provider)`
Static memory pool initializer in hungry mode.

7.21.2 Function Documentation

7.21.2.1 void chPoolInit (MemoryPool * mp, size_t size, memgetfunc_t provider)

Initializes an empty memory pool.

Note

The size is internally aligned to be a multiple of the `stkalign_t` type size.

Parameters

out	<code>mp</code>	pointer to a <code>MemoryPool</code> structure
in	<code>size</code>	the size of the objects contained in this memory pool, the minimum accepted size is the size of a pointer to void.
in	<code>provider</code>	memory provider function for the memory pool or <code>NULL</code> if the pool is not allowed to grow automatically

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.21.2.2 void * chPoolAlloc (MemoryPool * mp)

Allocates an object from a memory pool.

Parameters

in	<code>mp</code>	pointer to a <code>MemoryPool</code> structure
----	-----------------	--

Returns

The pointer to the allocated object.

Return values

`NULL` if pool is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.21.2.3 void * chPoolAlloc (MemoryPool * mp)

Allocates an object from a memory pool.

Parameters

in	<i>mp</i> pointer to a MemoryPool structure
----	---

Returns

The pointer to the allocated object.

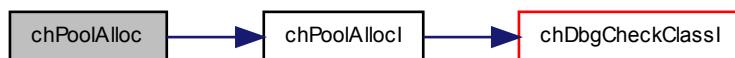
Return values

NULL if pool is empty.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.21.2.4 void chPoolFree (MemoryPool * mp, void * objp)

Releases (or adds) an object into (to) a memory pool.

Precondition

The freed object must be of the right size for the specified memory pool.

The freed object must be memory aligned to the size of `stkalign_t` type.

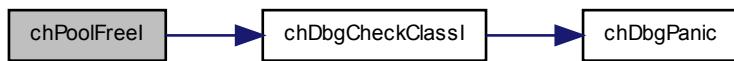
Parameters

in	<i>mp</i> pointer to a MemoryPool structure
in	<i>objp</i> the pointer to the object to be released or added

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.21.2.5 void chPoolFree (MemoryPool * mp, void * objp)

Releases (or adds) an object into (to) a memory pool.

Precondition

The freed object must be of the right size for the specified memory pool.

The freed object must be memory aligned to the size of `stkalign_t` type.

Parameters

in	<i>mp</i>	pointer to a <code>MemoryPool</code> structure
in	<i>objp</i>	the pointer to the object to be released or added

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.21.3 Define Documentation

7.21.3.1 #define _MEMORYPOOL_DATA(name, size, provider) {NULL, MEM_ALIGN_NEXT(size), provider}

Data part of a static memory pool initializer.

This macro should be used when statically initializing a memory pool that is part of a bigger structure.

Parameters

in	<i>name</i>	the name of the memory pool variable
----	-------------	--------------------------------------

in	<i>size</i>	size of the memory pool contained objects
in	<i>provider</i>	memory provider function for the memory pool

```
7.21.3.2 #define MEMORYPOOL_DECL( name, size, provider ) MemoryPool name = _MEMORYPOOL_DATA(name, size,  
                          provider)
```

Static memory pool initializer in hungry mode.

Statically initialized memory pools require no explicit initialization using [chPoolInit\(\)](#).

Parameters

in	<i>name</i>	the name of the memory pool variable
in	<i>size</i>	size of the memory pool contained objects
in	<i>provider</i>	memory provider function for the memory pool or NULL if the pool is not allowed to grow automatically

7.22 Dynamic Threads

7.22.1 Detailed Description

Dynamic threads related APIs and services.

Functions

- [Thread * chThdAddRef \(Thread *tp\)](#)
Adds a reference to a thread object.
- [void chThdRelease \(Thread *tp\)](#)
Releases a reference to a thread object.
- [Thread * chThdCreateFromHeap \(MemoryHeap *heapp, size_t size, tprio_t prio, tfunc_t pf, void *arg\)](#)
Creates a new thread allocating the memory from the heap.
- [Thread * chThdCreateFromMemoryPool \(MemoryPool *mp, tprio_t prio, tfunc_t pf, void *arg\)](#)
Creates a new thread allocating the memory from the specified memory pool.

7.22.2 Function Documentation

7.22.2.1 Thread * chThdAddRef (Thread * *tp*)

Adds a reference to a thread object.

Precondition

The configuration option CH_USE_DYNAMIC must be enabled in order to use this function.

Parameters

in	<i>tp</i>	pointer to the thread
----	-----------	-----------------------

Returns

The same thread pointer passed as parameter representing the new reference.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.22.2.2 void chThdRelease (Thread * tp)

Releases a reference to a thread object.

If the references counter reaches zero **and** the thread is in the THD_STATE_FINAL state then the thread's memory is returned to the proper allocator.

Precondition

The configuration option CH_USE_DYNAMIC must be enabled in order to use this function.

Note

Static threads are not affected.

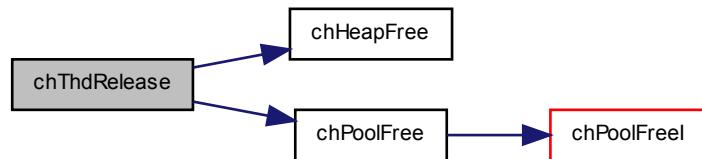
Parameters

in *tp* pointer to the thread

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

**7.22.2.3 Thread *chThdCreateFromHeap (MemoryHeap * heapp, size_t size, tprio_t prio, tfunc_t pf, void * arg)**

Creates a new thread allocating the memory from the heap.

Precondition

The configuration options CH_USE_DYNAMIC and CH_USE_HEAP must be enabled in order to use this function.

Note

A thread can terminate by calling [chThdExit \(\)](#) or by simply returning from its main function.

The memory allocated for the thread is not released when the thread terminates but when a [chThdWait \(\)](#) is performed.

Parameters

in *heapp* heap from which allocate the memory or NULL for the default heap

in	<i>size</i>	size of the working area to be allocated
in	<i>prio</i>	the priority level for the new thread
in	<i>pf</i>	the thread function
in	<i>arg</i>	an argument passed to the thread function. It can be NULL.

Returns

The pointer to the [Thread](#) structure allocated for the thread into the working space area.

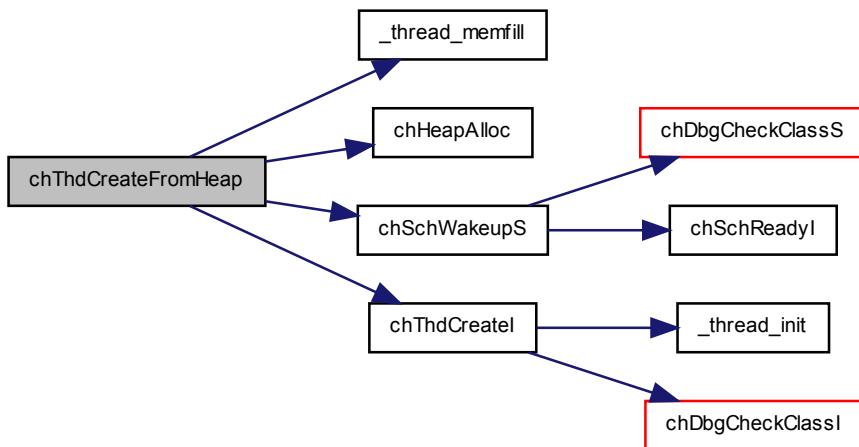
Return values

NULL if the memory cannot be allocated.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.22.2.4 Thread *chThdCreateFromMemoryPool (MemoryPool *mp, tprio_t prio, tfunc_t pf, void *arg)

Creates a new thread allocating the memory from the specified memory pool.

Precondition

The configuration options `CH_USE_DYNAMIC` and `CH_USE_MEMPOOLS` must be enabled in order to use this function.

Note

A thread can terminate by calling `chThdExit()` or by simply returning from its main function.

The memory allocated for the thread is not released when the thread terminates but when a `chThdWait()` is performed.

Parameters

in	<i>mp</i>	pointer to the memory pool object
----	-----------	-----------------------------------

in	<i>prio</i>	the priority level for the new thread
in	<i>pf</i>	the thread function
in	<i>arg</i>	an argument passed to the thread function. It can be NULL.

Returns

The pointer to the [Thread](#) structure allocated for the thread into the working space area.

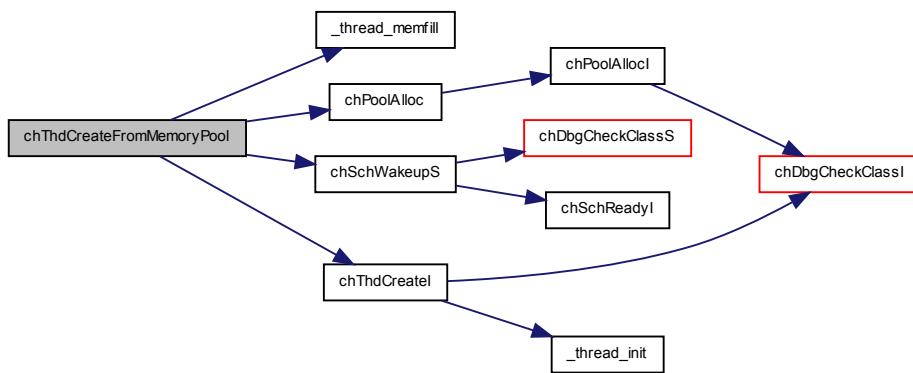
Return values

NULL if the memory pool is empty.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.23 I/O Support

7.23.1 Detailed Description

I/O related services.

Modules

- [Abstract Sequential Streams](#)
- [Abstract File Streams](#)
- [Abstract I/O Channels](#)
- [I/O Queues](#)

7.24 Abstract Sequential Streams

7.24.1 Detailed Description

This module define an abstract interface for generic data streams. Note that no code is present, streams are just abstract interfaces like structures, you should look at the systems as to a set of abstract C++ classes (even if written

in C). This system has the advantage to make the access to streams independent from the implementation logic. The stream interface can be used as base class for high level object types such as files, sockets, serial ports, pipes etc.

Data Structures

- struct **BaseSequentialStreamVMT**
BaseSequentialStream virtual methods table.
- struct **BaseSequentialStream**
Base stream class.

Macro Functions (**BaseSequentialStream**)

- #define **chSequentialStreamWrite**(ip, bp, n) ((ip)->vmt->write(ip, bp, n))
Sequential Stream write.
- #define **chSequentialStreamRead**(ip, bp, n) ((ip)->vmt->read(ip, bp, n))
Sequential Stream read.

Defines

- #define **_base_sequential_stream_methods**
BaseSequentialStream specific methods.
- #define **_base_sequential_stream_data**
BaseSequentialStream specific data.

7.24.2 Define Documentation

7.24.2.1 #define **_base_sequential_stream_methods**

Value:

```
/* Stream write buffer method.*/
size_t (*write)(void *instance, const uint8_t *bp, size_t n);           \
/* Stream read buffer method.*/
size_t (*read)(void *instance, uint8_t *bp, size_t n);                      \
```

BaseSequentialStream specific methods.

7.24.2.2 #define **_base_sequential_stream_data**

BaseSequentialStream specific data.

Note

It is empty because **BaseSequentialStream** is only an interface without implementation.

7.24.2.3 #define **chSequentialStreamWrite(ip, bp, n)** ((ip)->vmt->write(ip, bp, n))

Sequential Stream write.

The function writes data from a buffer to a stream.

Parameters

in	<i>ip</i>	pointer to a BaseSequentialStream or derived class
in	<i>bp</i>	pointer to the data buffer
in	<i>n</i>	the maximum amount of data to be transferred

Returns

The number of bytes transferred. The return value can be less than the specified number of bytes if the stream reaches a physical end of file and cannot be extended.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.24.2.4 #define chSequentialStreamRead(*ip*, *bp*, *n*) ((*ip*)>vmt->read(*ip*, *bp*, *n*))

Sequential Stream read.

The function reads data from a stream into a buffer.

Parameters

in	<i>ip</i>	pointer to a BaseSequentialStream or derived class
out	<i>bp</i>	pointer to the data buffer
in	<i>n</i>	the maximum amount of data to be transferred

Returns

The number of bytes transferred. The return value can be less than the specified number of bytes if the stream reaches the end of the available data.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25 Abstract File Streams

7.25.1 Detailed Description

This module define an abstract interface for generic data files by extending the [BaseSequentialStream](#) interface. Note that no code is present, data files are just abstract interface-like structures, you should look at the systems as to a set of abstract C++ classes (even if written in C). This system has the advantage to make the access to streams independent from the implementation logic.

The data files interface can be used as base class for high level object types such as an API for a File System implementation.

Data Structures

- struct [BaseFileStreamVMT](#)
BaseFileStream virtual methods table.
- struct [BaseFileStream](#)
Base file stream class.

Macro Functions ([BaseFileStream](#))

- #define [chFileStreamClose](#)(*ip*) ((*ip*)>vmt->close(*ip*))

Base file Stream close.

- **#define chFileStreamGetError(ip)** ((ip)->vmt->geterror(ip))
Returns an implementation dependent error code.
 - **#define chFileStreamGetSize(ip)** ((ip)->vmt->getposition(ip))
Returns the current file size.
 - **#define chFileStreamGetPosition(ip)** ((ip)->vmt->getposition(ip))
Returns the current file pointer position.
 - **#define chFileStreamSeek(ip, offset)** ((ip)->vmt->lseek(ip, offset))
Moves the file current pointer to an absolute position.

Defines

- `#define FILE_OK 0`
No error return code.
 - `#define FILE_ERROR 0xFFFFFFFFUL`
Error code from the file stream methods.
 - `#define _base_file_stream_methods`
BaseFileStream specific methods.
 - `#define _base_file_stream_data _base_sequential_stream_data`
BaseFileStream specific data.

TypeDefs

- `typedef uint32_t fileoffset_t`

File offset type.

7.25.2 Define Documentation

7.25.2.1 #define FILE_OK 0

No error return code.

7.25.2.2 #define FILE_ERROR 0xFFFFFFFFUL

Error code from the file stream methods.

7.25.2.3 #define base file stream methods

Value:

```
_base_sequential_stream_methods
/* File close method.*/
uint32_t (*close)(void *instance);
/* Get last error code method.*/
int (*geterror)(void *instance);
/* File get size method.*/
fileoffset_t (*getsize)(void *instance);
/* File get current position method.*/
fileoffset_t (*getposition)(void *instance);
/* File seek method.*/
uint32_t (*lseek)(void *instance, fileoffset_t offset);
```

[BaseFileStream](#) specific methods.

7.25.2.4 #define _base_file_stream_data _base_sequential_stream_data

BaseFileStream specific data.

Note

It is empty because BaseFileStream is only an interface without implementation.

7.25.2.5 #define chFileStreamClose(*ip*) ((*ip*)->vmt->close(*ip*))

Base file Stream close.

The function closes a file stream.

Parameters

in *ip* pointer to a BaseFileStream or derived class

Returns

The operation status.

Return values

<i>FILE_OK</i>	no error.
<i>FILE_ERROR</i>	operation failed.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25.2.6 #define chFileStreamGetError(*ip*) ((*ip*)->vmt->geterror(*ip*))

Returns an implementation dependent error code.

Parameters

in *ip* pointer to a BaseFileStream or derived class

Returns

Implementation dependent error code.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25.2.7 #define chFileStreamGetSize(*ip*) ((*ip*)->vmt->getposition(*ip*))

Returns the current file size.

Parameters

in *ip* pointer to a BaseFileStream or derived class

Returns

The file size.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25.2.8 #define chFileStreamGetPosition(*ip*) ((ip)->vmt->getposition(ip))

Returns the current file pointer position.

Parameters

in *ip* pointer to a [BaseFileStream](#) or derived class

Returns

The current position inside the file.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25.2.9 #define chFileStreamSeek(*ip*, *offset*) ((ip)->vmt->lseek(ip, offset))

Moves the file current pointer to an absolute position.

Parameters

in *ip* pointer to a [BaseFileStream](#) or derived class
in *offset* new absolute position

Returns

The operation status.

Return values

FILE_OK no error.
FILE_ERROR operation failed.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.25.3 Typedef Documentation

7.25.3.1 `typedef uint32_t fileoffset_t`

File offset type.

7.26 Abstract I/O Channels

7.26.1 Detailed Description

This module defines an abstract interface for I/O channels by extending the [BaseSequentialStream](#) interface. Note that no code is present, I/O channels are just abstract interface like structures, you should look at the systems as to a set of abstract C++ classes (even if written in C). Specific device drivers can use/extend the interface and implement them.

This system has the advantage to make the access to channels independent from the implementation logic.

Data Structures

- struct **BaseChannelVMT**
BaseChannel virtual methods table.
- struct **BaseChannel**
Base channel class.
- struct **BaseAsynchronousChannelVMT**
BaseAsynchronousChannel virtual methods table.
- struct **BaseAsynchronousChannel**
Base asynchronous channel class.

Macro Functions (BaseChannel)

- #define **chIOPutWouldBlock(ip)** ((ip)->vmt->putwouldblock(ip))
Channel output check.
- #define **chIOGetWouldBlock(ip)** ((ip)->vmt->getwouldblock(ip))
Channel input check.
- #define **chIOPut(ip, b)** ((ip)->vmt->put(ip, b, TIME_INFINITE))
Channel blocking byte write.
- #define **chIOPutTimeout(ip, b, time)** ((ip)->vmt->put(ip, b, time))
Channel blocking byte write with timeout.
- #define **chIOGet(ip)** ((ip)->vmt->get(ip, TIME_INFINITE))
Channel blocking byte read.
- #define **chIOGetTimeout(ip, time)** ((ip)->vmt->get(ip, time))
Channel blocking byte read with timeout.
- #define **chIOWriteTimeout(ip, bp, n, time)** ((ip)->vmt->writet(ip, bp, n, time))
Channel blocking write with timeout.
- #define **chIORReadTimeout(ip, bp, n, time)** ((ip)->vmt->readt(ip, bp, n, time))
Channel blocking read with timeout.

I/O status flags

- #define **IO_NO_ERROR** 0
No pending conditions.
- #define **IO_CONNECTED** 1
Connection happened.
- #define **IO_DISCONNECTED** 2
Disconnection happened.
- #define **IO_INPUT_AVAILABLE** 4
Data available in the input queue.
- #define **IO_OUTPUT_EMPTY** 8
Output queue empty.
- #define **IO_TRANSMISSION_END** 16
Transmission end.

Macro Functions (BaseAsynchronousChannel)

- #define **chIOGetEventSource(ip)** (&((ip)->event))
Returns the I/O condition event source.
- #define **chIOAddFlags1(ip, mask)**
Adds status flags to the channel's mask.
- #define **chIOGetAndClearFlags(ip)** ((ip)->vmt->getflags(ip))
Returns and clears the status flags associated to the channel.

Defines

- `#define _base_channel_methods`
`BaseChannel specific methods.`
- `#define _base_channel_data _base_sequential_stream_data`
`BaseChannel specific data.`
- `#define _base_asynchronous_channel_methods`
`BaseAsynchronousChannel specific methods.`
- `#define _base_asynchronous_channel_data`
`BaseAsynchronousChannel specific data.`
- `#define _ch_get_and_clear_flags_impl(ip)`
`Default implementation of the getflags virtual method.`

Typedefs

- `typedef uint_fast16_t ioflags_t`
`Type of an I/O condition flags mask.`

7.26.2 Define Documentation

7.26.2.1 #define _base_channel_methods

Value:

```
_base_sequential_stream_methods
/* Channel output check.*/
bool_t (*putwouldblock)(void *instance);
/* Channel input check.*/
bool_t (*getwouldblock)(void *instance);
/* Channel put method with timeout specification.*/
msg_t (*put)(void *instance, uint8_t b, systime_t time);
/* Channel get method with timeout specification.*/
msg_t (*get)(void *instance, systime_t time);
/* Channel write method with timeout specification.*/
size_t (*writet)(void *instance, const uint8_t *bp,
                  size_t n, systime_t time);
/* Channel read method with timeout specification.*/
size_t (*readt)(void *instance, uint8_t *bp, size_t n, systime_t time);
```

`BaseChannel` specific methods.

7.26.2.2 #define _base_channel_data _base_sequential_stream_data

`BaseChannel` specific data.

Note

It is empty because `BaseChannel` is only an interface without implementation.

7.26.2.3 #define chIOPutWouldBlock(ip) ((ip)->vmt->putwouldblock(ip))

Channel output check.

This function verifies if a subsequent put/write operation would block.

Parameters

in	<code>ip</code> pointer to a <code>BaseChannel</code> or derived class
----	--

Returns

The output queue status.

Return values

FALSE if the output queue has space and would not block a write operation.

TRUE if the output queue is full and would block a write operation.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.4 #define chIOGetWouldBlock(*ip*) ((*ip*)->vmt->getwouldblock(*ip*))

Channel input check.

This function verifies if a subsequent get/read operation would block.

Parameters

in *ip* pointer to a [BaseChannel](#) or derived class

Returns

The input queue status.

Return values

FALSE if the input queue contains data and would not block a read operation.

TRUE if the input queue is empty and would block a read operation.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.5 #define chIOPut(*ip*, *b*) ((*ip*)->vmt->put(*ip*, *b*, TIME_INFINITE))

Channel blocking byte write.

This function writes a byte value to a channel. If the channel is not ready to accept data then the calling thread is suspended.

Parameters

in *ip* pointer to a [BaseChannel](#) or derived class
in *b* the byte value to be written to the channel

Returns

The operation status.

Return values

Q_OK if the operation succeeded.

Q_RESET if the channel associated queue (if any) was reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.6 #define chIOPutTimeout(ip, b, time) ((ip)->vmt->put(ip, b, time))

Channel blocking byte write with timeout.

This function writes a byte value to a channel. If the channel is not ready to accept data then the calling thread is suspended.

Parameters

in	<i>ip</i>	pointer to a BaseChannel or derived class
in	<i>b</i>	the byte value to be written to the channel
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

The operation status.

Return values

<i>Q_OK</i>	if the operation succeeded.
<i>Q_TIMEOUT</i>	if the specified time expired.
<i>Q_RESET</i>	if the channel associated queue (if any) was reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.7 #define chIOGet(ip) ((ip)->vmt->get(ip, TIME_INFINITE))

Channel blocking byte read.

This function reads a byte value from a channel. If the data is not available then the calling thread is suspended.

Parameters

in	<i>ip</i>	pointer to a BaseChannel or derived class
----	-----------	---

Returns

A byte value from the queue.

Return values

<i>Q_RESET</i>	if the channel associated queue (if any) has been reset.
----------------	--

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.8 #define chIOGetTimeout(ip, time) ((ip)->vmt->get(ip, time))

Channel blocking byte read with timeout.

This function reads a byte value from a channel. If the data is not available then the calling thread is suspended.

Parameters

in	<i>ip</i>	pointer to a BaseChannel or derived class
----	-----------	---

in	<i>time</i> the number of ticks before the operation timeouts, the following special values are allowed:
	<ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

A byte value from the queue.

Return values

<i>Q_TIMEOUT</i>	if the specified time expired.
<i>Q_RESET</i>	if the channel associated queue (if any) has been reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.9 #define chIOWriteTimeout(*ip*, *bp*, *n*, *time*) ((*ip*)->vmt->writet(*ip*, *bp*, *n*, *time*))

Channel blocking write with timeout.

The function writes data from a buffer to a channel. If the channel is not ready to accept data then the calling thread is suspended.

Parameters

in	<i>ip</i> pointer to a BaseChannel or derived class
out	<i>bp</i> pointer to the data buffer
in	<i>n</i> the maximum amount of data to be transferred
in	<i>time</i> the number of ticks before the operation timeouts, the following special values are allowed:
	<ul style="list-style-type: none"> • <i>TIME_IMMEDIATE</i> immediate timeout. • <i>TIME_INFINITE</i> no timeout.

Returns

The number of bytes transferred.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.10 #define chIORReadTimeout(*ip*, *bp*, *n*, *time*) ((*ip*)->vmt->readt(*ip*, *bp*, *n*, *time*))

Channel blocking read with timeout.

The function reads data from a channel into a buffer. If the data is not available then the calling thread is suspended.

Parameters

in	<i>ip</i> pointer to a BaseChannel or derived class
in	<i>bp</i> pointer to the data buffer
in	<i>n</i> the maximum amount of data to be transferred

in *time* the number of ticks before the operation timeouts, the following special values are allowed:

- *TIME_IMMEDIATE* immediate timeout.
- *TIME_INFINITE* no timeout.

Returns

The number of bytes transferred.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.11 #define IO_NO_ERROR 0

No pending conditions.

7.26.2.12 #define IO_CONNECTED 1

Connection happened.

7.26.2.13 #define IO_DISCONNECTED 2

Disconnection happened.

7.26.2.14 #define IO_INPUT_AVAILABLE 4

Data available in the input queue.

7.26.2.15 #define IO_OUTPUT_EMPTY 8

Output queue empty.

7.26.2.16 #define IO_TRANSMISSION_END 16

Transmission end.

7.26.2.17 #define _base_asynchronous_channel_methods**Value:**

```
_base_channel_methods
/* Channel read method with timeout specification.*/
    ioflags_t (*getflags)(void *instance);
```

[BaseAsynchronousChannel](#) specific methods.

7.26.2.18 #define _base_asynchronous_channel_data

Value:

```
_base_channel_data
/* I/O condition event source.*/
EventSource          event;
/* I/O condition flags.*/
ioflags_t            flags;
```

BaseAsynchronousChannel specific data.

7.26.2.19 #define chIOGetEventSource(ip) (&((ip)->event))

Returns the I/O condition event source.

The event source is broadcasted when an I/O condition happens.

Parameters

in	<i>ip</i> pointer to a BaseAsynchronousChannel or derived class
----	---

Returns

A pointer to an EventSource object.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.20 #define chIOAddFlags(ip, mask)

Value:

```
{
    (ip)->flags |= (mask);
    chEvtBroadcastI(&(ip)->event);
}
```

Adds status flags to the channel's mask.

This function is usually called from the I/O ISTs in order to notify I/O conditions such as data events, errors, signal changes etc.

Parameters

in	<i>ip</i> pointer to a BaseAsynchronousChannel or derived class
in	<i>mask</i> condition flags to be added to the mask

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.26.2.21 #define chIOGetAndClearFlags(ip) ((ip)->vmt->getflags(ip))

Returns and clears the status flags associated to the channel.

Parameters

in	<i>ip</i> pointer to a BaseAsynchronousChannel or derived class
----	---

Returns

The condition flags modified since last time this function was invoked.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.26.2.22 #define _ch_get_and_clear_flags_impl(ip)**Value:**

```
ioflags_t mask;
chSysLock();
mask = ((BaseAsynchronousChannel *) (ip)) ->flags;
((BaseAsynchronousChannel *) (ip)) ->flags = IO_NO_ERROR;
chSysUnlock();
return mask
```

Default implementation of the `getflags` virtual method.

Parameters

in	<i>ip</i> pointer to a BaseAsynchronousChannel or derived class
----	---

Returns

The condition flags modified since last time this function was invoked.

Function Class:

Not an API, this function is for internal use only.

7.26.3 Typedef Documentation**7.26.3.1 typedef uint_fast16_t ioflags_t**

Type of an I/O condition flags mask.

7.27 I/O Queues**7.27.1 Detailed Description**

ChibiOS/RT queues are mostly used in serial-like device drivers. The device drivers are usually designed to have a lower side (lower driver, it is usually an interrupt service routine) and an upper side (upper driver, accessed by the application threads).

There are several kind of queues:

- **Input queue**, unidirectional queue where the writer is the lower side and the reader is the upper side.
- **Output queue**, unidirectional queue where the writer is the upper side and the reader is the lower side.
- **Full duplex queue**, bidirectional queue. Full duplex queues are implemented by pairing an input queue and an output queue together.

I/O queues are usually used as an implementation layer for the I/O channels interface, also see [Abstract I/O Channels](#).

Precondition

In order to use the I/O queues the `CH_USE_QUEUES` option must be enabled in `chconf.h`.

Data Structures

- struct `GenericQueue`
Generic I/O queue structure.

Functions

- void `chlQInit` (`InputQueue *iqp`, `uint8_t *bp`, `size_t size`, `qnotify_t infy`)
Initializes an input queue.
- void `chlQResetl` (`InputQueue *iqp`)
Resets an input queue.
- `msg_t chlQPutl` (`InputQueue *iqp`, `uint8_t b`)
Input queue write.
- `msg_t chlQGetTimeout` (`InputQueue *iqp`, `systime_t time`)
Input queue read with timeout.
- `size_t chlQReadTimeout` (`InputQueue *iqp`, `uint8_t *bp`, `size_t n`, `systime_t time`)
Input queue read with timeout.
- void `choQInit` (`OutputQueue *oqp`, `uint8_t *bp`, `size_t size`, `qnotify_t onfy`)
Initializes an output queue.
- void `choQResetl` (`OutputQueue *oqp`)
Resets an output queue.
- `msg_t choQPutTimeout` (`OutputQueue *oqp`, `uint8_t b`, `systime_t time`)
Output queue write with timeout.
- `msg_t choQGetl` (`OutputQueue *oqp`)
Output queue read.
- `size_t choQWriteTimeout` (`OutputQueue *oqp`, `const uint8_t *bp`, `size_t n`, `systime_t time`)
Output queue write with timeout.

Queue functions returned status value

- `#define Q_OK RDY_OK`
Operation successful.
- `#define Q_TIMEOUT RDY_TIMEOUT`
Timeout condition.
- `#define Q_RESET RDY_RESET`
Queue has been reset.
- `#define Q_EMPTY -3`
Queue empty.
- `#define Q_FULL -4`
Queue full.

Macro Functions

- `#define chQSizeI(qp) ((size_t)((qp)->q_top - (qp)->q_buffer))`
Returns the queue's buffer size.
- `#define chQSpaceI(qp) ((qp)->q_counter)`
Queue space.
- `#define chIQGetFullI(iqp) chQSpaceI(iqp)`
Returns the filled space into an input queue.
- `#define chIQGetEmptyI(iqp) (chQSizeI(iqp) - chQSpaceI(iqp))`
Returns the empty space into an input queue.
- `#define chIQIsEmptyI(iqp) ((bool_t)(chQSpaceI(iqp) <= 0))`
Evaluates to TRUE if the specified input queue is empty.
- `#define chIQIsFullI(iqp)`
Evaluates to TRUE if the specified input queue is full.
- `#define chIQGet(iqp) chIQGetTimeout(iqp, TIME_INFINITE)`
Input queue read.
- `#define chOQGetFullI(oqp) (chQSizeI(oqp) - chQSpaceI(oqp))`
Returns the filled space into an output queue.
- `#define chOQGetEmptyI(iqp) chQSpaceI(oqp)`
Returns the empty space into an output queue.
- `#define chOQIsEmptyI(oqp)`
Evaluates to TRUE if the specified output queue is empty.
- `#define chOQIsFullI(oqp) ((bool_t)(chQSpaceI(oqp) <= 0))`
Evaluates to TRUE if the specified output queue is full.
- `#define chOQPut(oqp, b) chOQPutTimeout(oqp, b, TIME_INFINITE)`
Output queue write.

Defines

- `#define _INPUTQUEUE_DATA(name, buffer, size, inotify)`
Data part of a static input queue initializer.
- `#define INPUTQUEUE_DECL(name, buffer, size, inotify) InputQueue name = _INPUTQUEUE_DATA(name, buffer, size, inotify)`
Static input queue initializer.
- `#define _OUTPUTQUEUE_DATA(name, buffer, size, onotify)`
Data part of a static output queue initializer.
- `#define OUTPUTQUEUE_DECL(name, buffer, size, onotify) OutputQueue name = _OUTPUTQUEUE_DATA(name, buffer, size, onotify)`
Static output queue initializer.

TypeDefs

- `typedef struct GenericQueue GenericQueue`
Type of a generic I/O queue structure.
- `typedef void(* qnotify_t)(GenericQueue *qp)`
Queue notification callback type.
- `typedef GenericQueue InputQueue`
Type of an input queue structure.
- `typedef GenericQueue OutputQueue`
Type of an output queue structure.

7.27.2 Function Documentation

7.27.2.1 void chIQInit (InputQueue * *iqp*, uint8_t * *bp*, size_t *size*, qnotify_t *infy*)

Initializes an input queue.

A [Semaphore](#) is internally initialized and works as a counter of the bytes contained in the queue.

Note

The callback is invoked from within the S-Locked system state, see [System States](#).

Parameters

out	<i>iqp</i>	pointer to an <code>InputQueue</code> structure
in	<i>bp</i>	pointer to a memory area allocated as queue buffer
in	<i>size</i>	size of the queue buffer
in	<i>infy</i>	pointer to a callback function that is invoked when data is read from the queue. The value can be <code>NULL</code> .

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.27.2.2 void chIQResetI (InputQueue * *iqp*)

Resets an input queue.

All the data in the input queue is erased and lost, any waiting thread is resumed with status `Q_RESET`.

Note

A reset operation can be used by a low level driver in order to obtain immediate attention from the high level layers.

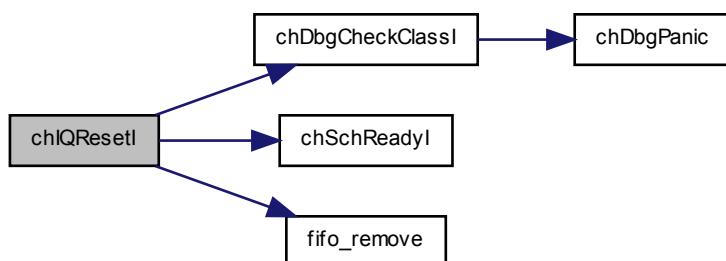
Parameters

in	<i>iqp</i>	pointer to an <code>InputQueue</code> structure
----	------------	---

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.27.2.3 msg_t chIQPutl (InputQueue * *iqp*, uint8_t *b*)

Input queue write.

A byte value is written into the low end of an input queue.

Parameters

in	<i>iqp</i>	pointer to an <code>InputQueue</code> structure
in	<i>b</i>	the byte value to be written in the queue

Returns

The operation status.

Return values

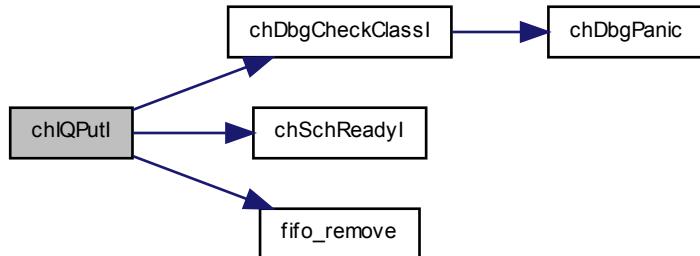
`Q_OK` if the operation has been completed with success.

`Q_FULL` if the queue is full and the operation cannot be completed.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.27.2.4 msg_t chIQGetTimeout (InputQueue * *iqp*, systime_t *time*)

Input queue read with timeout.

This function reads a byte value from an input queue. If the queue is empty then the calling thread is suspended until a byte arrives in the queue or a timeout occurs.

Note

The callback is invoked before reading the character from the buffer or before entering the state `THD__STATE_WTQUEUE`.

Parameters

in	<i>iqp</i>	pointer to an <code>InputQueue</code> structure
----	------------	---

in *time* the number of ticks before the operation timeouts, the following special values are allowed:

- *TIME_IMMEDIATE* immediate timeout.
- *TIME_INFINITE* no timeout.

Returns

A byte value from the queue.

Return values

Q_TIMEOUT if the specified time expired.
Q_RESET if the queue has been reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.2.5 size_t chIQReadTimeout(InputQueue * *iqp*, uint8_t * *bp*, size_t *n*, systime_t *time*)

Input queue read with timeout.

The function reads data from an input queue into a buffer. The operation completes when the specified amount of data has been transferred or after the specified timeout or if the queue has been reset.

Note

The function is not atomic, if you need atomicity it is suggested to use a semaphore or a mutex for mutual exclusion.

The callback is invoked before reading each character from the buffer or before entering the state THD_STATE_WTQUEUE.

Parameters

in	<i>iqp</i>	pointer to an <code>InputQueue</code> structure
out	<i>bp</i>	pointer to the data buffer
in	<i>n</i>	the maximum amount of data to be transferred, the value 0 is reserved
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed:

- *TIME_IMMEDIATE* immediate timeout.
- *TIME_INFINITE* no timeout.

Returns

The number of bytes effectively transferred.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.2.6 void chOQInit(OutputQueue * *oqp*, uint8_t * *bp*, size_t *size*, qnotify_t *onfy*)

Initializes an output queue.

A [Semaphore](#) is internally initialized and works as a counter of the free bytes in the queue.

Note

The callback is invoked from within the S-Locked system state, see [System States](#).

Parameters

out	<i>oqp</i>	pointer to an <code>OutputQueue</code> structure
in	<i>bp</i>	pointer to a memory area allocated as queue buffer
in	<i>size</i>	size of the queue buffer
in	<i>onfy</i>	pointer to a callback function that is invoked when data is written to the queue. The value can be <code>NULL</code> .

Function Class:

Initializer, this function just initializes an object and can be invoked before the kernel is initialized.

7.27.2.7 void chOQResetI (`OutputQueue * oqp`)

Resets an output queue.

All the data in the output queue is erased and lost, any waiting thread is resumed with status `Q_RESET`.

Note

A reset operation can be used by a low level driver in order to obtain immediate attention from the high level layers.

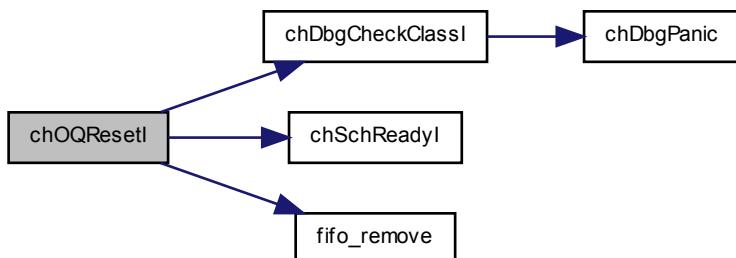
Parameters

in	<i>oqp</i>	pointer to an <code>OutputQueue</code> structure
----	------------	--

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:

**7.27.2.8 msg_t chOQPutTimeout (`OutputQueue * oqp, uint8_t b, systime_t time`)**

Output queue write with timeout.

This function writes a byte value to an output queue. If the queue is full then the calling thread is suspended until there is space in the queue or a timeout occurs.

Note

The callback is invoked after writing the character into the buffer.

Parameters

in	<i>oqp</i>	pointer to an <code>OutputQueue</code> structure
in	<i>b</i>	the byte value to be written in the queue
in	<i>time</i>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none">• <code>TIME_IMMEDIATE</code> immediate timeout.• <code>TIME_INFINITE</code> no timeout.

Returns

The operation status.

Return values

<code>Q_OK</code>	if the operation succeeded.
<code>Q_TIMEOUT</code>	if the specified time expired.
<code>Q_RESET</code>	if the queue has been reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.2.9 `msg_t chOQGetl (OutputQueue * oqp)`

Output queue read.

A byte value is read from the low end of an output queue.

Parameters

in	<i>oqp</i>	pointer to an <code>OutputQueue</code> structure

Returns

The byte value from the queue.

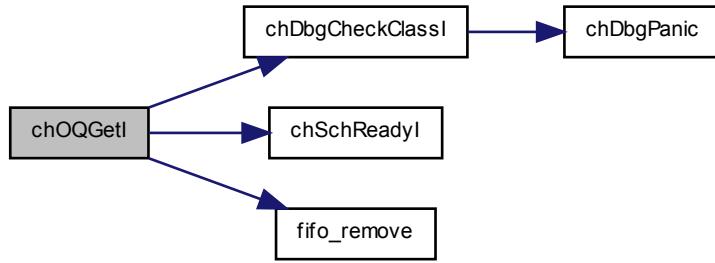
Return values

<code>Q_EMPTY</code>	if the queue is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

Here is the call graph for this function:



7.27.2.10 size_t chOQWriteTimeout (OutputQueue * oqp, const uint8_t * bp, size_t n, systime_t time)

Output queue write with timeout.

The function writes data from a buffer to an output queue. The operation completes when the specified amount of data has been transferred or after the specified timeout or if the queue has been reset.

Note

The function is not atomic, if you need atomicity it is suggested to use a semaphore or a mutex for mutual exclusion.

The callback is invoked after writing each character into the buffer.

Parameters

in	<code>oqp</code>	pointer to an <code>OutputQueue</code> structure
out	<code>bp</code>	pointer to the data buffer
in	<code>n</code>	the maximum amount of data to be transferred, the value 0 is reserved
in	<code>time</code>	the number of ticks before the operation timeouts, the following special values are allowed: <ul style="list-style-type: none"> • <code>TIME_IMMEDIATE</code> immediate timeout. • <code>TIME_INFINITE</code> no timeout.

Returns

The number of bytes effectively transferred.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.3 Define Documentation

7.27.3.1 #define Q_OK RDY_OK

Operation successful.

7.27.3.2 #define Q_TIMEOUT RDY_TIMEOUT

Timeout condition.

7.27.3.3 #define Q_RESET RDY_RESET

Queue has been reset.

7.27.3.4 #define Q_EMPTY -3

Queue empty.

7.27.3.5 #define Q_FULL -4

Queue full..

7.27.3.6 #define chQSize(*qp*) ((size_t)((*qp*)>q_top - (*qp*)>q_buffer))

Returns the queue's buffer size.

Parameters

in *qp* pointer to a [GenericQueue](#) structure.

Returns

The buffer size.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.7 #define chQSpaceI(*qp*) ((*qp*)>q_counter)

Queue space.

Returns the used space if used on an input queue or the empty space if used on an output queue.

Parameters

in *qp* pointer to a [GenericQueue](#) structure.

Returns

The buffer space.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.8 #define chIQGetFullI(*iqp*) chQSpaceI(*iqp*)

Returns the filled space into an input queue.

Parameters

in *iqp* pointer to an `InputQueue` structure

Returns

The number of full bytes in the queue.

Return values

0 if the queue is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.9 `#define chIQGetEmpty(iqp) (chQSize(iqp) - chQSpace(iqp))`

Returns the empty space into an input queue.

Parameters

in *iqp* pointer to an `InputQueue` structure

Returns

The number of empty bytes in the queue.

Return values

0 if the queue is full.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.10 `#define chIQIsEmpty(iqp) ((bool_t)(chQSpace(iqp) <= 0))`

Evaluates to TRUE if the specified input queue is empty.

Parameters

in *iqp* pointer to an `InputQueue` structure.

Returns

The queue status.

Return values

`FALSE` The queue is not empty.
`TRUE` The queue is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.11 #define chIQIsFull(*iqp*)**Value:**

```
((bool_t) (((iwp)->q_wptr == (iwp)->q_rdptr) && \
           ((iwp)->q_counter != 0)))
```

Evaluates to TRUE if the specified input queue is full.

Parameters

in *iwp* pointer to an InputQueue structure.

Returns

The queue status.

Return values

FALSE The queue is not full.

TRUE The queue is full.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.12 #define chIQGet(*iqp*) chIQGetTimeout(iqp, TIME_INFINITE)

Input queue read.

This function reads a byte value from an input queue. If the queue is empty then the calling thread is suspended until a byte arrives in the queue.

Parameters

in *iwp* pointer to an InputQueue structure

Returns

A byte value from the queue.

Return values

Q_RESET if the queue has been reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.3.13 #define _INPUTQUEUE_DATA(*name*, *buffer*, *size*, *inotify*)**Value:**

```
{
    \
    _THREADSQUEUE_DATA(name), \
    0, \
    (uint8_t *) (buffer), \
    (uint8_t *) (buffer) + (size), \
    (uint8_t *) (buffer), \
    (uint8_t *) (buffer), \
    inotify \
}
```

Data part of a static input queue initializer.

This macro should be used when statically initializing an input queue that is part of a bigger structure.

Parameters

in	<i>name</i>	the name of the input queue variable
in	<i>buffer</i>	pointer to the queue buffer area
in	<i>size</i>	size of the queue buffer area
in	<i>inotify</i>	input notification callback pointer

7.27.3.14 #define INPUTQUEUE_DECL(*name*, *buffer*, *size*, *inotify*) InputQueue *name* = _INPUTQUEUE_DATA(*name*, *buffer*, *size*, *inotify*)

Static input queue initializer.

Statically initialized input queues require no explicit initialization using [chIQInit\(\)](#).

Parameters

in	<i>name</i>	the name of the input queue variable
in	<i>buffer</i>	pointer to the queue buffer area
in	<i>size</i>	size of the queue buffer area
in	<i>inotify</i>	input notification callback pointer

7.27.3.15 #define chOQGetFull(*oqp*) (chQSize(*oqp*) - chQSpace(*oqp*))

Returns the filled space into an output queue.

Parameters

in	<i>oqp</i>	pointer to an OutputQueue structure
----	------------	-------------------------------------

Returns

The number of full bytes in the queue.

Return values

0 if the queue is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.16 #define chOQGetEmpty(*iqp*) chQSpace(*oqp*)

Returns the empty space into an output queue.

Parameters

in	<i>iqp</i>	pointer to an OutputQueue structure
----	------------	-------------------------------------

Returns

The number of empty bytes in the queue.

Return values

O if the queue is full.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.17 #define chOQIsEmpty(*oqp*)

Value:

```
((bool_t) ((oqp)->q_wptr == (oqp)->q_rptr) && \
((oqp)->q_counter != 0)))
```

Evaluates to TRUE if the specified output queue is empty.

Parameters

in *oqp* pointer to an `OutputQueue` structure.

Returns

The queue status.

Return values

FALSE The queue is not empty.

TRUE The queue is empty.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.18 #define chOQIsFull(*oqp*) ((*bool_t*)(chQSpace(*oqp*) <= 0))

Evaluates to TRUE if the specified output queue is full.

Parameters

in *oqp* pointer to an `OutputQueue` structure.

Returns

The queue status.

Return values

FALSE The queue is not full.

TRUE The queue is full.

Function Class:

This is an **I-Class** API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

7.27.3.19 #define chOQPut(*oqp*, *b*) chOQPutTimeout(*oqp*, *b*, TIME_INFINITE)

Output queue write.

This function writes a byte value to an output queue. If the queue is full then the calling thread is suspended until there is space in the queue.

Parameters

in	<i>oqp</i>	pointer to an OutputQueue structure
in	<i>b</i>	the byte value to be written in the queue

Returns

The operation status.

Return values

<i>Q_OK</i>	if the operation succeeded.
<i>Q_RESET</i>	if the queue has been reset.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.27.3.20 #define _OUTPUTQUEUE_DATA(*name*, *buffer*, *size*, *onnotify*)

Value:

```
{
    \
    _THREADSQUEUE_DATA(name),
    (size),
    (uint8_t *) (buffer),
    (uint8_t *) (buffer) + (size),
    (uint8_t *) (buffer),
    (uint8_t *) (buffer),
    onnotify
}
```

Data part of a static output queue initializer.

This macro should be used when statically initializing an output queue that is part of a bigger structure.

Parameters

in	<i>name</i>	the name of the output queue variable
in	<i>buffer</i>	pointer to the queue buffer area
in	<i>size</i>	size of the queue buffer area
in	<i>onnotify</i>	output notification callback pointer

7.27.3.21 #define OUTPUTQUEUE_DECL(*name*, *buffer*, *size*, *onnotify*) OutputQueue *name* =
_OUTPUTQUEUE_DATA(name, buffer, size, onnotify)

Static output queue initializer.

Statically initialized output queues require no explicit initialization using [chOQInit\(\)](#).

Parameters

in	<i>name</i>	the name of the output queue variable
in	<i>buffer</i>	pointer to the queue buffer area
in	<i>size</i>	size of the queue buffer area
in	<i>onnotify</i>	output notification callback pointer

7.27.4 Typedef Documentation

7.27.4.1 `typedef struct GenericQueue GenericQueue`

Type of a generic I/O queue structure.

7.27.4.2 `typedef void(* qnotify_t)(GenericQueue *qp)`

Queue notification callback type.

7.27.4.3 `typedef GenericQueue InputQueue`

Type of an input queue structure.

This structure represents a generic asymmetrical input queue. Writing to the queue is non-blocking and can be performed from interrupt handlers or from within a kernel lock zone (see **I-Locked** and **S-Locked** states in [System States](#)). Reading the queue can be a blocking operation and is supposed to be performed by a system thread.

7.27.4.4 `typedef GenericQueue OutputQueue`

Type of an output queue structure.

This structure represents a generic asymmetrical output queue. Reading from the queue is non-blocking and can be performed from interrupt handlers or from within a kernel lock zone (see **I-Locked** and **S-Locked** states in [System States](#)). Writing the queue can be a blocking operation and is supposed to be performed by a system thread.

7.28 Registry

7.28.1 Detailed Description

Threads Registry related APIs and services.

Operation mode

The Threads Registry is a double linked list that holds all the active threads in the system.

Operations defined for the registry:

- **First**, returns the first, in creation order, active thread in the system.
- **Next**, returns the next, in creation order, active thread in the system.

The registry is meant to be mainly a debug feature, for example, using the registry a debugger can enumerate the active threads in any given moment or the shell can print the active threads and their state.

Another possible use is for centralized threads memory management, terminating threads can pulse an event source and an event handler can perform a scansion of the registry in order to recover the memory.

Precondition

In order to use the threads registry the `CH_USE_REGISTRY` option must be enabled in [`chconf.h`](#).

Functions

- `Thread * chRegFirstThread (void)`
Returns the first thread in the system.
- `Thread * chRegNextThread (Thread *tp)`
Returns the thread next to the specified one.

Macro Functions

- `#define chRegSetThreadName(p) (currp->p_name = (p))`
Sets the current thread name.
- `#define chRegGetThreadName(tp) ((tp)->p_name)`
Returns the name of the specified thread.

Defines

- `#define REG_REMOVE(tp)`
Removes a thread from the registry list.
- `#define REG_INSERT(tp)`
Adds a thread to the registry list.

7.28.2 Function Documentation

7.28.2.1 `Thread * chRegFirstThread (void)`

Returns the first thread in the system.

Returns the most ancient thread in the system, usually this is the main thread unless it terminated. A reference is added to the returned thread in order to make sure its status is not lost.

Note

This function cannot return `NULL` because there is always at least one thread in the system.

Returns

A reference to the most ancient thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.28.2.2 `Thread * chRegNextThread (Thread * tp)`

Returns the thread next to the specified one.

The reference counter of the specified thread is decremented and the reference counter of the returned thread is incremented.

Parameters

in `tp` pointer to the thread

Returns

A reference to the next thread.

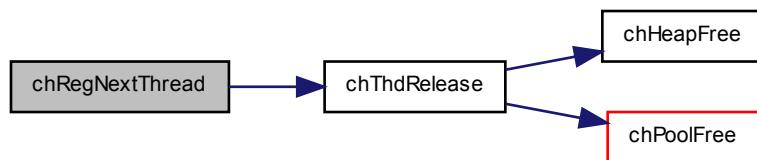
Return values

`NULL` if there is no next thread.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:

**7.28.3 Define Documentation****7.28.3.1 #define chRegSetThreadName(p) (currp->p.name = (p))**

Sets the current thread name.

Precondition

This function only stores the pointer to the name if the option CH_USE_REGISTRY is enabled else no action is performed.

Parameters

in	<i>p</i> thread name as a zero terminated string
----	--

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.28.3.2 #define chRegGetThreadName(tp) ((tp)->p.name)

Returns the name of the specified thread.

Precondition

This function only returns the pointer to the name if the option CH_USE_REGISTRY is enabled else `NULL` is returned.

Parameters

in	<i>tp</i> pointer to the thread
----	---------------------------------

Returns

`Thread` name as a zero terminated string.

Return values

`NULL` if the thread name has not been set.

7.28.3.3 `#define REG_REMOVE(tp)`**Value:**

```
{
  (tp)->p_older->p_newer = (tp)->p_newer;
  (tp)->p_newer->p_older = (tp)->p_older;
}
```

Removes a thread from the registry list.

Note

This macro is not meant for use in application code.

Parameters

in *tp* thread to remove from the registry

7.28.3.4 `#define REG_INSERT(tp)`**Value:**

```
{
  (tp)->p_newer = (Thread *)&rlist;
  (tp)->p_older = rlist.r_older;
  (tp)->p_older->p_newer = rlist.r_older = (tp);
}
```

Adds a thread to the registry list.

Note

This macro is not meant for use in application code.

Parameters

in *tp* thread to add to the registry

7.29 Debug

7.29.1 Detailed Description

Debug APIs and services:

- Runtime system state and call protocol check. The following panic messages can be generated:
 - SV#1, misplaced `chSysDisable()`.
 - SV#2, misplaced `chSysSuspend()`.
 - SV#3, misplaced `chSysEnable()`.
 - SV#4, misplaced `chSysLock()`.
 - SV#5, misplaced `chSysUnlock()`.
 - SV#6, misplaced `chSysLockFromIsr()`.

- SV#7, misplaced `chSysUnlockFromIsr()`.
- SV#8, misplaced `CH_IRQ_PROLOGUE()`.
- SV#9, misplaced `CH_IRQ_EPILOGUE()`.
- SV#10, misplaced I-class function.
- SV#11, misplaced S-class function.
- Trace buffer.
- Parameters check.
- Kernel assertions.
- Kernel panics.

Note

Stack checks are not implemented in this module but in the port layer in an architecture-dependent way.

Data Structures

- struct `ch_swc_event_t`
Trace buffer record.
- struct `ch_trace_buffer_t`
Trace buffer header.

Functions

- void `_trace_init` (void)
Trace circular buffer subsystem initialization.
- void `dbg_trace` (`Thread` *otp)
Inserts in the circular debug trace buffer a context switch record.
- void `dbg_check_disable` (void)
Guard code for `chSysDisable()`.
- void `dbg_check_suspend` (void)
Guard code for `chSysSuspend()`.
- void `dbg_check_enable` (void)
Guard code for `chSysEnable()`.
- void `dbg_check_lock` (void)
Guard code for `chSysLock()`.
- void `dbg_check_unlock` (void)
Guard code for `chSysUnlock()`.
- void `dbg_check_lock_from_isr` (void)
Guard code for `chSysLockFromIsr()`.
- void `dbg_check_unlock_from_isr` (void)
Guard code for `chSysUnlockFromIsr()`.
- void `dbg_check_enter_isr` (void)
Guard code for `CH_IRQ_PROLOGUE()`.
- void `dbg_check_leave_isr` (void)
Guard code for `CH_IRQ_EPILOGUE()`.
- void `chDbgCheckClassI` (void)
I-class functions context check.
- void `chDbgCheckClassS` (void)
S-class functions context check.
- void `chDbgPanic` (`char` *msg)
Prints a panic message on the console and then halts the system.

Variables

- `char * dbg_panic_msg`
Pointer to the panic message.
- `cnt_t dbg_isr_cnt`
ISR nesting level.
- `cnt_t dbg_lock_cnt`
Lock nesting level.
- `ch_trace_buffer_t dbg_trace_buffer`
Public trace buffer.
- `char * dbg_panic_msg`
Pointer to the panic message.

Debug related settings

- `#define CH_TRACE_BUFFER_SIZE 64`
Trace buffer entries.
- `#define CH_STACK_FILL_VALUE 0x55`
Fill value for thread stack area in debug mode.
- `#define CH_THREAD_FILL_VALUE 0xFF`
Fill value for thread area in debug mode.

Macro Functions

- `#define chDbgCheck(c, func)`
Function parameter check.
- `#define chDbgAssert(c, m, r)`
Condition assertion.

7.29.2 Function Documentation

7.29.2.1 void _trace_init(void)

Trace circular buffer subsystem initialization.

Note

Internal use only.

7.29.2.2 void dbg_trace(Thread * otp)

Inserts in the circular debug trace buffer a context switch record.

Parameters

in `otp` the thread being switched out

Function Class:

Not an API, this function is for internal use only.

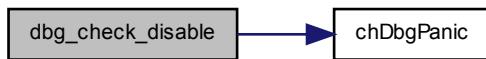
7.29.2.3 void dbg_check_disable (void)

Guard code for [chSysDisable\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.29.2.4 void dbg_check_suspend (void)

Guard code for [chSysSuspend\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



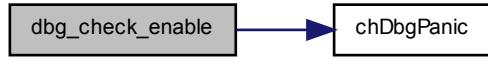
7.29.2.5 void dbg_check_enable (void)

Guard code for [chSysEnable\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



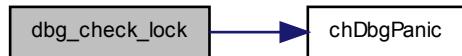
7.29.2.6 void dbg_check_lock(void)

Guard code for [chSysLock\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.29.2.7 void dbg_check_unlock(void)

Guard code for [chSysUnlock\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.29.2.8 void dbg_check_lock_from_isr(void)

Guard code for [chSysLockFromIsr\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



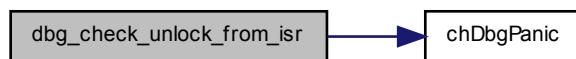
7.29.2.9 void dbg_check_unlock_from_isr(void)

Guard code for [chSysUnlockFromIsr\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



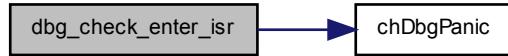
7.29.2.10 void dbg_check_enter_isr(void)

Guard code for [CH_IRQ_PROLOGUE\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.29.2.11 void dbg_check_leave_isr(void)

Guard code for [CH_IRQ_EPILOGUE\(\)](#).

Function Class:

Not an API, this function is for internal use only.

Here is the call graph for this function:



7.29.2.12 void chDbgCheckClassI(void)

I-class functions context check.

Verifies that the system is in an appropriate state for invoking an I-class API function. A panic is generated if the state is not compatible.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.29.2.13 void chDbgCheckClassS (void)

S-class functions context check.

Verifies that the system is in an appropriate state for invoking an S-class API function. A panic is generated if the state is not compatible.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Here is the call graph for this function:



7.29.2.14 void chDbgPanic (char * msg)

Prints a panic message on the console and then halts the system.

Parameters

in *msg* the pointer to the panic message string

7.29.3 Variable Documentation

7.29.3.1 char* dbg_panic_msg

Pointer to the panic message.

This pointer is meant to be accessed through the debugger, it is written once and then the system is halted.

7.29.3.2 cnt_t dbg_isr_cnt

ISR nesting level.

7.29.3.3 cnt_t dbg_lock_cnt

Lock nesting level.

7.29.3.4 ch_trace_buffer_t dbg_trace_buffer

Public trace buffer.

7.29.3.5 char* dbg_panic_msg

Pointer to the panic message.

This pointer is meant to be accessed through the debugger, it is written once and then the system is halted.

7.29.4 Define Documentation

7.29.4.1 #define CH_TRACE_BUFFER_SIZE 64

Trace buffer entries.

7.29.4.2 #define CH_STACK_FILL_VALUE 0x55

Fill value for thread stack area in debug mode.

7.29.4.3 #define CH_THREAD_FILL_VALUE 0xFF

Fill value for thread area in debug mode.

Note

The chosen default value is 0xFF in order to make evident which thread fields were not initialized when inspecting the memory with a debugger. A uninitialized field is not an error in itself but it better to know it.

7.29.4.4 #define chDbgCheck(c, func)

Value:

```
{
    if (!c) \
        chDbgPanic(__QUOTE_THIS(func) "()"); \
}
```

Function parameter check.

If the condition check fails then the kernel panics and halts.

Note

The condition is tested only if the `CH_DBG_ENABLE_CHECKS` switch is specified in `chconf.h` else the macro does nothing.

Parameters

in	<code>c</code>	the condition to be verified to be true
in	<code>func</code>	the undecorated function name

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.29.4.5 #define chDbgAssert(c, m, r)

Value:

```
{
    if (!c) \
        chDbgPanic(m); \
}
```

Condition assertion.

If the condition check fails then the kernel panics with the specified message and halts.

Note

The condition is tested only if the CH_DBG_ENABLE_ASSERTS switch is specified in `chconf.h` else the macro does nothing.

The convention for the message is the following:

`<function_name>(), #<assert_number>`

The remark string is not currently used except for putting a comment in the code about the assertion.

Parameters

in	<i>c</i>	the condition to be verified to be true
in	<i>m</i>	the text message
in	<i>r</i>	a remark string

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

7.30 Internals

7.30.1 Detailed Description

All the functions present in this module, while public, are not OS APIs and should not be directly used in the user applications code.

Data Structures

- struct `ThreadsQueue`
Generic threads bidirectional linked list header and element.
- struct `ThreadsList`
Generic threads single link list, it works like a stack.

Functions

- void `prio_insert (Thread *tp, ThreadsQueue *tqp)`
Inserts a thread into a priority ordered queue.
- void `queue_insert (Thread *tp, ThreadsQueue *tqp)`
Inserts a Thread into a queue.
- `Thread * fifo_remove (ThreadsQueue *tqp)`
Removes the first-out Thread from a queue and returns it.
- `Thread * lifo_remove (ThreadsQueue *tqp)`
Removes the last-out Thread from a queue and returns it.
- `Thread * dequeue (Thread *tp)`
Removes a Thread from a queue and returns it.
- void `list_insert (Thread *tp, ThreadsList *tlp)`
Pushes a Thread on top of a stack list.
- `Thread * list_remove (ThreadsList *tlp)`
Pops a Thread from the top of a stack list and returns it.

Defines

- `#define queue_init(tqp) ((tqp)->p_next = (tqp)->p_prev = (Thread *)(tqp));`
Threads queue initialization.
- `#define list_init(tlp) ((tlp)->p_next = (Thread *)(tlp))`
Threads list initialization.
- `#define isempty(p) ((p)->p_next == (Thread *)(p))`
Evaluates to TRUE if the specified threads queue or list is empty.
- `#define notempty(p) ((p)->p_next != (Thread *)(p))`
Evaluates to TRUE if the specified threads queue or list is not empty.
- `#define _THREADSQUEUE_DATA(name) {(<Thread *>)&name, (<Thread *>)&name}`
Data part of a static threads queue initializer.
- `#define THREADSQUEUE_DECL(name) ThreadsQueue name = _THREADSQUEUE_DATA(name)`
Static threads queue initializer.

7.30.2 Function Documentation

7.30.2.1 void prio_insert (Thread * tp, ThreadsQueue * tqp)

Inserts a thread into a priority ordered queue.

Note

The insertion is done by scanning the list from the highest priority toward the lowest.

Parameters

in	<code>tp</code>	the pointer to the thread to be inserted in the list
in	<code>tqp</code>	the pointer to the threads list header

Function Class:

Not an API, this function is for internal use only.

7.30.2.2 void queue_insert (Thread * tp, ThreadsQueue * tqp)

Inserts a `Thread` into a queue.

Parameters

in	<code>tp</code>	the pointer to the thread to be inserted in the list
in	<code>tqp</code>	the pointer to the threads list header

Function Class:

Not an API, this function is for internal use only.

7.30.2.3 Thread * fifo_remove (ThreadsQueue * tqp)

Removes the first-out `Thread` from a queue and returns it.

Note

If the queue is priority ordered then this function returns the thread with the highest priority.

Parameters

in *tqp* the pointer to the threads list header

Returns

The removed thread pointer.

Function Class:

Not an API, this function is for internal use only.

7.30.2.4 Thread * lifo_remove (ThreadsQueue * *tqp*)

Removes the last-out [Thread](#) from a queue and returns it.

Note

If the queue is priority ordered then this function returns the thread with the lowest priority.

Parameters

in *tqp* the pointer to the threads list header

Returns

The removed thread pointer.

Function Class:

Not an API, this function is for internal use only.

7.30.2.5 Thread * dequeue (Thread * *tp*)

Removes a [Thread](#) from a queue and returns it.

The thread is removed from the queue regardless of its relative position and regardless the used insertion method.

Parameters

in *tp* the pointer to the thread to be removed from the queue

Returns

The removed thread pointer.

Function Class:

Not an API, this function is for internal use only.

7.30.2.6 void list_insert (Thread * *tp*, ThreadsList * *tlp*)

Pushes a [Thread](#) on top of a stack list.

Parameters

in *tp* the pointer to the thread to be inserted in the list
in *tlp* the pointer to the threads list header

Function Class:

Not an API, this function is for internal use only.

7.30.2.7 Thread * list_remove (ThreadsList * tlp)

Pops a [Thread](#) from the top of a stack list and returns it.

Precondition

The list must be non-empty before calling this function.

Parameters

in *tlp* the pointer to the threads list header

Returns

The removed thread pointer.

Function Class:

Not an API, this function is for internal use only.

7.30.3 Define Documentation**7.30.3.1 #define queue_init(*tqp*) ((*tqp*)>p_next = (*tqp*)>p_prev = (Thread *)(*tqp*));**

Threads queue initialization.

Function Class:

Not an API, this function is for internal use only.

7.30.3.2 #define list_init(*tlp*) ((*tlp*)>p_next = (Thread *)(*tlp*))

Threads list initialization.

Function Class:

Not an API, this function is for internal use only.

7.30.3.3 #define isempty(*p*) ((*p*)>p_next == (Thread *)(*p*))

Evaluates to TRUE if the specified threads queue or list is empty.

Function Class:

Not an API, this function is for internal use only.

7.30.3.4 #define notempty(*p*) ((*p*)>p_next != (Thread *)(*p*))

Evaluates to TRUE if the specified threads queue or list is not empty.

Function Class:

Not an API, this function is for internal use only.

7.30.3.5 `#define _THREADSQUEUE_DATA(name) {(Thread *)&name, (Thread *)&name}`

Data part of a static threads queue initializer.

This macro should be used when statically initializing a threads queue that is part of a bigger structure.

Parameters

in *name* the name of the threads queue variable

7.30.3.6 `#define THREADSQUEUE_DECL(name) ThreadsQueue name = _THREADSQUEUE_DATA(name)`

Static threads queue initializer.

Statically initialized threads queues require no explicit initialization using `queue_init()`.

Parameters

in *name* the name of the threads queue variable

7.31 ARM7/9

7.31.1 Detailed Description

ARM7/9 port for the GCC compiler.

7.31.2 Introduction

The ARM7/9-GCC port supports the ARM7/9 core in the following three modes:

- **Pure ARM** mode, this is the preferred mode for code speed, this mode increases the memory footprint however. This mode is enabled when all the modules are compiled in ARM mode, see the Makefiles.
- **Pure THUMB** mode, this is the preferred mode for code size. In this mode the execution speed is slower than the ARM mode. This mode is enabled when all the modules are compiled in THUMB mode, see the Makefiles.
- **Interworking** mode, when in the system there are ARM modules mixed with THUMB modules then the interworking compiler option is enabled. This is usually the slowest mode and the code size is not as good as in pure THUMB mode.

7.31.3 Mapping of the System States in the ARM7/9 port

The ChibiOS/RT logical system states are mapped as follow in the ARM7/9 port:

- **Init.** This state is represented by the startup code and the initialization code before `chSysInit()` is executed. It has not a special hardware state associated, usually the CPU goes through several hardware states during the startup phase.
- **Normal.** This is the state the system has after executing `chSysInit()`. In this state the CPU has both the interrupt sources (IRQ and FIQ) enabled and is running in ARM System Mode.
- **Suspended.** In this state the IRQ sources are disabled but the FIQ sources are served, the core is running in ARM System Mode.
- **Disabled.** Both the IRQ and FIQ sources are disabled, the core is running in ARM System Mode.

- **Sleep.** ARM7/9 cores does not have an explicit built-in low power mode but there are clock stop modes implemented in custom ways by the various silicon vendors. This state is implemented in each microcontroller support code in a different way, the core is running (or freezed...) in ARM System Mode.
- **S-Locked.** IRQ sources disabled, core running in ARM System Mode.
- **I-Locked.** IRQ sources disabled, core running in ARM IRQ Mode. Note that this state is not different from the SRI state in this port, the `chSysLockI()` and `chSysUnlockI()` APIs do nothing (still use them in order to formally change state because this may change).
- **Serving Regular Interrupt.** IRQ sources disabled, core running in ARM IRQ Mode. See also the I-Locked state.
- **Serving Fast Interrupt.** IRQ and FIQ sources disabled, core running in ARM FIQ Mode.
- **Serving Non-Maskable Interrupt.** There are no asynchronous NMI sources in ARM7/9 architecture but synchronous SVC, ABT and UND exception handlers can be seen as belonging to this category.
- **Halted.** Implemented as an infinite loop after disabling both IRQ and FIQ sources. The ARM state is whatever the processor was running when `chSysHalt()` was invoked.

7.31.4 The ARM7/9 port notes

The ARM7/9 port is organized as follow:

- The `main()` function is invoked in system mode.
 - Each thread has a private user/system stack, the system has a single interrupt stack where all the interrupts are processed.
 - The threads are started in system mode.
 - The threads code can run in system mode or user mode, however the code running in user mode cannot invoke the ChibiOS/RT APIs directly because privileged instructions are used inside.
- The kernel APIs can be eventually invoked by using a SWI entry point that handles the switch in system mode and the return in user mode.
- Other modes are not preempt-able because the system code assumes the threads running in system mode. When running in supervisor or other modes make sure that the interrupts are globally disabled.
 - Interrupts nesting is not supported in the ARM7/9 port because their implementation, even if possible, is not really efficient in this architecture.
 - FIQ sources can preempt the kernel (by design) so it is not possible to invoke the kernel APIs from inside a FIQ handler. FIQ handlers are not affected by the kernel activity so there is not added jitter.

7.31.5 ARM7/9 Interrupt Handlers

In the current implementation the ARM7/9 Interrupt handlers do not save function-saved registers so you need to make sure your code saves them or does not use them (this happens because in the ARM7/9 port all the OS interrupt handler functions are declared naked).

Function-trashed registers (R0-R3, R12, LR, SR) are saved/restored by the system macros `CH_IRQ_PROLOGUE()` and `CH_IRQ_EPILOGUE()`.

The easiest way to ensure this is to just invoke a normal function from within the interrupt handler, the function code will save all the required registers.

Example:

```

CH_IRQ_HANDLER(irq_handler) {
    CH_IRQ_PROLOGUE();

    serve_interrupt();

    VICVectAddr = 0; // This is LPC214x-specific.
    CH_IRQ_EPILOGUE();
}

```

This is not a bug but an implementation choice, this solution allows to have interrupt handlers compiled in thumb mode without have to use an interworking mode (the mode switch is hidden in the macros), this greatly improves code efficiency and size. You can look at the serial driver for real examples of interrupt handlers.

It is important that the `serve_interrupt()` interrupt function is not inlined by the compiler into the ISR or the code could still modify the unsaved registers, this can be accomplished using GCC by adding the attribute "noinline" to the function:

```

#if defined(__GNUC__)
__attribute__((noinline))
#endif
static void serve_interrupt(void) {
}

```

Note that several commercial compilers support a GNU-like functions attribute mechanism.

Alternative ways are to use an appropriate pragma directive or disable inlining optimizations in the modules containing the interrupt handlers.

Modules

- [Configuration Options](#)
- [Core Port Implementation](#)
- [Startup Support](#)
- [Specific Implementations](#)

7.32 Configuration Options

ARM7/9 specific configuration options. The ARM7/9 port allows some architecture-specific configurations settings that can be overridden by redefining them in `chconf.h`. Usually there is no need to change the default values.

- `INT_REQUIRED_STACK`, this value represent the amount of stack space used by an interrupt handler between the `extctx` and `intctx` structures.

In practice this value is the stack space used by the `chSchDoReschedule()` stack frame.

This value can be affected by a variety of external things like compiler version, compiler options, kernel settings (speed/size) and so on.

The default for this value is `0x10` which should be a safe value, you can trim this down by defining the macro externally. This would save some valuable RAM space for each thread present in the system.

The default value is set into `./os/ports/GCC/ARM/chcore.h`.

- `IDLE_THREAD_STACK_SIZE`, stack area size to be assigned to the IDLE thread. Usually there is no need to change this value unless inserting code in the IDLE thread using the `IDLE_LOOP_HOOK` hook macro.
- `ARM_ENABLE_WFI_IDLE`, if set to TRUE enables the use of the an implementation-specific clock stop mode from within the idle loop. This option is defaulted to FALSE because it can create problems with some debuggers. Setting this option to TRUE reduces the system power requirements.

7.33 Core Port Implementation

7.33.1 Detailed Description

ARM7/9 specific port code, structures and macros.

Data Structures

- struct `extctx`
Interrupt saved context.
- struct `intctx`
System saved context.
- struct `context`
Platform dependent part of the `Thread` structure.

Functions

- void `port_halt` (void)

Defines

- #define `ARM_CORE_ARM7TDMI` 7
- #define `ARM_CORE_ARM9` 9
- #define `ARM_ENABLE_WFI_IDLE` FALSE
If enabled allows the idle thread to enter a low power mode.
- #define `CH_ARCHITECTURE_ARM`
Macro defining a generic ARM architecture.
- #define `CH_ARCHITECTURE_ARMx`
Macro defining the specific ARM architecture.
- #define `CH_ARCHITECTURE_NAME` "ARMx"
Name of the implemented architecture.
- #define `CH_CORE_VARIANT_NAME` "ARMxy"
Name of the architecture variant (optional).
- #define `CH_PORT_INFO` "ARM|THUMB|Interworking"
Port-specific information string.
- #define `CH_PORT_INFO` "Pure ARM mode"
Port-specific information string.
- #define `CH_COMPILER_NAME` "GCC" __VERSION__
Name of the compiler supported by this port.
- #define `SETUP_CONTEXT`(workspace, wsize, pf, arg)
Platform dependent part of the `chThdCreateI()` API.
- #define `PORT_IDLE_THREAD_STACK_SIZE` 4
Stack size for the system idle thread.
- #define `PORT_INT_REQUIRED_STACK` 0x10
Per-thread stack overhead for interrupts servicing.
- #define `STACK_ALIGN`(n) (((n) - 1) | (sizeof(stkalign_t) - 1)) + 1
Enforces a correct alignment for a stack area size value.
- #define `THD_WA_SIZE`(n)
Computes the thread working area global size.
- #define `WORKING_AREA`(s, n) `stkalign_t` s[`THD_WA_SIZE`(n) / sizeof(`stkalign_t`)]

- `#define PORT_IRQ_PROLOGUE()`
IRQ prologue code.
- `#define PORT_IRQ_EPILOGUE()`
IRQ epilogue code.
- `#define PORT_IRQ_HANDLER(id) __attribute__((naked)) void id(void)`
IRQ handler function declaration.
- `#define PORT_FAST_IRQ_HANDLER(id) __attribute__((interrupt("FIQ"))) void id(void)`
Fast IRQ handler function declaration.
- `#define port_init()`
Port-related initialization code.
- `#define port_lock() asm volatile ("msr CPSR_c, #0x9F" :: : "memory")`
Kernel-lock action.
- `#define port_unlock() asm volatile ("msr CPSR_c, #0x1F" :: : "memory")`
Kernel-unlock action.
- `#define port_lock_from_isr()`
Kernel-lock action from an interrupt handler.
- `#define port_unlock_from_isr()`
Kernel-unlock action from an interrupt handler.
- `#define port_disable()`
Disables all the interrupt sources.
- `#define port_suspend() asm volatile ("msr CPSR_c, #0x9F" :: : "memory")`
Disables the interrupt sources below kernel-level priority.
- `#define port_enable() asm volatile ("msr CPSR_c, #0x1F" :: : "memory")`
Enables all the interrupt sources.
- `#define port_switch(ntp, otp)`
Performs a context switch between two threads.
- `#define INLINE inline`
Inline function modifier.
- `#define ROMCONST const`
ROM constant modifier.
- `#define PACK_STRUCT_STRUCT __attribute__((packed))`
Packed structure modifier (within).
- `#define PACK_STRUCT_BEGIN`
Packed structure modifier (before).
- `#define PACK_STRUCT_END`
Packed structure modifier (after).

Typedefs

- `typedef uint32_t stkalign_t`
32 bits stack and memory alignment enforcement.
- `typedef void * regarm_t`
Generic ARM register.
- `typedef int32_t bool_t`
- `typedef uint8_t tmode_t`
- `typedef uint8_t tstate_t`
- `typedef uint8_t trefs_t`
- `typedef uint32_t tprio_t`
- `typedef int32_t msg_t`
- `typedef int32_t eventid_t`
- `typedef uint32_t eventmask_t`
- `typedef uint32_t systime_t`
- `typedef int32_t cnt_t`

7.33.2 Function Documentation

7.33.2.1 `void port_halt(void)`

Halts the system.

7.33.3 Define Documentation

7.33.3.1 `#define ARM_CORE_ARM7TDMI 7`

ARM7TDMI core identifier.

7.33.3.2 `#define ARM_CORE_ARM9 9`

ARM9 core identifier.

7.33.3.3 `#define ARM_ENABLE_WFI_IDLE FALSE`

If enabled allows the idle thread to enter a low power mode.

7.33.3.4 `#define CH_ARCHITECTURE_ARM`

Macro defining a generic ARM architecture.

7.33.3.5 `#define CH_ARCHITECTURE_ARMx`

Macro defining the specific ARM architecture.

Note

This macro is for documentation only, the real name changes depending on the selected architecture, the possible names are:

- `CH_ARCHITECTURE_ARM7TDMI`.
- `CH_ARCHITECTURE_ARM9`.

7.33.3.6 `#define CH_ARCHITECTURE_NAME "ARMx"`

Name of the implemented architecture.

Note

The value is for documentation only, the real value changes depending on the selected architecture, the possible values are:

- "ARM7".
- "ARM9".

7.33.3.7 `#define CH_CORE_VARIANT_NAME "ARMxy"`

Name of the architecture variant (optional).

Note

The value is for documentation only, the real value changes depending on the selected architecture, the possible values are:

- "ARM7TDMI"
- "ARM9"

7.33.3.8 #define CH_PORT_INFO "ARM|THUMB|Interworking"

Port-specific information string.

Note

The value is for documentation only, the real value changes depending on the selected options, the possible values are:

- "Pure ARM"
- "Pure THUMB"
- "Interworking"

7.33.3.9 #define CH_PORT_INFO "Pure ARM mode"

Port-specific information string.

Note

The value is for documentation only, the real value changes depending on the selected options, the possible values are:

- "Pure ARM"
- "Pure THUMB"
- "Interworking"

7.33.3.10 #define CH_COMPILER_NAME "GCC" __VERSION__

Name of the compiler supported by this port.

7.33.3.11 #define SETUP_CONTEXT(workspace, wsize, pf, arg)**Value:**

```
{
    tp->p_ctx.r13 = (struct intctx *)((uint8_t *)workspace +
                                         wsize -
                                         sizeof(struct intctx));
    tp->p_ctx.r13->r4 = pf;
    tp->p_ctx.r13->r5 = arg;
    tp->p_ctx.r13->lr = _port_thread_start;
}
```

Platform dependent part of the [chThdCreateI\(\)](#) API.

This code usually setup the context switching frame represented by an `intctx` structure.

7.33.3.12 #define PORT_IDLE_THREAD_STACK_SIZE 4

Stack size for the system idle thread.

This size depends on the idle thread implementation, usually the idle thread should take no more space than those reserved by `PORT_INT_REQUIRED_STACK`.

Note

In this port it is set to 4 because the idle thread does have a stack frame when compiling without optimizations.

7.33.3.13 #define PORT_INT_REQUIRED_STACK 0x10

Per-thread stack overhead for interrupts servicing.

This constant is used in the calculation of the correct working area size. This value can be zero on those architecture where there is a separate interrupt stack and the stack space between `intctx` and `extctx` is known to be zero.

Note

In this port 0x10 is a safe value, it can be reduced after careful analysis of the generated code.

7.33.3.14 #define STACK_ALIGN(n) (((n) - 1) | (sizeof(stkalign_t) - 1)) + 1

Enforces a correct alignment for a stack area size value.

7.33.3.15 #define THD_WA_SIZE(n)

Value:

```
STACK_ALIGN(sizeof(Thread) +
           \
           sizeof(struct intctx) +
           sizeof(struct extctx) +
           (n) + (PORT_INT_REQUIRED_STACK))
```

Computes the thread working area global size.

7.33.3.16 #define WORKING_AREA(s, n) stkalign_t s[THD_WA_SIZE(n) / sizeof(stkalign_t)]

Static working area allocation.

This macro is used to allocate a static thread working area aligned as both position and size.

7.33.3.17 #define PORT_IRQ_PROLOGUE()

Value:

```
{
    asm volatile ("stmfd    sp!, {r0-r3, r12, lr}" : : : "memory");
}
```

IRQ prologue code.

This macro must be inserted at the start of all IRQ handlers enabled to invoke system APIs.

Note

This macro has a different implementation depending if compiled in ARM or THUMB mode.

The THUMB implementation starts with ARM code because interrupt vectors are always invoked in ARM mode regardless the bit 0 value. The switch in THUMB mode is done in the function prologue so it is transparent to the user code.

7.33.3.18 #define PORT_IRQ_EPILOGUE()

Note:

```
{           _port_irq_common" : : : "memory");\n}
```

IRQ epilogue code.

This macro must be inserted at the end of all IRQ handlers enabled to invoke system APIs.

Note

This macro has a different implementation depending if compiled in ARM or THUMB mode.

7.33.3.19 #define PORT_IRQ_HANDLER(*id*) __attribute__((naked)) void *id*(void)

IRQ handler function declaration.

Note

id can be a function name or a vector number depending on the port implementation.

7.33.3.20 #define PORT_FAST_IRQ_HANDLER(*id*) __attribute__((interrupt("FIQ"))) void *id*(void)

Fast IRQ handler function declaration.

Note

id can be a function name or a vector number depending on the port implementation.

7.33.3.21 #define port_init()

Port-related initialization code.

Note

This function is empty in this port.

7.33.3.22 #define port_lock() asm volatile ("msr CPSR_c, #0x9F" : : : "memory")

Kernel-lock action.

Usually this function just disables interrupts but may perform more actions.

Note

In this port it disables the IRQ sources and keeps FIQ sources enabled.

7.33.3.23 #define port_unlock() asm volatile ("msr CPSR_c, #0x1F" : : : "memory")

Kernel-unlock action.

Usually this function just enables interrupts but may perform more actions.

Note

In this port it enables both the IRQ and FIQ sources.

7.33.3.24 #define port_lock_from_isr()

Kernel-lock action from an interrupt handler.

This function is invoked before invoking I-class APIs from interrupt handlers. The implementation is architecture dependent, in its simplest form it is void.

Note

Empty in this port.

7.33.3.25 #define port_unlock_from_isr()

Kernel-unlock action from an interrupt handler.

This function is invoked after invoking I-class APIs from interrupt handlers. The implementation is architecture dependent, in its simplest form it is void.

Note

Empty in this port.

7.33.3.26 #define port_disable()

Value:

```
{
    asm volatile ("mrs      r3, CPSR          \n\t"
                 "orr      r3, #0x80        \n\t"
                 "msr      CPSR_c, r3       \n\t"
                 "orr      r3, #0x40        \n\t"
                 "msr      CPSR_c, r3" : : : "r3", "memory"); \n
}
```

Disables all the interrupt sources.

Note

Of course non maskable interrupt sources are not included.

In this port it disables both the IRQ and FIQ sources.

Implements a workaround for spurious interrupts taken from the NXP LPC214x datasheet.

7.33.3.27 #define port_suspend() asm volatile ("msr CPSR_c, #0x9F" :: : "memory")

Disables the interrupt sources below kernel-level priority.

Note

Interrupt sources above kernel level remains enabled.

In this port it disables the IRQ sources and enables the FIQ sources.

7.33.3.28 #define port_enable() asm volatile ("msr CPSR_c, #0x1F" :: : "memory")

Enables all the interrupt sources.

Note

In this port it enables both the IRQ and FIQ sources.

7.33.3.29 #define port_switch(*ntp*, *otp*)

Value:

```
{
    register struct intctx *r13 asm ("r13");
    if ((stkalign_t *) (r13 - 1) < otp->p_stklimit)
        chDbgPanic("stack overflow");
    _port_switch_arm(ntp, otp);
}
```

Performs a context switch between two threads.

This is the most critical code in any port, this function is responsible for the context switch between 2 threads.

Note

The implementation of this code affects **directly** the context switch performance so optimize here as much as you can.

Implemented as inlined code for performance reasons.

Parameters

in	<i>ntp</i>	the thread to be switched in
in	<i>otp</i>	the thread to be switched out

7.33.3.30 #define INLINE inline

Inline function modifier.

7.33.3.31 #define ROMCONST const

ROM constant modifier.

Note

It is set to use the "const" keyword in this port.

7.33.3.32 #define PACK_STRUCT_STRUCT __attribute__((packed))

Packed structure modifier (within).

Note

It uses the "packed" GCC attribute.

7.33.3.33 #define PACK_STRUCT_BEGIN

Packed structure modifier (before).

Note

Empty in this port.

7.33.3.34 #define PACK_STRUCT_END

Packed structure modifier (after).

Note

Empty in this port.

7.33.4 Typedef Documentation

7.33.4.1 `typedef uint32_t stkalign_t`

32 bits stack and memory alignment enforcement.

7.33.4.2 `typedef void* regarm_t`

Generic ARM register.

7.33.4.3 `typedef int32_t bool_t`

Fast boolean type.

7.33.4.4 `typedef uint8_t tmode_t`

[Thread](#) flags.

7.33.4.5 `typedef uint8_t tstate_t`

[Thread](#) state.

7.33.4.6 `typedef uint8_t trefs_t`

[Thread](#) references counter.

7.33.4.7 `typedef uint32_t tprio_t`

[Thread](#) priority.

7.33.4.8 `typedef int32_t msg_t`

Inter-thread message.

7.33.4.9 `typedef int32_t eventid_t`

Event Id.

7.33.4.10 `typedef uint32_t eventmask_t`

Events mask.

7.33.4.11 `typedef uint32_t systime_t`

System time.

7.33.4.12 `typedef int32_t cnt_t`

Resources counter.

7.34 Startup Support

ARM7/9 startup code support. ChibiOS/RT provides its own generic startup file for the ARM7/9 port. Of course it is not mandatory to use it but care should be taken about the startup phase details.

7.34.1 Startup Process

The startup process, as implemented, is the following:

1. The stacks are initialized by assigning them the sizes defined in the linker script (usually named `ch.ld`). Stack areas are allocated from the highest RAM location downward.
2. The ARM state is switched to System with both IRQ and FIQ sources disabled.
3. An early initialization routine `hwinit0` is invoked, if the symbol is not defined then an empty default routine is executed (weak symbol).
4. DATA and BSS segments are initialized.
5. A late initialization routine `hwinit1` is invoked, if the symbol not defined then an empty default routine is executed (weak symbol).

This late initialization function is also the proper place for a *bootloader*, if your application requires one.

6. The `main()` function is invoked with the parameters `argc` and `argv` set to zero.
7. Should the `main()` function return a branch is performed to the weak symbol `_main_exit_handler`. The default code is an endless empty loop.

7.34.2 Expected linker symbols

The startup code starts at the symbol `ResetHandler` and expects the following symbols to be defined in the linker script:

- `__ram_end__` RAM end location +1.
- `__und_stack_size__` Undefined Instruction stack size.
- `__abt_stack_size__` Memory Abort stack size.
- `__fiq_stack_size__` FIQ service stack size.
- `__irq_stack_size__` IRQ service stack size.
- `__svc_stack_size__` SVC service stack size.
- `__sys_stack_size__` System/User stack size. This is the stack area used by the `main()` function.
- `_textdata` address of the data segment source read only data.
- `_data` data segment start location.

- `_edata` data segment end location +1.
- `_bss_start` BSS start location.
- `_bss_end` BSS end location +1.

7.35 Specific Implementations

7.35.1 Detailed Description

Platform-specific port code.

Modules

- [AT91SAM7 Specific Parameters](#)
- [AT91SAM7 Interrupt Vectors](#)
- [LPC214x Specific Parameters](#)
- [LPC214x Interrupt Vectors](#)

7.36 AT91SAM7 Specific Parameters

7.36.1 Detailed Description

This file contains the ARM specific parameters for the AT91SAM7 platform.

Defines

- `#define ARM_CORE ARM_CORE_ARM7TDMI`
ARM core model.
- `#define port_wait_for_interrupt()`
AT91SAM7-specific wait for interrupt.

7.36.2 Define Documentation

7.36.2.1 `#define ARM_CORE ARM_CORE_ARM7TDMI`

ARM core model.

7.36.2.2 `#define port_wait_for_interrupt()`

Value:

```
{          \
    (*((volatile uint32_t *)0xFFFFFC04)) = 1;          \
}
```

AT91SAM7-specific wait for interrupt.

This implementation writes 1 into the PMC_SCDR register.

7.37 AT91SAM7 Interrupt Vectors

7.37.1 Detailed Description

Interrupt vectors for the AT91SAM7 family.

Functions

- void `_unhandled_exception` (void)
Unhandled exceptions handler.

7.37.2 Function Documentation

7.37.2.1 void `_unhandled_exception` (void)

Unhandled exceptions handler.

Any undefined exception vector points to this function by default. This function simply stops the system into an infinite loop.

Function Class:

Not an API, this function is for internal use only.

7.38 LPC214x Specific Parameters

7.38.1 Detailed Description

This file contains the ARM specific parameters for the LPC214x platform.

Defines

- `#define ARM_CORE ARM_CORE_ARM7TDMI`
ARM core model.
- `#define port_wait_for_interrupt()`
LPC214x-specific wait for interrupt code.

7.38.2 Define Documentation

7.38.2.1 `#define ARM_CORE ARM_CORE_ARM7TDMI`

ARM core model.

7.38.2.2 `#define port_wait_for_interrupt()`

Value:

```
{
    ((volatile uint32_t *) 0xE01FC0C0)) = 1; \
}
```

LPC214x-specific wait for interrupt code.

This implementation writes 1 into the PCON register.

7.39 LPC214x Interrupt Vectors

7.39.1 Detailed Description

Interrupt vectors for the LPC214x family.

Functions

- void [_unhandled_exception](#) (void)

Unhandled exceptions handler.

7.39.2 Function Documentation

7.39.2.1 void _unhandled_exception (void)

Unhandled exceptions handler.

Any undefined exception vector points to this function by default. This function simply stops the system into an infinite loop.

Function Class:

Not an API, this function is for internal use only.

Chapter 8

Data Structure Documentation

8.1 BaseAsynchronousChannel Struct Reference

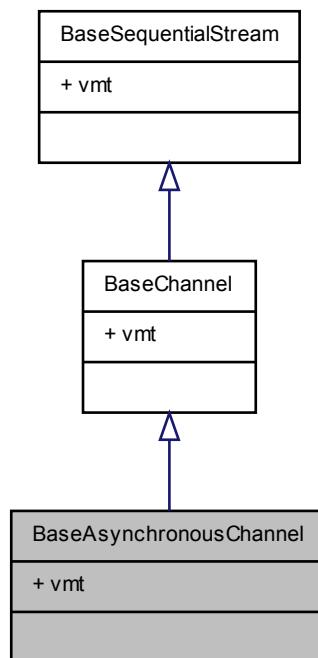
8.1.1 Detailed Description

Base asynchronous channel class.

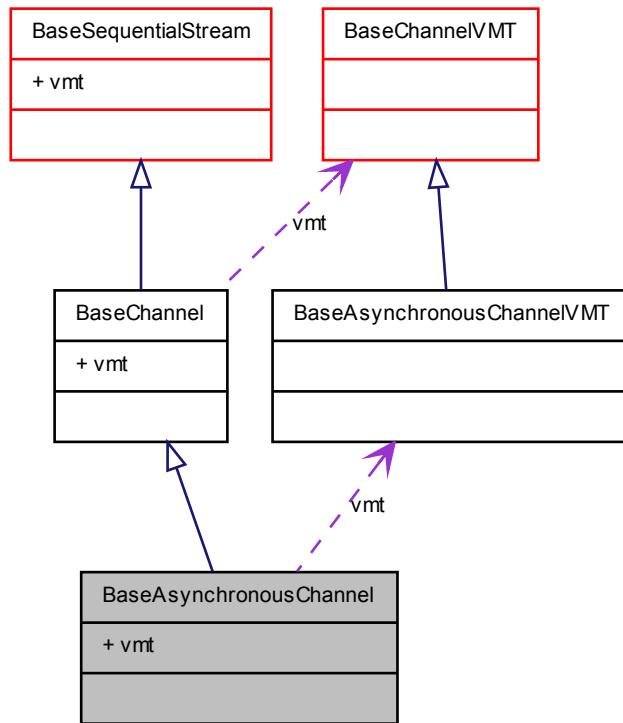
This class extends [BaseChannel](#) by adding event sources fields for asynchronous I/O for use in an event-driven environment.

```
#include <chioch.h>
```

Inheritance diagram for BaseAsynchronousChannel:



Collaboration diagram for BaseAsynchronousChannel:



Data Fields

- struct `BaseAsynchronousChannelVMT` * `vmt`
Virtual Methods Table.

8.1.2 Field Documentation

8.1.2.1 struct `BaseAsynchronousChannelVMT`* `BaseAsynchronousChannel::vmt`

Virtual Methods Table.

Reimplemented from `BaseChannel`.

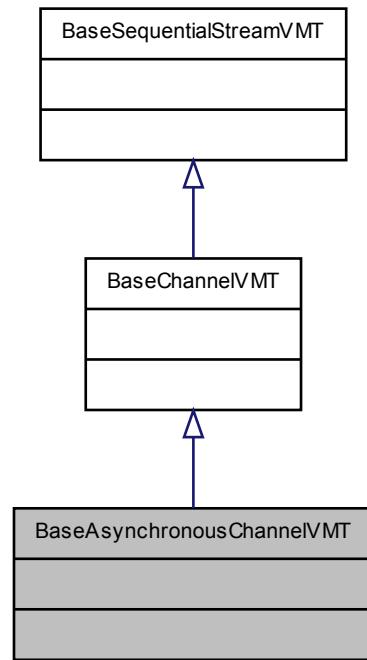
8.2 BaseAsynchronousChannelVMT Struct Reference

8.2.1 Detailed Description

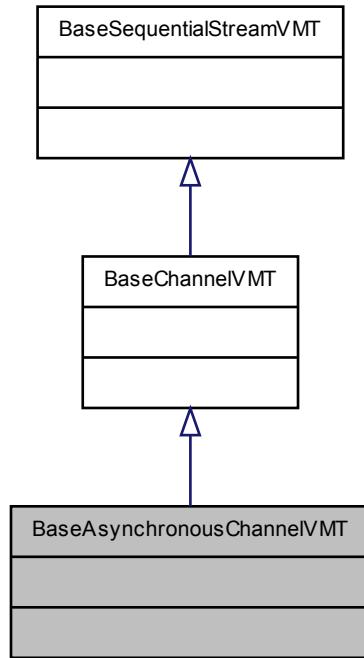
`BaseAsynchronousChannel` virtual methods table.

```
#include <chioch.h>
```

Inheritance diagram for BaseAsynchronousChannelVMT:



Collaboration diagram for BaseAsynchronousChannelVMT:



8.3 BaseChannel Struct Reference

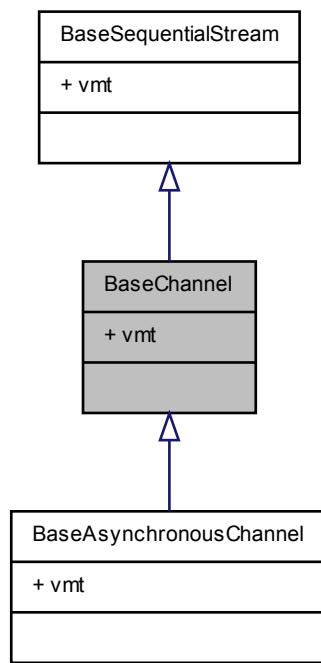
8.3.1 Detailed Description

Base channel class.

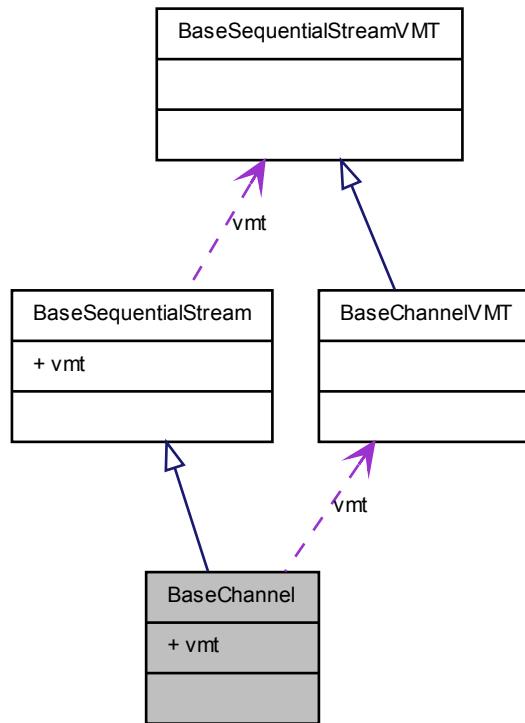
This class represents a generic, byte-wide, I/O channel. This class introduces generic I/O primitives with timeout specification.

```
#include <chioch.h>
```

Inheritance diagram for BaseChannel:



Collaboration diagram for BaseChannel:



Data Fields

- struct [BaseChannelVMT](#) * **vmt**
Virtual Methods Table.

8.3.2 Field Documentation

8.3.2.1 struct [BaseChannelVMT](#)* **BaseChannel::vmt**

Virtual Methods Table.

Reimplemented from [BaseSequentialStream](#).

Reimplemented in [BaseAsynchronousChannel](#).

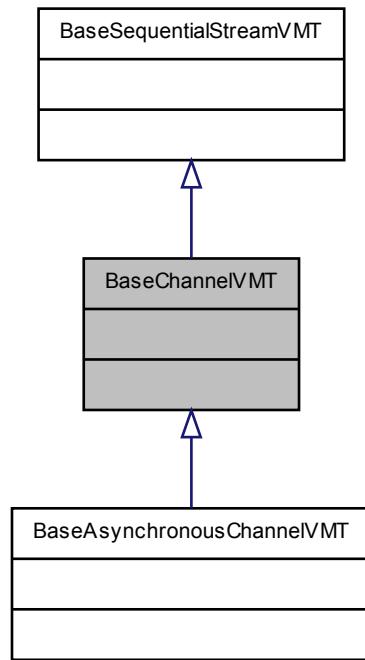
8.4 BaseChannelVMT Struct Reference

8.4.1 Detailed Description

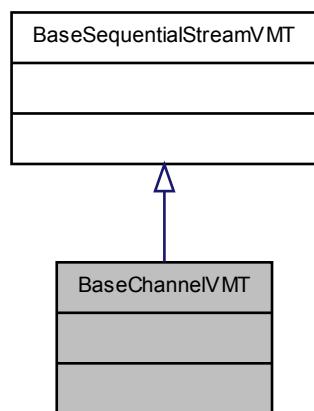
[BaseChannel](#) virtual methods table.

```
#include <chioch.h>
```

Inheritance diagram for BaseChannelVMT:



Collaboration diagram for BaseChannelVMT:



8.5 BaseFileStream Struct Reference

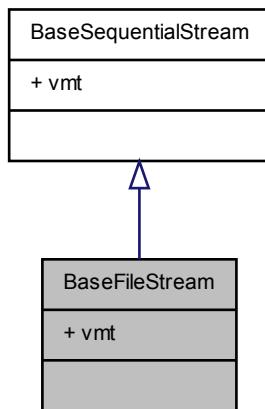
8.5.1 Detailed Description

Base file stream class.

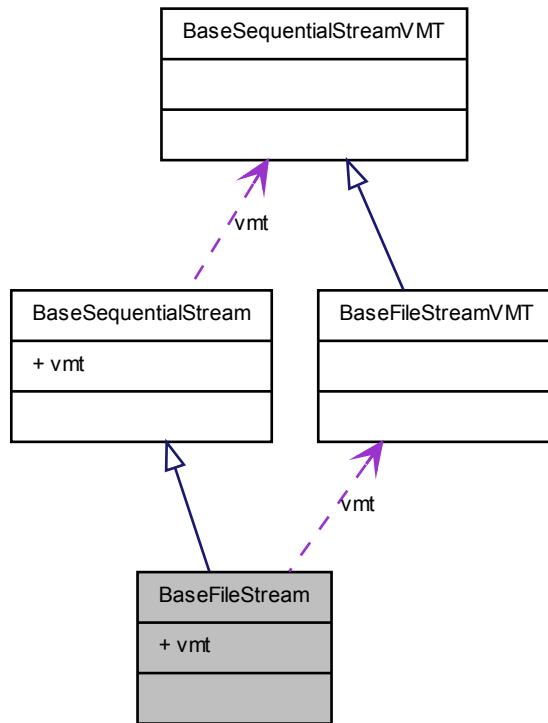
This class represents a generic file data stream.

```
#include <chfiles.h>
```

Inheritance diagram for BaseFileStream:



Collaboration diagram for BaseFileStream:



Data Fields

- struct [BaseFileStreamVMT](#) * vmt

Virtual Methods Table.

8.5.2 Field Documentation

8.5.2.1 struct [BaseFileStreamVMT](#)* **BaseFileStream::vmt**

Virtual Methods Table.

Reimplemented from [BaseSequentialStream](#).

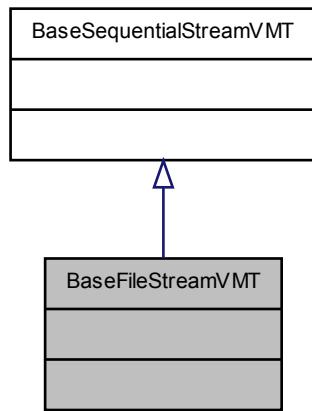
8.6 BaseFileStreamVMT Struct Reference

8.6.1 Detailed Description

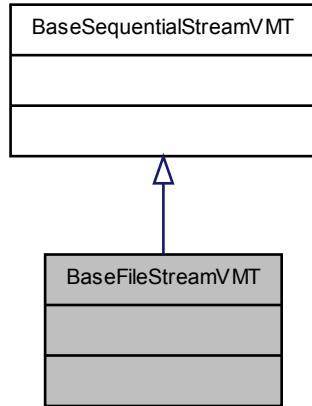
[BaseFileStream](#) virtual methods table.

```
#include <chfiles.h>
```

Inheritance diagram for BaseFileStreamVMT:



Collaboration diagram for BaseFileStreamVMT:



8.7 BaseSequentialStream Struct Reference

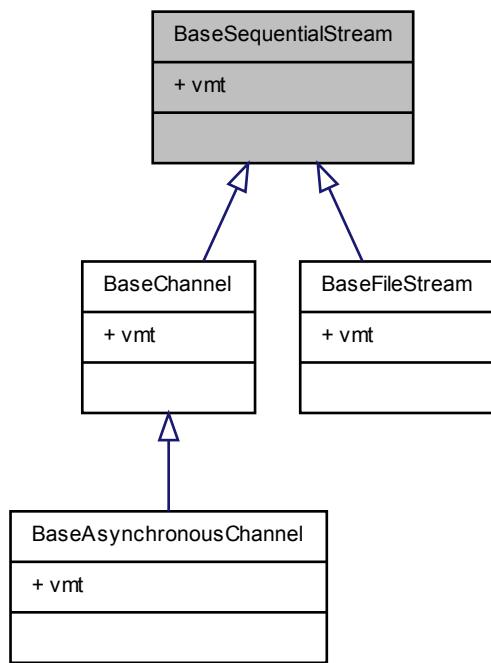
8.7.1 Detailed Description

Base stream class.

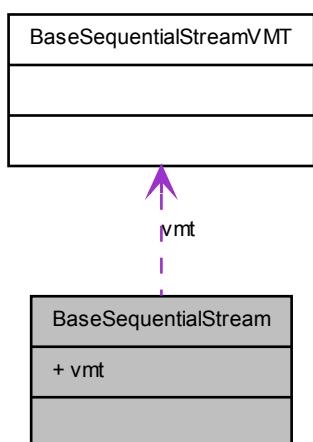
This class represents a generic blocking unbuffered sequential data stream.

```
#include <chstreams.h>
```

Inheritance diagram for BaseSequentialStream:



Collaboration diagram for BaseSequentialStream:



Data Fields

- struct [BaseSequentialStreamVMT](#) * vmt
Virtual Methods Table.

8.7.2 Field Documentation

8.7.2.1 struct [BaseSequentialStreamVMT](#)* [BaseSequentialStream::vmt](#)

Virtual Methods Table.

Reimplemented in [BaseFileStream](#), [BaseChannel](#), and [BaseAsynchronousChannel](#).

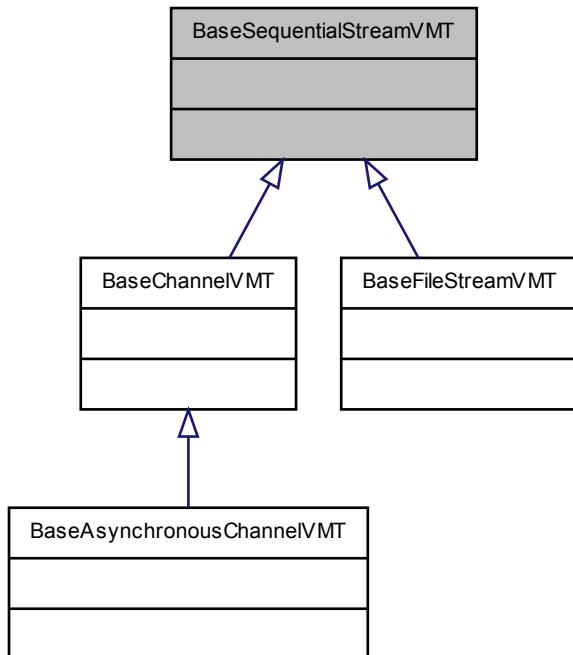
8.8 BaseSequentialStreamVMT Struct Reference

8.8.1 Detailed Description

[BaseSequentialStream](#) virtual methods table.

```
#include <chstreams.h>
```

Inheritance diagram for [BaseSequentialStreamVMT](#):



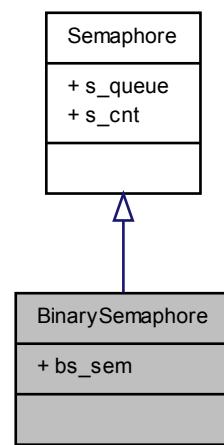
8.9 BinarySemaphore Struct Reference

8.9.1 Detailed Description

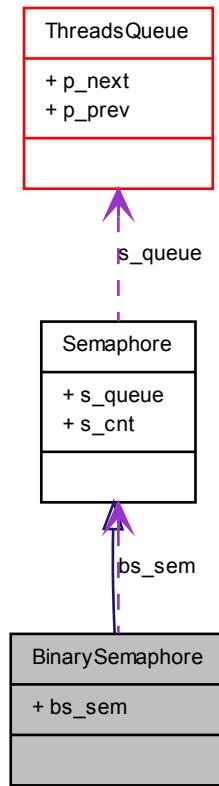
Binary semaphore type.

```
#include <chbsem.h>
```

Inheritance diagram for BinarySemaphore:



Collaboration diagram for BinarySemaphore:



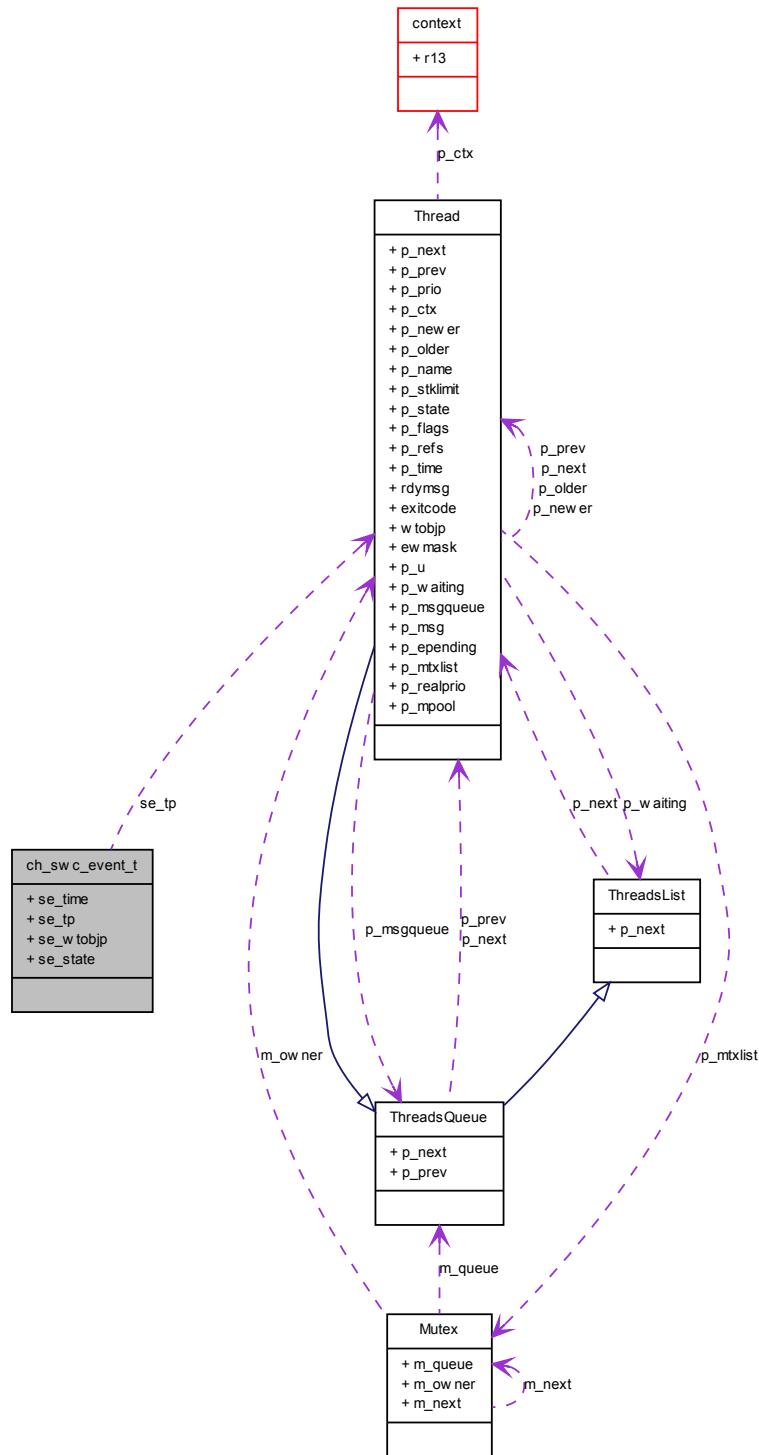
8.10 ch_swc_event_t Struct Reference

8.10.1 Detailed Description

Trace buffer record.

```
#include <chdebug.h>
```

Collaboration diagram for ch_swc_event_t:



Data Fields

- `systime_t se_time`

Time of the switch event.

- `Thread * se_tp`
Switched in thread.
- `void * se_wtobjp`
Object where going to sleep.
- `uint8_t se_state`
Switched out thread state.

8.10.2 Field Documentation

8.10.2.1 systime_t ch_swc_event_t::se_time

Time of the switch event.

8.10.2.2 Thread* ch_swc_event_t::se_tp

Switched in thread.

8.10.2.3 void* ch_swc_event_t::se_wtobjp

Object where going to sleep.

8.10.2.4 uint8_t ch_swc_event_t::se_state

Switched out thread state.

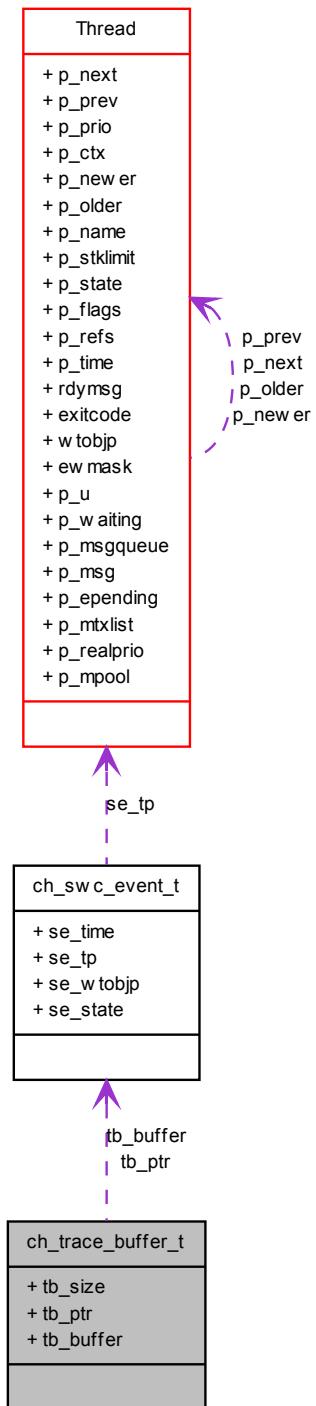
8.11 ch_trace_buffer_t Struct Reference

8.11.1 Detailed Description

Trace buffer header.

```
#include <chdebug.h>
```

Collaboration diagram for ch_trace_buffer_t:



Data Fields

- `unsigned tb_size`
Trace buffer size (entries).

- `ch_swc_event_t * tb_ptr`
Pointer to the buffer front.
- `ch_swc_event_t tb_buffer [CH_TRACE_BUFFER_SIZE]`
Ring buffer.

8.11.2 Field Documentation

8.11.2.1 `unsigned ch_trace_buffer_t::tb_size`

Trace buffer size (entries).

8.11.2.2 `ch_swc_event_t* ch_trace_buffer_t::tb_ptr`

Pointer to the buffer front.

8.11.2.3 `ch_swc_event_t ch_trace_buffer_t::tb_buffer[CH_TRACE_BUFFER_SIZE]`

Ring buffer.

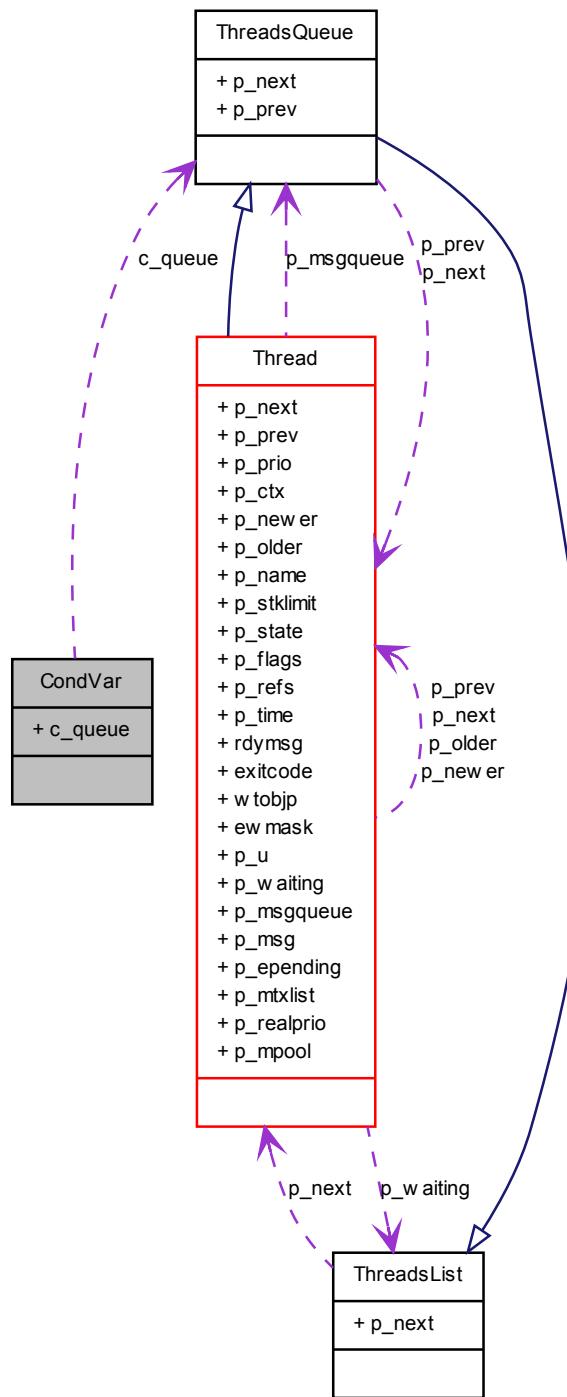
8.12 CondVar Struct Reference

8.12.1 Detailed Description

`CondVar` structure.

```
#include <chcond.h>
```

Collaboration diagram for CondVar:



Data Fields

- `ThreadsQueue c_queue`

CondVar threads queue.

8.12.2 Field Documentation

8.12.2.1 ThreadsQueue CondVar::c_queue

[CondVar](#) threads queue.

8.13 context Struct Reference

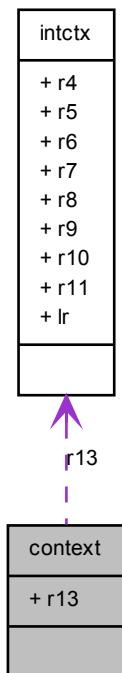
8.13.1 Detailed Description

Platform dependent part of the [Thread](#) structure.

In this port the structure just holds a pointer to the `intctx` structure representing the stack pointer at context switch time.

```
#include <chcore.h>
```

Collaboration diagram for context:



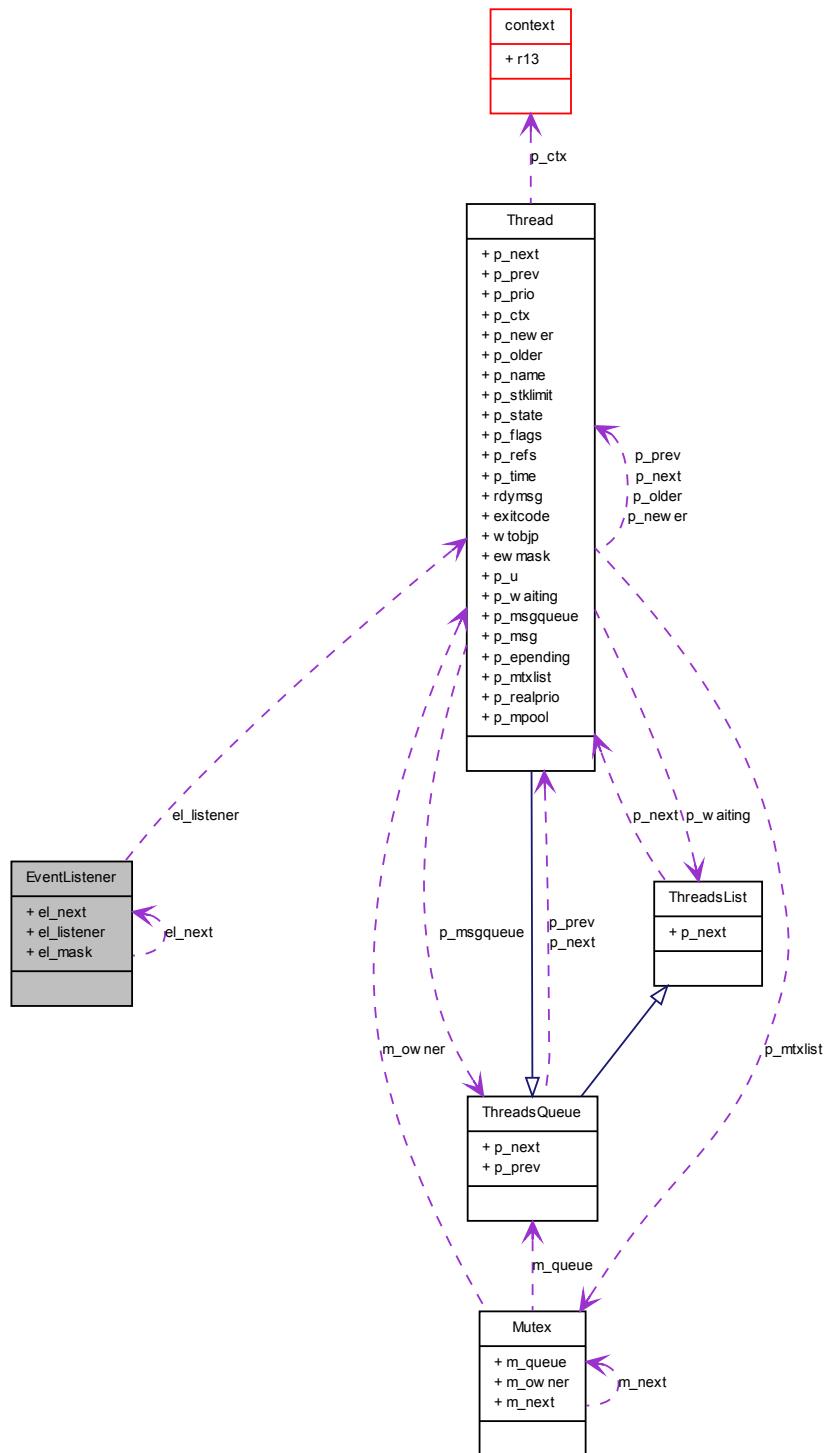
8.14 EventListener Struct Reference

8.14.1 Detailed Description

Event Listener structure.

```
#include <chevents.h>
```

Collaboration diagram for EventListener:



Data Fields

- `EventListener * el_next`

Next Event Listener registered on the Event Source.

- `Thread * el_listener`
Thread interested in the Event Source.
- `eventmask_t el_mask`
Event flags mask associated by the thread to the Event Source.

8.14.2 Field Documentation

8.14.2.1 `EventListener* EventListener::el_next`

Next Event Listener registered on the Event Source.

8.14.2.2 `Thread* EventListener::el_listener`

`Thread` interested in the Event Source.

8.14.2.3 `eventmask_t EventListener::el_mask`

Event flags mask associated by the thread to the Event Source.

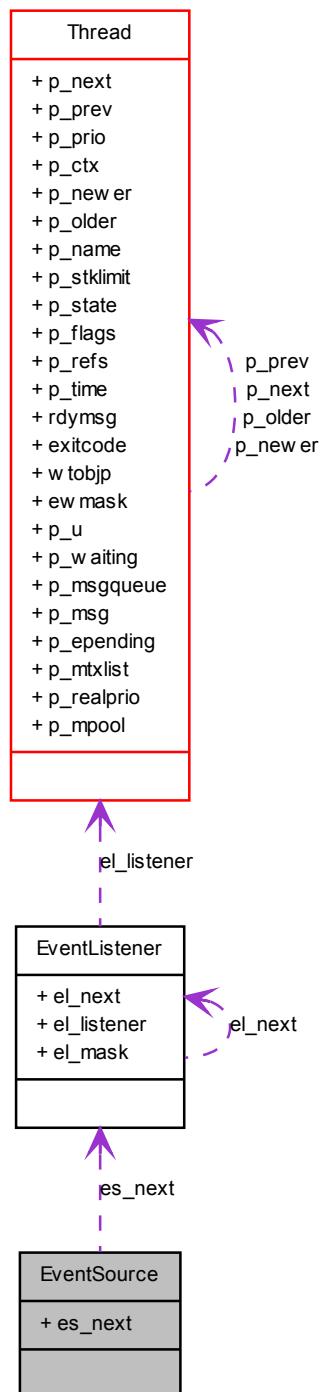
8.15 EventSource Struct Reference

8.15.1 Detailed Description

Event Source structure.

```
#include <chevents.h>
```

Collaboration diagram for EventSource:



Data Fields

- `EventListener * es_next`

First Event Listener registered on the Event Source.

8.15.2 Field Documentation

8.15.2.1 EventListener* EventSource::es_next

First Event Listener registered on the Event Source.

8.16 extctx Struct Reference

8.16.1 Detailed Description

Interrupt saved context.

This structure represents the stack frame saved during a preemption-capable interrupt handler.

```
#include <chcore.h>
```

8.17 GenericQueue Struct Reference

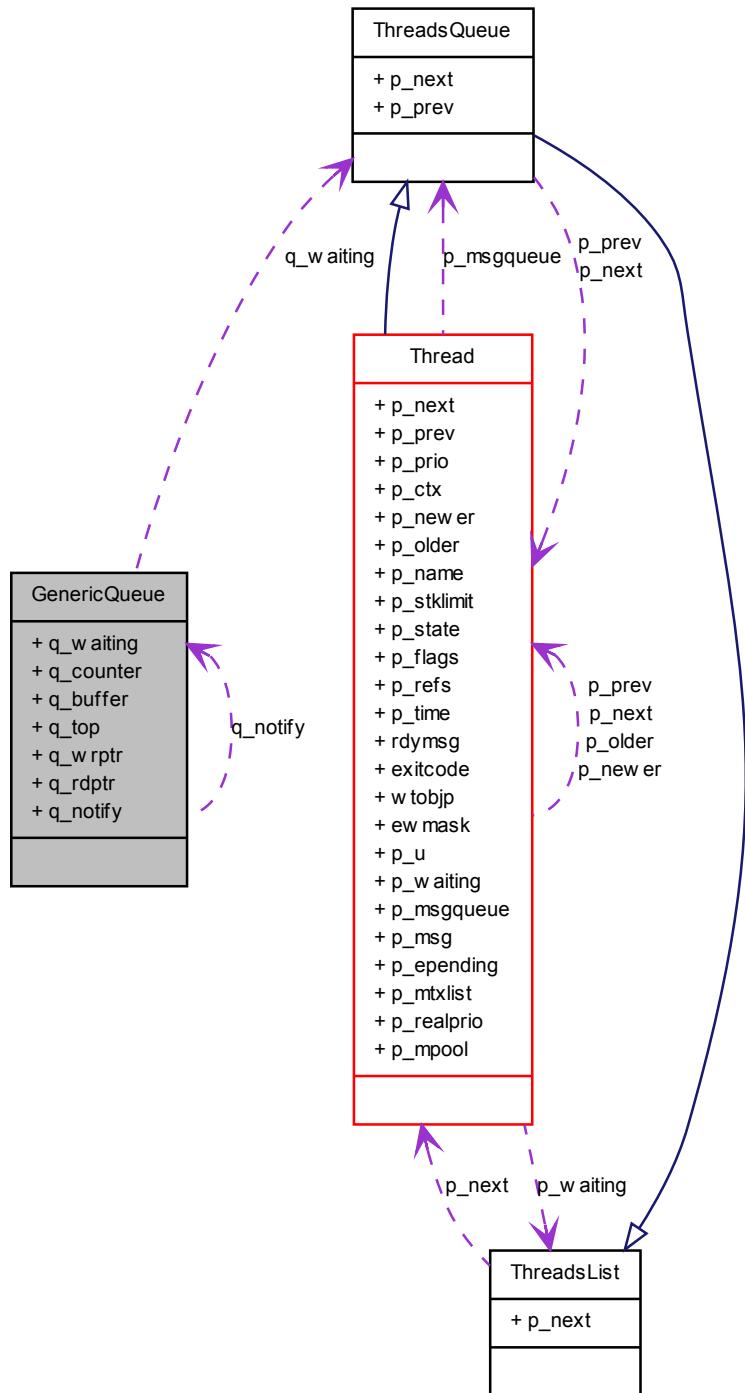
8.17.1 Detailed Description

Generic I/O queue structure.

This structure represents a generic Input or Output asymmetrical queue. The queue is asymmetrical because one end is meant to be accessed from a thread context, and thus can be blocking, the other end is accessible from interrupt handlers or from within a kernel lock zone (see **I-Locked** and **S-Locked** states in [System States](#)) and is non-blocking.

```
#include <chqueues.h>
```

Collaboration diagram for GenericQueue:



Data Fields

- **ThreadsQueue q_waiting**

Queue of waiting threads.

- `size_t q_counter`
Resources counter.
- `uint8_t * q_buffer`
Pointer to the queue buffer.
- `uint8_t * q_top`
Pointer to the first location after the buffer.
- `uint8_t * q_wptr`
Write pointer.
- `uint8_t * q_rptr`
Read pointer.
- `qnotify_t q_notify`
Data notification callback.

8.17.2 Field Documentation

8.17.2.1 ThreadsQueue GenericQueue::q_waiting

Queue of waiting threads.

8.17.2.2 size_t GenericQueue::q_counter

Resources counter.

8.17.2.3 uint8_t* GenericQueue::q_buffer

Pointer to the queue buffer.

8.17.2.4 uint8_t* GenericQueue::q_top

Pointer to the first location after the buffer.

8.17.2.5 uint8_t* GenericQueue::q_wptr

Write pointer.

8.17.2.6 uint8_t* GenericQueue::q_rptr

Read pointer.

8.17.2.7 qnotify_t GenericQueue::q_notify

Data notification callback.

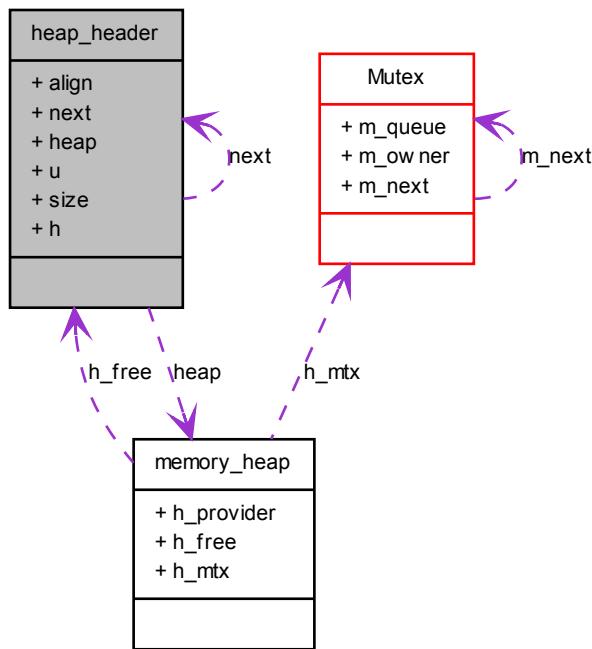
8.18 heap_header Union Reference

8.18.1 Detailed Description

Memory heap block header.

```
#include <chheap.h>
```

Collaboration diagram for heap_header:



8.18.2 Field Documentation

8.18.2.1 union heap_header* heap_header::next

Next block in free list.

8.18.2.2 MemoryHeap* heap_header::heap

Block owner heap.

8.18.2.3 union { ... } heap_header::u

Overlapped fields.

8.18.2.4 size_t heap_header::size

Size of the memory block.

8.19 intctx Struct Reference

8.19.1 Detailed Description

System saved context.

This structure represents the inner stack frame during a context switching.

```
#include <chcore.h>
```

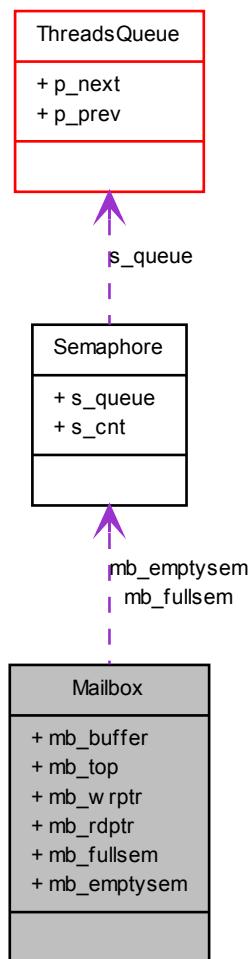
8.20 Mailbox Struct Reference

8.20.1 Detailed Description

Structure representing a mailbox object.

```
#include <chmboxes.h>
```

Collaboration diagram for Mailbox:



Data Fields

- `msg_t * mb_buffer`
Pointer to the mailbox buffer.
- `msg_t * mb_top`

Pointer to the location after the buffer.

- **msg_t * mb_wptr**
Write pointer.
- **msg_t * mb_rdptr**
Read pointer.
- **Semaphore mb_fullsem**
Full counter [Semaphore](#).
- **Semaphore mb_emptysem**
Empty counter [Semaphore](#).

8.20.2 Field Documentation

8.20.2.1 msg_t* Mailbox::mb_buffer

Pointer to the mailbox buffer.

8.20.2.2 msg_t* Mailbox::mb_top

Pointer to the location after the buffer.

8.20.2.3 msg_t* Mailbox::mb_wptr

Write pointer.

8.20.2.4 msg_t* Mailbox::mb_rdptr

Read pointer.

8.20.2.5 Semaphore Mailbox::mb_fullsem

Full counter [Semaphore](#).

8.20.2.6 Semaphore Mailbox::mb_emptysem

Empty counter [Semaphore](#).

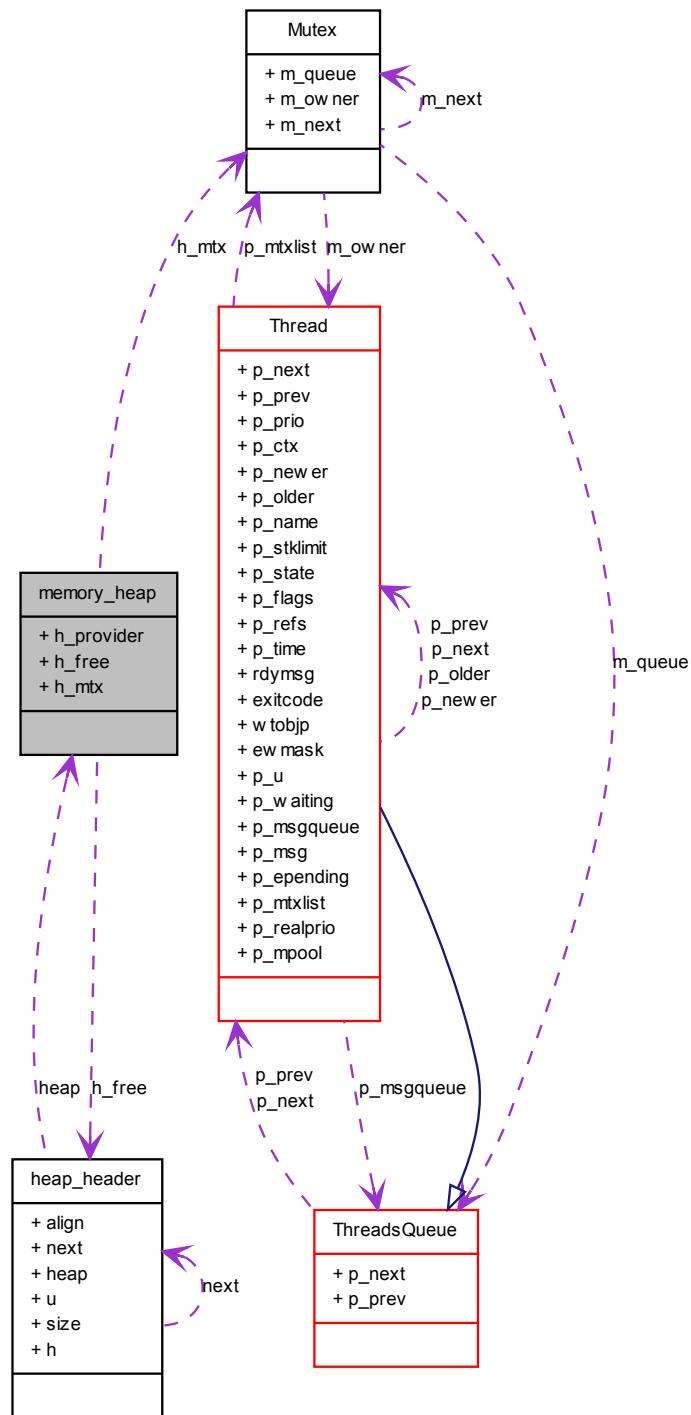
8.21 memory_heap Struct Reference

8.21.1 Detailed Description

Structure describing a memory heap.

```
#include <chheap.h>
```

Collaboration diagram for memory_heap:



Data Fields

- `memgetfunc_t h_provider`

Memory blocks provider for this heap.

- union **heap_header h_free**
Free blocks list header.
- **Mutex h_mtx**
Heap access mutex.

8.21.2 Field Documentation

8.21.2.1 memgetfunc_t memory_heap::h_provider

Memory blocks provider for this heap.

8.21.2.2 union heap_header memory_heap::h_free

Free blocks list header.

8.21.2.3 Mutex memory_heap::h_mtx

Heap access mutex.

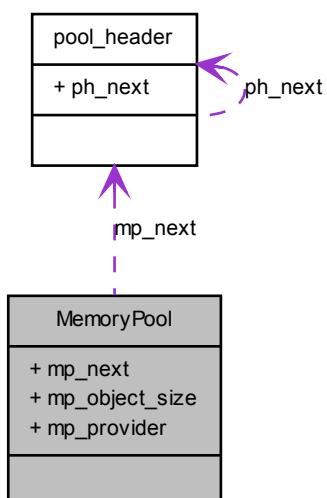
8.22 MemoryPool Struct Reference

8.22.1 Detailed Description

Memory pool descriptor.

```
#include <chmempools.h>
```

Collaboration diagram for MemoryPool:



Data Fields

- struct [pool_header](#) * **mp_next**
Pointer to the header.
- size_t **mp_object_size**
Memory pool objects size.
- memgetfunc_t **mp_provider**
Memory blocks provider for this pool.

8.22.2 Field Documentation

8.22.2.1 struct pool_header* MemoryPool::mp_next

Pointer to the header.

8.22.2.2 size_t MemoryPool::mp_object_size

Memory pool objects size.

8.22.2.3 memgetfunc_t MemoryPool::mp_provider

Memory blocks provider for this pool.

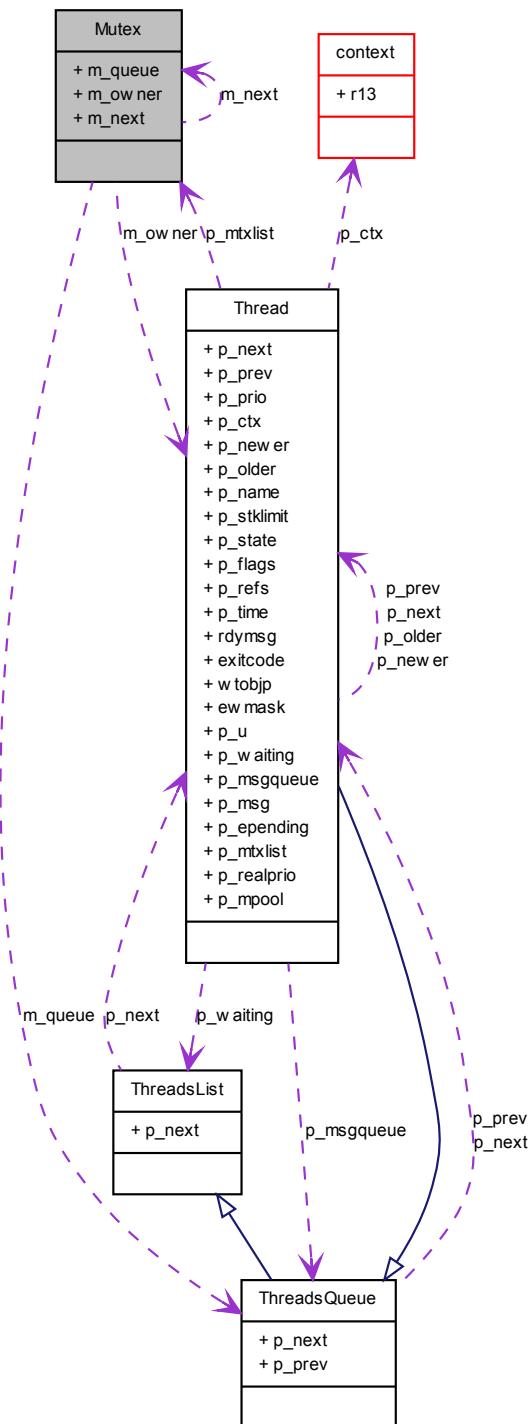
8.23 Mutex Struct Reference

8.23.1 Detailed Description

[Mutex](#) structure.

```
#include <chmtx.h>
```

Collaboration diagram for Mutex:



Data Fields

- [ThreadsQueue m_queue](#)

Queue of the threads sleeping on this Mutex.

- `Thread * m_owner`
Owner `Thread` pointer or `NULL`.
- struct `Mutex * m_next`
Next `Mutex` into an owner-list or `NULL`.

8.23.2 Field Documentation

8.23.2.1 ThreadsQueue Mutex::m_queue

Queue of the threads sleeping on this `Mutex`.

8.23.2.2 Thread* Mutex::m_owner

Owner `Thread` pointer or `NULL`.

8.23.2.3 struct Mutex* Mutex::m_next

Next `Mutex` into an owner-list or `NULL`.

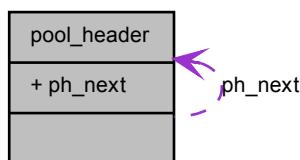
8.24 pool_header Struct Reference

8.24.1 Detailed Description

Memory pool free object header.

```
#include <chmempools.h>
```

Collaboration diagram for pool_header:



Data Fields

- struct `pool_header * ph_next`
Pointer to the next pool header in the list.

8.24.2 Field Documentation

8.24.2.1 struct pool_header* pool_header::ph_next

Pointer to the next pool header in the list.

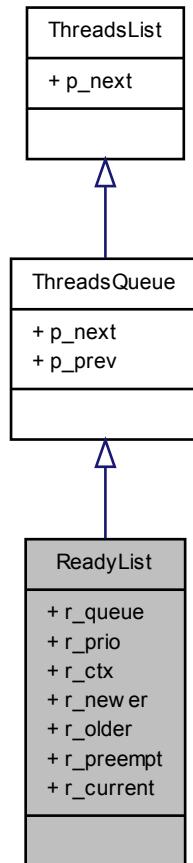
8.25 ReadyList Struct Reference

8.25.1 Detailed Description

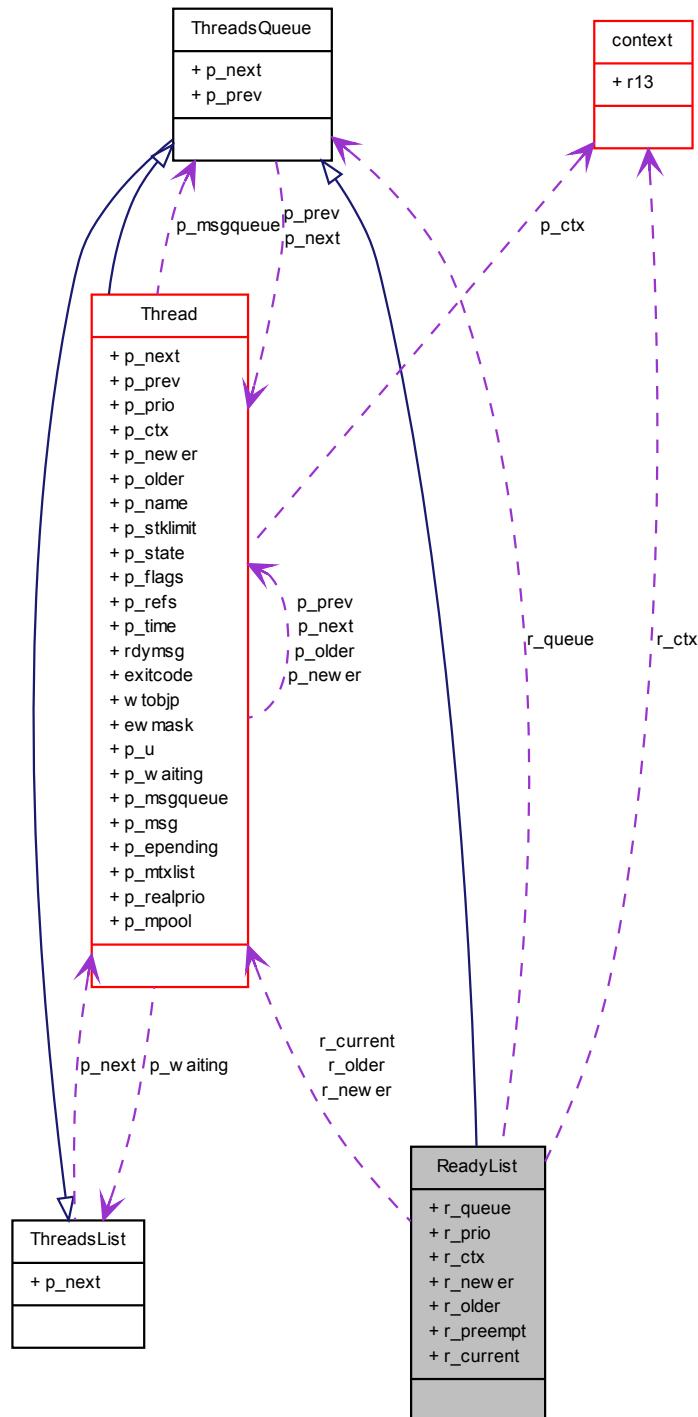
Ready list header.

```
#include <chsched.h>
```

Inheritance diagram for ReadyList:



Collaboration diagram for ReadyList:



Data Fields

- [ThreadsQueue r_queue](#)

Threads queue.

- `tprio_t r_prio`
This field must be initialized to zero.
- struct `context r_ctx`
Not used, present because offsets.
- `Thread * r_newer`
Newer registry element.
- `Thread * r_older`
Older registry element.
- `cnt_t r_preempt`
Round robin counter.
- `Thread * r_current`
The currently running thread.

8.25.2 Field Documentation

8.25.2.1 ThreadsQueue ReadyList::r_queue

Threads queue.

8.25.2.2 tprio_t ReadyList::r_prio

This field must be initialized to zero.

8.25.2.3 struct context ReadyList::r_ctx

Not used, present because offsets.

8.25.2.4 Thread* ReadyList::r_newer

Newer registry element.

8.25.2.5 Thread* ReadyList::r_older

Older registry element.

8.25.2.6 cnt_t ReadyList::r_preempt

Round robin counter.

8.25.2.7 Thread* ReadyList::r_current

The currently running thread.

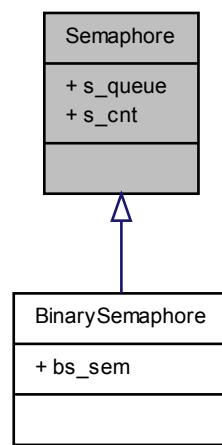
8.26 Semaphore Struct Reference

8.26.1 Detailed Description

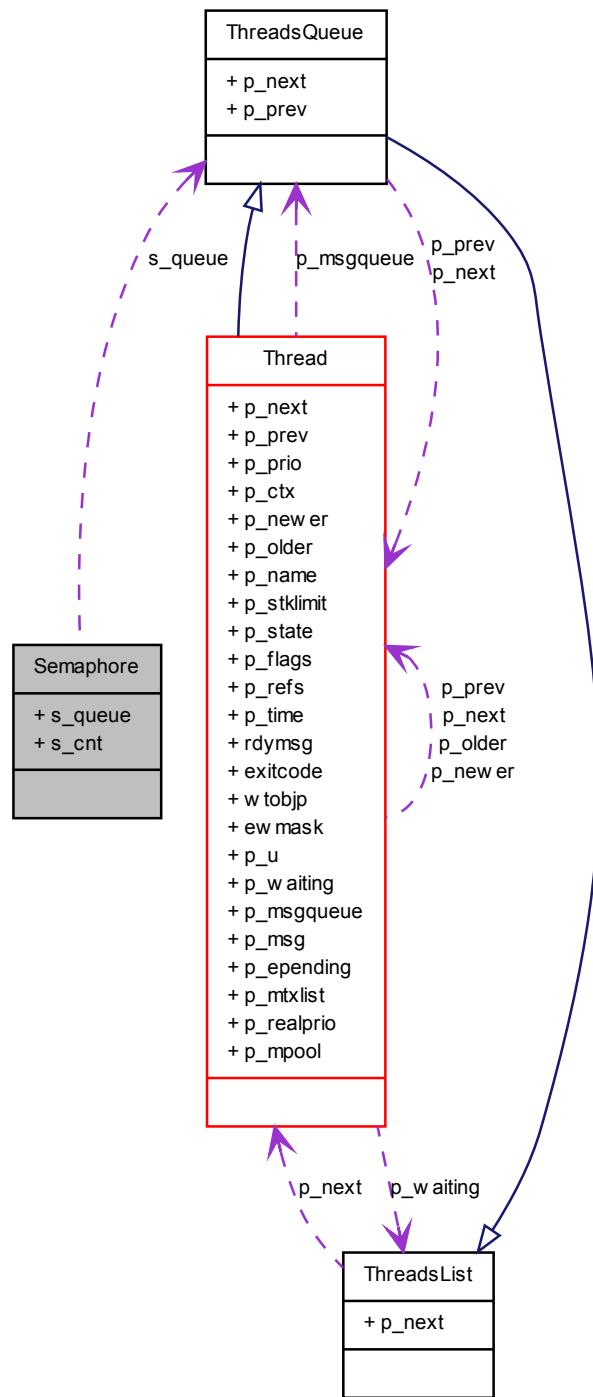
Semaphore structure.

```
#include <chsem.h>
```

Inheritance diagram for Semaphore:



Collaboration diagram for Semaphore:



Data Fields

- **ThreadsQueue s_queue**

Queue of the threads sleeping on this semaphore.

- `cnt_t s_cnt`
The semaphore counter.

8.26.2 Field Documentation

8.26.2.1 ThreadsQueue Semaphore::s_queue

Queue of the threads sleeping on this semaphore.

8.26.2.2 cnt_t Semaphore::s_cnt

The semaphore counter.

8.27 Thread Struct Reference

8.27.1 Detailed Description

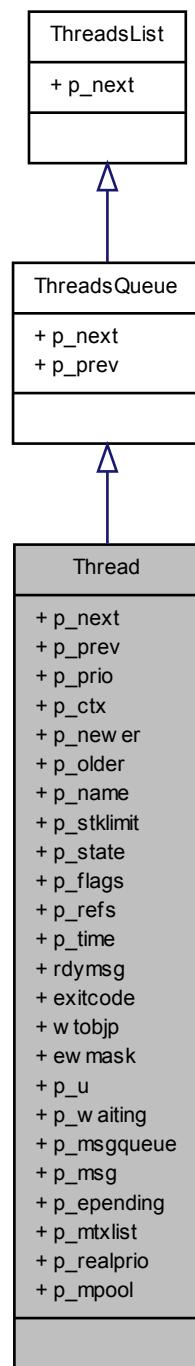
Structure representing a thread.

Note

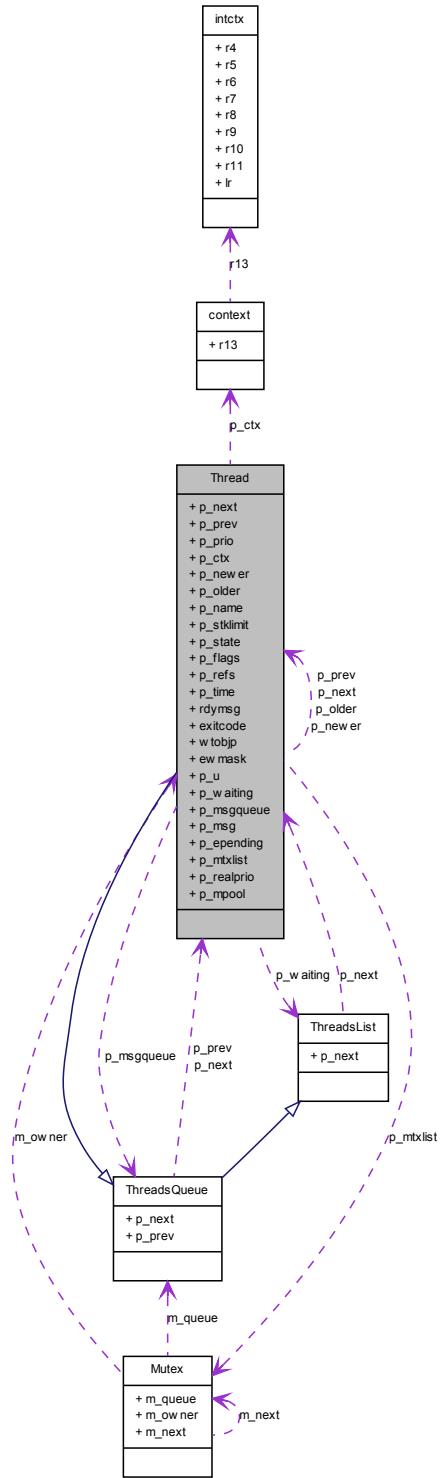
Not all the listed fields are always needed, by switching off some not needed ChibiOS/RT subsystems it is possible to save RAM space by shrinking the `Thread` structure.

```
#include <chthreads.h>
```

Inheritance diagram for Thread:



Collaboration diagram for Thread:



Data Fields

- `Thread * p_next`

Next in the list/queue.

- `Thread * p_prev`
Previous in the queue.
- `tprio_t p_prio`
Thread priority.
- `struct context p_ctxt`
Processor context.
- `Thread * p_newer`
Newer registry element.
- `Thread * p_older`
Older registry element.
- `const char * p_name`
Thread name or NULL.
- `stkalign_t * p_stklimit`
Thread stack boundary.
- `tstate_t p_state`
Current thread state.
- `tmode_t p_flags`
Various thread flags.
- `trefs_t p_refs`
References to this thread.
- `volatile systime_t p_time`
Thread consumed time in ticks.
- `ThreadsList p_waiting`
Termination waiting list.
- `ThreadsQueue p_msgqueue`
Messages queue.
- `msg_t p_msg`
Thread message.
- `eventmask_t p_epending`
Pending events mask.
- `Mutex * p_mtxlist`
List of the mutexes owned by this thread.
- `tprio_t p_realprio`
Thread's own, non-inherited, priority.
- `void * p_mpool`
Memory Pool where the thread workspace is returned.
- `msg_t rdymsg`
Thread wakeup code.
- `msg_t exitcode`
Thread exit code.
- `void * wtobjp`
Pointer to a generic "wait" object.
- `eventmask_t ewmask`
Enabled events mask.

8.27.2 Field Documentation

8.27.2.1 `Thread* Thread::p_next`

Next in the list/queue.

Reimplemented from [ThreadsQueue](#).

8.27.2.2 Thread* Thread::p_prev

Previous in the queue.

Reimplemented from [ThreadsQueue](#).

8.27.2.3 tprio_t Thread::p_prio

[Thread](#) priority.

8.27.2.4 struct context Thread::p_ctx

Processor context.

8.27.2.5 Thread* Thread::p_newer

Newer registry element.

8.27.2.6 Thread* Thread::p_older

Older registry element.

8.27.2.7 const char* Thread::p_name

[Thread](#) name or NULL.

8.27.2.8 stkalign_t* Thread::p_stklimit

[Thread](#) stack boundary.

8.27.2.9 tstate_t Thread::p_state

Current thread state.

8.27.2.10 tmode_t Thread::p_flags

Various thread flags.

8.27.2.11 trefs_t Thread::p_refs

References to this thread.

8.27.2.12 volatile systime_t Thread::p_time

[Thread](#) consumed time in ticks.

Note

This field can overflow.

8.27.2.13 msg_t Thread::rdymsg

[Thread](#) wakeup code.

Note

This field contains the low level message sent to the thread by the waking thread or interrupt handler. The value is valid after exiting the [chSchWakeupS\(\)](#) function.

8.27.2.14 msg_t Thread::exitcode

[Thread](#) exit code.

Note

The thread termination code is stored in this field in order to be retrieved by the thread performing a [chThdWait\(\)](#) on this thread.

8.27.2.15 void* Thread::wtobjp

Pointer to a generic "wait" object.

Note

This field is used to get a generic pointer to a synchronization object and is valid when the thread is in one of the wait states.

8.27.2.16 eventmask_t Thread::ewmask

Enabled events mask.

Note

This field is only valid while the thread is in the THD_STATE_WTOREVT or THD_STATE_WTANDEVT states.

8.27.2.17 ThreadsList Thread::p_waiting

Termination waiting list.

8.27.2.18 ThreadsQueue Thread::p_msgqueue

Messages queue.

8.27.2.19 msg_t Thread::p_msg

[Thread](#) message.

8.27.2.20 eventmask_t Thread::p_epending

Pending events mask.

8.27.2.21 Mutex* Thread::p_mtxlist

List of the mutexes owned by this thread.

Note

The list is terminated by a `NULL` in this field.

8.27.2.22 tprio_t Thread::p_realprio

Thread's own, non-inherited, priority.

8.27.2.23 void* Thread::p_mpool

Memory Pool where the thread workspace is returned.

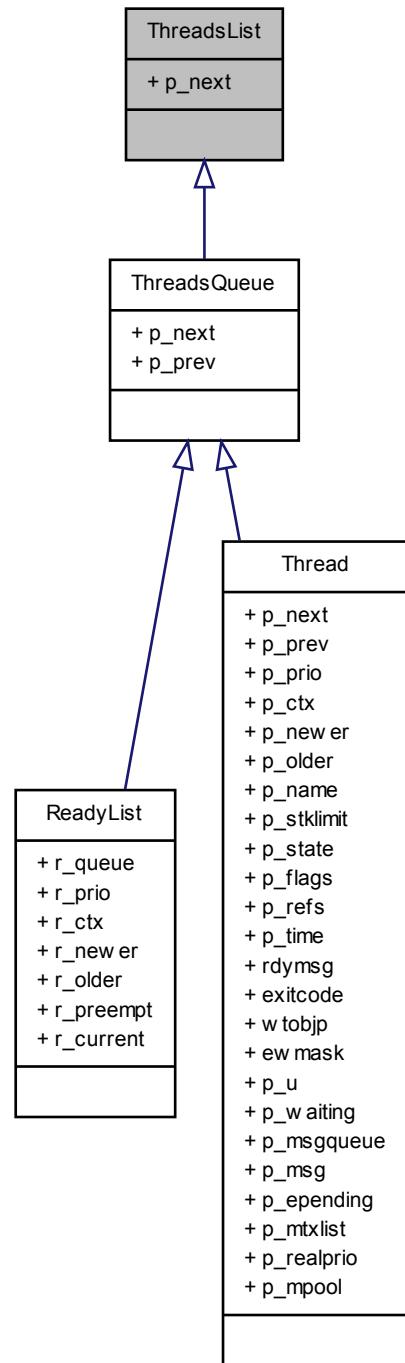
8.28 ThreadsList Struct Reference

8.28.1 Detailed Description

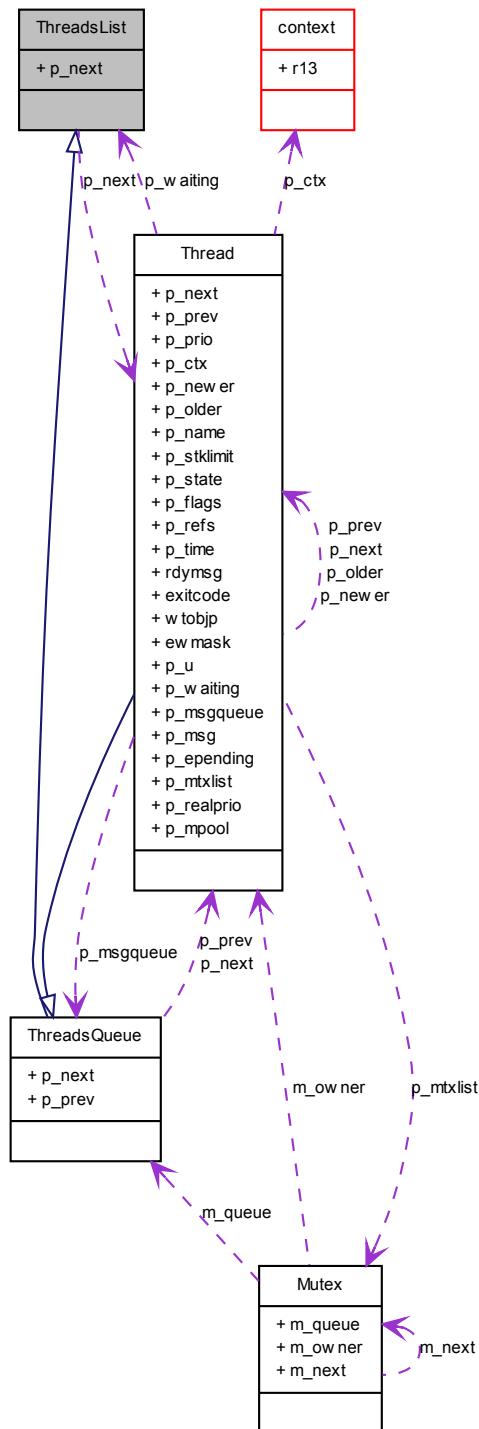
Generic threads single link list, it works like a stack.

```
#include <chlists.h>
```

Inheritance diagram for ThreadsList:



Collaboration diagram for ThreadsList:



Data Fields

- `Thread * p_next`

8.28.2 Field Documentation

8.28.2.1 Thread* ThreadsList::p_next

Last pushed [Thread](#) on the stack list, or pointer to itself if empty.

Reimplemented in [ThreadsQueue](#), and [Thread](#).

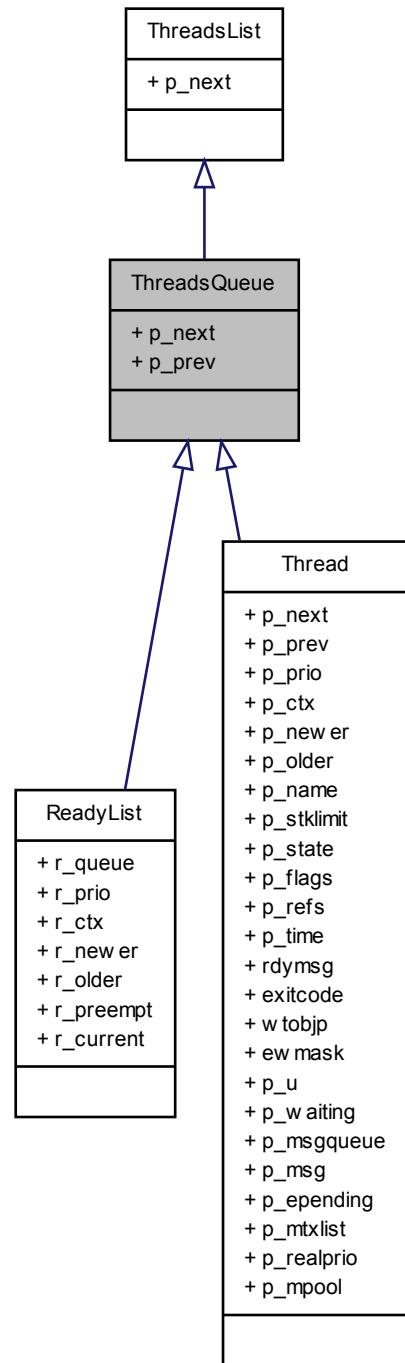
8.29 ThreadsQueue Struct Reference

8.29.1 Detailed Description

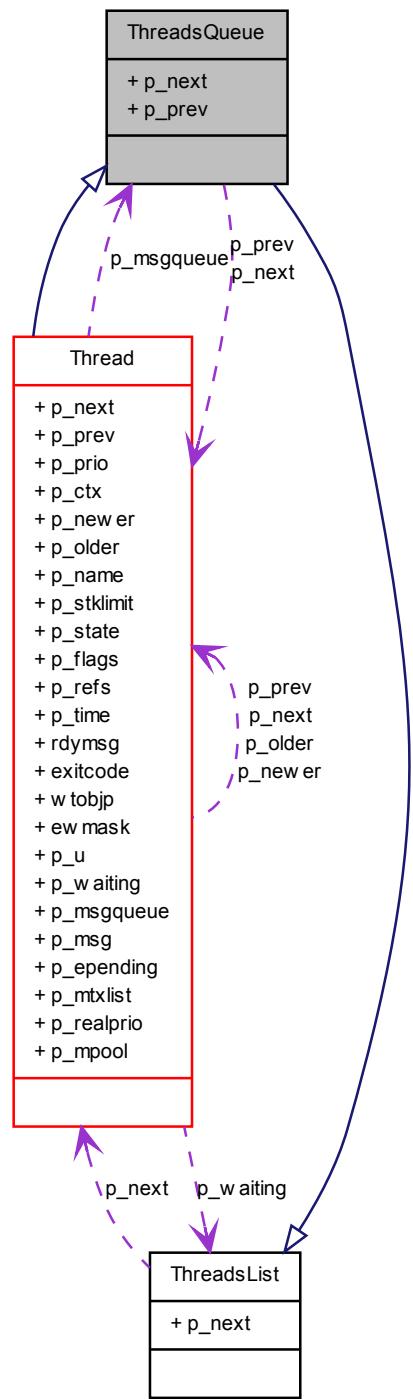
Generic threads bidirectional linked list header and element.

```
#include <chlists.h>
```

Inheritance diagram for ThreadsQueue:



Collaboration diagram for ThreadsQueue:



Data Fields

- `Thread * p_next`
- `Thread * p_prev`

8.29.2 Field Documentation

8.29.2.1 Thread* ThreadsQueue::p_next

First [Thread](#) in the queue, or [ThreadQueue](#) when empty.

Reimplemented from [ThreadsList](#).

Reimplemented in [Thread](#).

8.29.2.2 Thread* ThreadsQueue::p_prev

Last [Thread](#) in the queue, or [ThreadQueue](#) when empty.

Reimplemented in [Thread](#).

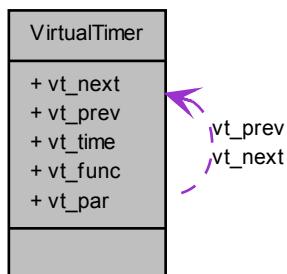
8.30 VirtualTimer Struct Reference

8.30.1 Detailed Description

Virtual Timer descriptor structure.

```
#include <chvt.h>
```

Collaboration diagram for VirtualTimer:



Data Fields

- [VirtualTimer * vt_next](#)
Next timer in the delta list.
- [VirtualTimer * vt_prev](#)
Previous timer in the delta list.
- [systime_t vt_time](#)
Time delta before timeout.
- [vfunc_t vt_func](#)
Timer callback function pointer.
- [void * vt_par](#)
Timer callback function parameter.

8.30.2 Field Documentation

8.30.2.1 VirtualTimer* VirtualTimer::vt_next

Next timer in the delta list.

8.30.2.2 VirtualTimer* VirtualTimer::vt_prev

Previous timer in the delta list.

8.30.2.3 systime_t VirtualTimer::vt_time

Time delta before timeout.

8.30.2.4 vfunc_t VirtualTimer::vt_func

Timer callback function pointer.

8.30.2.5 void* VirtualTimer::vt_par

Timer callback function parameter.

8.31 VTLIST Struct Reference

8.31.1 Detailed Description

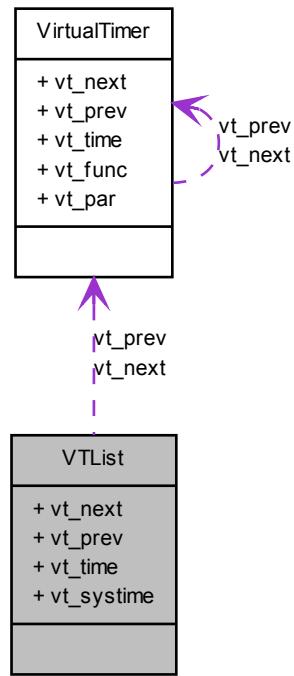
Virtual timers list header.

Note

The delta list is implemented as a double link bidirectional list in order to make the unlink time constant, the reset of a virtual timer is often used in the code.

```
#include <chvt.h>
```

Collaboration diagram for VTLIST:



Data Fields

- `VirtualTimer * vt_next`
Next timer in the delta list.
- `VirtualTimer * vt_prev`
Last timer in the delta list.
- `systime_t vt_time`
Must be initialized to -1.
- `volatile systime_t vt_systime`
System Time counter.

8.31.2 Field Documentation

8.31.2.1 `VirtualTimer* VTLIST::vt_next`

Next timer in the delta list.

8.31.2.2 `VirtualTimer* VTLIST::vt_prev`

Last timer in the delta list.

8.31.2.3 systime_t VTList::vt_time

Must be initialized to -1.

8.31.2.4 volatile systime_t VTList::vt_systime

System Time counter.

Chapter 9

File Documentation

9.1 armparams.h File Reference

9.1.1 Detailed Description

ARM7 AT91SAM7 Specific Parameters.

Defines

- `#define ARM_CORE ARM_CORE_ARM7TDMI`
ARM core model.
- `#define port_wait_for_interrupt()`
AT91SAM7-specific wait for interrupt.

9.2 armparams.h File Reference

9.2.1 Detailed Description

ARM7 LPC214x Specific Parameters.

Defines

- `#define ARM_CORE ARM_CORE_ARM7TDMI`
ARM core model.
- `#define port_wait_for_interrupt()`
LPC214x-specific wait for interrupt code.

9.3 ch.h File Reference

9.3.1 Detailed Description

ChibiOS/RT main include file. This header includes all the required kernel headers so it is the only kernel header you usually want to include in your application. `#include "chconf.h"`

```
#include "ctypes.h"  
#include "chlists.h"
```

```
#include "chcore.h"
#include "chsys.h"
#include "chvt.h"
#include "chsched.h"
#include "chsem.h"
#include "chbsem.h"
#include "chmtx.h"
#include "chcond.h"
#include "chevents.h"
#include "chmsg.h"
#include "chmboxes.h"
#include "chmemcore.h"
#include "chheap.h"
#include "chmempools.h"
#include "chthreads.h"
#include "chdynamic.h"
#include "chregistry.h"
#include "chinline.h"
#include "chqueues.h"
#include "chstreams.h"
#include "chioch.h"
#include "chfiles.h"
#include "chdebug.h"
```

Functions

- void [_idle_thread](#) (void *p)
This function implements the idle thread infinite loop.

Defines

- #define [_CHIBIOS_RT_](#)
ChibiOS/RT identification macro.
- #define [CH_KERNEL_VERSION](#) "2.4.0"
Kernel version string.

Kernel version

- #define [CH_KERNEL_MAJOR](#) 2
Kernel version major number.
- #define [CH_KERNEL_MINOR](#) 4
Kernel version minor number.
- #define [CH_KERNEL_PATCH](#) 0
Kernel version patch number.

9.4 chbsem.h File Reference

9.4.1 Detailed Description

Binary semaphores structures and macros.

Data Structures

- struct [BinarySemaphore](#)

Binary semaphore type.

Defines

- `#define _BSEMAPHORE_DATA(name, taken) {_SEMAPHORE_DATA(name.bs_sem, ((taken) ? 0 : 1))}`
Data part of a static semaphore initializer.
- `#define BSEMAPHORE_DECL(name, taken) BinarySemaphore name = _BSEMAPHORE_DATA(name, taken)`
Static semaphore initializer.

Macro Functions

- `#define chBSemInit(bsp, taken) chSemInit(&(bsp)->bs_sem, (taken) ? 0 : 1)`
Initializes a binary semaphore.
- `#define chBSemWait(bsp) chSemWait(&(bsp)->bs_sem)`
Wait operation on the binary semaphore.
- `#define chBSemWaitS(bsp) chSemWaitS(&(bsp)->bs_sem)`
Wait operation on the binary semaphore.
- `#define chBSemWaitTimeout(bsp, time) chSemWaitTimeout(&(bsp)->bs_sem, (time))`
Wait operation on the binary semaphore.
- `#define chBSemWaitTimeoutS(bsp, time) chSemWaitTimeoutS(&(bsp)->bs_sem, (time))`
Wait operation on the binary semaphore.
- `#define chBSemReset(bsp, taken) chSemReset(&(bsp)->bs_sem, (taken) ? 0 : 1)`
Reset operation on the binary semaphore.
- `#define chBSemResetI(bsp, taken) chSemResetI(&(bsp)->bs_sem, (taken) ? 0 : 1)`
Reset operation on the binary semaphore.
- `#define chBSemSignal(bsp)`
Performs a signal operation on a binary semaphore.
- `#define chBSemSignall(bsp)`
Performs a signal operation on a binary semaphore.
- `#define chBSemGetStateI(bsp) ((bsp)->bs_sem.s_cnt > 0 ? FALSE : TRUE)`
Returns the binary semaphore current state.

9.5 chcond.c File Reference

9.5.1 Detailed Description

Condition Variables code. `#include "ch.h"`

Functions

- `void chCondInit (CondVar *cp)`
Initializes a [CondVar](#) structure.
- `void chCondSignal (CondVar *cp)`

- `void chCondSignal (CondVar *cp)`
Signals one thread that is waiting on the condition variable.
- `void chCondBroadcast (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- `void chCondBroadcastl (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- `msg_t chCondWait (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitS (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeout (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeoutS (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.

9.6 chcond.h File Reference

9.6.1 Detailed Description

Condition Variables macros and structures.

Data Structures

- struct `CondVar`
CondVar structure.

Functions

- `void chCondInit (CondVar *cp)`
Initializes a `CondVar` structure.
- `void chCondSignal (CondVar *cp)`
Signals one thread that is waiting on the condition variable.
- `void chCondSignall (CondVar *cp)`
Signals one thread that is waiting on the condition variable.
- `void chCondBroadcast (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- `void chCondBroadcastl (CondVar *cp)`
Signals all threads that are waiting on the condition variable.
- `msg_t chCondWait (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitS (CondVar *cp)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeout (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.
- `msg_t chCondWaitTimeoutS (CondVar *cp, systime_t time)`
Waits on the condition variable releasing the mutex lock.

Defines

- `#define _CONDVAR_DATA(name) {_THREADSQUEUE_DATA(name.c_queue)}`
Data part of a static condition variable initializer.
- `#define CONDVAR_DECL(name) CondVar name = _CONDVAR_DATA(name)`
Static condition variable initializer.

Typedefs

- `typedef struct CondVar CondVar`
CondVar structure.

9.7 chconf.h File Reference

9.7.1 Detailed Description

Configuration file template. A copy of this file must be placed in each project directory, it contains the application specific kernel settings.

Defines

Kernel parameters and options

- `#define CH_FREQUENCY 1000`
System tick frequency.
- `#define CH_TIME_QUANTUM 20`
Round robin interval.
- `#define CH_MEMCORE_SIZE 0`
Managed RAM size.
- `#define CH_NO_IDLE_THREAD FALSE`
Idle thread automatic spawn suppression.

Performance options

- `#define CH_OPTIMIZE_SPEED TRUE`
OS optimization.

Subsystem options

- `#define CH_USE_REGISTRY TRUE`
Threads registry APIs.
- `#define CH_USE_WAITEXIT TRUE`
Threads synchronization APIs.
- `#define CH_USE_SEMAPHORES TRUE`
Semaphores APIs.
- `#define CH_USE_SEMAPHORES_PRIORITY FALSE`
Semaphores queuing mode.
- `#define CH_USE_SEMSW TRUE`
Atomic semaphore API.
- `#define CH_USE_MUTEXES TRUE`
Mutexes APIs.
- `#define CH_USE_CONDVAR TRUE`
Conditional Variables APIs.
- `#define CH_USE_CONDVAR_TIMEOUT TRUE`
Conditional Variables APIs with timeout.

- #define **CH_USE_EVENTS** TRUE
Events Flags APIs.
- #define **CH_USE_EVENTS_TIMEOUT** TRUE
Events Flags APIs with timeout.
- #define **CH_USE_MESSAGES** TRUE
Synchronous Messages APIs.
- #define **CH_USE_MESSAGES_PRIORITY** FALSE
Synchronous Messages queuing mode.
- #define **CH_USE_MAILBOXES** TRUE
Mailboxes APIs.
- #define **CH_USE_QUEUES** TRUE
I/O Queues APIs.
- #define **CH_USE_MEMCORE** TRUE
Core Memory Manager APIs.
- #define **CH_USE_HEAP** TRUE
Heap Allocator APIs.
- #define **CH_USE_MALLOC_HEAP** FALSE
C-runtime allocator.
- #define **CH_USE_MEMPOOLS** TRUE
Memory Pools Allocator APIs.
- #define **CH_USE_DYNAMIC** TRUE
Dynamic Threads APIs.

Debug options

- #define **CH_DBG_SYSTEM_STATE_CHECK** FALSE
Debug option, system state check.
- #define **CH_DBG_ENABLE_CHECKS** FALSE
Debug option, parameters checks.
- #define **CH_DBG_ENABLE_ASSERTS** FALSE
Debug option, consistency checks.
- #define **CH_DBG_ENABLE_TRACE** FALSE
Debug option, trace buffer.
- #define **CH_DBG_ENABLE_STACK_CHECK** FALSE
Debug option, stack checks.
- #define **CH_DBG_FILL_THREADS** FALSE
Debug option, stacks initialization.
- #define **CH_DBG_THREADS_PROFILING** TRUE
Debug option, threads profiling.

Kernel hooks

- #define **THREAD_EXT_FIELDS**
Threads descriptor structure extension.
- #define **THREAD_EXT_INIT_HOOK(tp)**
Threads initialization hook.
- #define **THREAD_EXT_EXIT_HOOK(tp)**
Threads finalization hook.
- #define **THREAD_CONTEXT_SWITCH_HOOK(ntp, otp)**
Context switch hook.
- #define **IDLE_LOOP_HOOK()**
Idle Loop hook.
- #define **SYSTEM_TICK_EVENT_HOOK()**
System tick event hook.
- #define **SYSTEM_HALT_HOOK()**
System halt hook.

9.8 chcore.c File Reference

9.8.1 Detailed Description

ARM7/9 architecture port code. #include "ch.h"

Functions

- void [port_halt](#) (void)

9.9 chcore.h File Reference

9.9.1 Detailed Description

ARM7/9 architecture port macros and structures. #include "armparams.h"

Data Structures

- struct [extctx](#)
Interrupt saved context.
- struct [intctx](#)
System saved context.
- struct [context](#)
Platform dependent part of the [Thread](#) structure.

Functions

- void [port_halt](#) (void)

Defines

- #define [ARM_CORE_ARM7TDMI](#) 7
- #define [ARM_CORE_ARM9](#) 9
- #define [ARM_ENABLE_WFI_IDLE](#) FALSE
If enabled allows the idle thread to enter a low power mode.
- #define [CH_ARCHITECTURE_ARM](#)
Macro defining a generic ARM architecture.
- #define [CH_ARCHITECTURE_ARMx](#)
Macro defining the specific ARM architecture.
- #define [CH_ARCHITECTURE_NAME](#) "ARMx"
Name of the implemented architecture.
- #define [CH_CORE_VARIANT_NAME](#) "ARMxy"
Name of the architecture variant (optional).
- #define [CH_PORT_INFO](#) "ARM|THUMB|Interworking"
Port-specific information string.
- #define [CH_PORT_INFO](#) "Pure ARM mode"
Port-specific information string.
- #define [CH_COMPILER_NAME](#) "GCC" __VERSION__
Name of the compiler supported by this port.

- `#define SETUP_CONTEXT(workspace, wsize, pf, arg)`
Platform dependent part of the `chThdCreateI()` API.
- `#define PORT_IDLE_THREAD_STACK_SIZE 4`
Stack size for the system idle thread.
- `#define PORT_INT_REQUIRED_STACK 0x10`
Per-thread stack overhead for interrupts servicing.
- `#define STACK_ALIGN(n) (((n) - 1) | (sizeof(stkalign_t) - 1)) + 1`
Enforces a correct alignment for a stack area size value.
- `#define THD_WA_SIZE(n)`
Computes the thread working area global size.
- `#define WORKING_AREA(s, n) stkalign_t s[THD_WA_SIZE(n) / sizeof(stkalign_t)]`
Static working area allocation.
- `#define PORT_IRQ_PROLOGUE()`
IRQ prologue code.
- `#define PORT_IRQ_EPILOGUE()`
IRQ epilogue code.
- `#define PORT_IRQ_HANDLER(id) __attribute__((naked)) void id(void)`
IRQ handler function declaration.
- `#define PORT_FAST_IRQ_HANDLER(id) __attribute__((interrupt("FIQ"))) void id(void)`
Fast IRQ handler function declaration.
- `#define port_init()`
Port-related initialization code.
- `#define port_lock() asm volatile ("msr CPSR_c, #0x9F" :: : "memory")`
Kernel-lock action.
- `#define port_unlock() asm volatile ("msr CPSR_c, #0x1F" :: : "memory")`
Kernel-unlock action.
- `#define port_lock_from_isr()`
Kernel-lock action from an interrupt handler.
- `#define port_unlock_from_isr()`
Kernel-unlock action from an interrupt handler.
- `#define port_disable()`
Disables all the interrupt sources.
- `#define port_suspend() asm volatile ("msr CPSR_c, #0x9F" :: : "memory")`
Disables the interrupt sources below kernel-level priority.
- `#define port_enable() asm volatile ("msr CPSR_c, #0x1F" :: : "memory")`
Enables all the interrupt sources.
- `#define port_switch(ntp, otp)`
Performs a context switch between two threads.

Typedefs

- `typedef uint32_t stkalign_t`
32 bits stack and memory alignment enforcement.
- `typedef void * regarm_t`
Generic ARM register.

9.10 chcoreasm.s File Reference

9.10.1 Detailed Description

ARM7/9 architecture port low level code. `#include "chconf.h"`

9.11 chdebug.c File Reference

9.11.1 Detailed Description

ChibiOS/RT Debug code. #include "ch.h"

Functions

- void `dbg_check_disable` (void)
Guard code for chSysDisable () .
- void `dbg_check_suspend` (void)
Guard code for chSysSuspend () .
- void `dbg_check_enable` (void)
Guard code for chSysEnable () .
- void `dbg_check_lock` (void)
Guard code for chSysLock () .
- void `dbg_check_unlock` (void)
Guard code for chSysUnlock () .
- void `dbg_check_lock_from_isr` (void)
Guard code for chSysLockFromIsr () .
- void `dbg_check_unlock_from_isr` (void)
Guard code for chSysUnlockFromIsr () .
- void `dbg_check_enter_isr` (void)
Guard code for CH_IRQ_PROLOGUE () .
- void `dbg_check_leave_isr` (void)
Guard code for CH_IRQ_EPILOGUE () .
- void `chDbgCheckClassI` (void)
I-class functions context check.
- void `chDbgCheckClassS` (void)
S-class functions context check.
- void `_trace_init` (void)
Trace circular buffer subsystem initialization.
- void `dbg_trace` (Thread *otp)
Inserts in the circular debug trace buffer a context switch record.
- void `chDbgPanic` (char *msg)
Prints a panic message on the console and then halts the system.

Variables

- `cnt_t dbg_isr_cnt`
ISR nesting level.
- `cnt_t dbg_lock_cnt`
Lock nesting level.
- `ch_trace_buffer_t dbg_trace_buffer`
Public trace buffer.
- char * `dbg_panic_msg`
Pointer to the panic message.

9.12 chdebug.h File Reference

9.12.1 Detailed Description

Debug macros and structures.

Data Structures

- struct `ch_swc_event_t`
Trace buffer record.
- struct `ch_trace_buffer_t`
Trace buffer header.

Functions

- void `_trace_init` (void)
Trace circular buffer subsystem initialization.
- void `dbg_trace` (`Thread` *otp)
Inserts in the circular debug trace buffer a context switch record.

Variables

- char * `dbg_panic_msg`
Pointer to the panic message.

Defines

Debug related settings

- `#define CH_TRACE_BUFFER_SIZE 64`
Trace buffer entries.
- `#define CH_STACK_FILL_VALUE 0x55`
Fill value for thread stack area in debug mode.
- `#define CH_THREAD_FILL_VALUE 0xFF`
Fill value for thread area in debug mode.

Macro Functions

- `#define chDbgCheck(c, func)`
Function parameter check.
- `#define chDbgAssert(c, m, r)`
Condition assertion.

9.13 chdynamic.c File Reference

9.13.1 Detailed Description

Dynamic threads code. `#include "ch.h"`

Functions

- `Thread * chThdAddRef (Thread *tp)`
Adds a reference to a thread object.
- `void chThdRelease (Thread *tp)`
Releases a reference to a thread object.
- `Thread * chThdCreateFromHeap (MemoryHeap *heapp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread allocating the memory from the heap.
- `Thread * chThdCreateFromMemoryPool (MemoryPool *mp, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread allocating the memory from the specified memory pool.

9.14 chdynamic.h File Reference

9.14.1 Detailed Description

Dynamic threads macros and structures.

Functions

- `Thread * chThdAddRef (Thread *tp)`
Adds a reference to a thread object.
- `void chThdRelease (Thread *tp)`
Releases a reference to a thread object.
- `Thread * chThdCreateFromHeap (MemoryHeap *heapp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread allocating the memory from the heap.
- `Thread * chThdCreateFromMemoryPool (MemoryPool *mp, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread allocating the memory from the specified memory pool.

9.15 chevents.c File Reference

9.15.1 Detailed Description

Events code. #include "ch.h"

Functions

- `void chEvtRegisterMask (EventSource *esp, EventListener *elp, eventmask_t mask)`
Registers an Event Listener on an Event Source.
- `void chEvtUnregister (EventSource *esp, EventListener *elp)`
Unregisters an Event Listener from its Event Source.
- `eventmask_t chEvtClearFlags (eventmask_t mask)`
Clears the pending events specified in the mask.
- `eventmask_t chEvtAddFlags (eventmask_t mask)`
*Adds (OR) a set of event flags on the current thread, this is **much** faster than using `chEvtBroadcast ()` or `chEvtSignal ()`.*
- `void chEvtSignalFlags (Thread *tp, eventmask_t mask)`
Adds (OR) a set of event flags on the specified Thread.
- `void chEvtSignalFlagsl (Thread *tp, eventmask_t mask)`
Adds (OR) a set of event flags on the specified Thread.

- void [chEvtBroadcastFlags](#) ([EventSource](#) *esp, [eventmask_t](#) mask)
Signals all the Event Listeners registered on the specified Event Source.
- void [chEvtBroadcastFlagsI](#) ([EventSource](#) *esp, [eventmask_t](#) mask)
Signals all the Event Listeners registered on the specified Event Source.
- void [chEvtDispatch](#) ([const evhandler_t](#) *handlers, [eventmask_t](#) mask)
Invokes the event handlers associated to an event flags mask.
- [eventmask_t](#) [chEvtWaitOne](#) ([eventmask_t](#) mask)
Waits for exactly one of the specified events.
- [eventmask_t](#) [chEvtWaitAny](#) ([eventmask_t](#) mask)
Waits for any of the specified events.
- [eventmask_t](#) [chEvtWaitAll](#) ([eventmask_t](#) mask)
Waits for all the specified events.
- [eventmask_t](#) [chEvtWaitOneTimeout](#) ([eventmask_t](#) mask, [systime_t](#) time)
Waits for exactly one of the specified events.
- [eventmask_t](#) [chEvtWaitAnyTimeout](#) ([eventmask_t](#) mask, [systime_t](#) time)
Waits for any of the specified events.
- [eventmask_t](#) [chEvtWaitAllTimeout](#) ([eventmask_t](#) mask, [systime_t](#) time)
Waits for all the specified events.

9.16 chevents.h File Reference

9.16.1 Detailed Description

Events macros and structures.

Data Structures

- struct [EventListener](#)
Event Listener structure.
- struct [EventSource](#)
Event Source structure.

Functions

- void [chEvtRegisterMask](#) ([EventSource](#) *esp, [EventListener](#) *elp, [eventmask_t](#) mask)
Registers an Event Listener on an Event Source.
- void [chEvtUnregister](#) ([EventSource](#) *esp, [EventListener](#) *elp)
Unregisters an Event Listener from its Event Source.
- [eventmask_t](#) [chEvtClearFlags](#) ([eventmask_t](#) mask)
Clears the pending events specified in the mask.
- [eventmask_t](#) [chEvtAddFlags](#) ([eventmask_t](#) mask)
*Adds (OR) a set of event flags on the current thread, this is **much** faster than using [chEvtBroadcast\(\)](#) or [chEvtSignal\(\)](#).*
- void [chEvtSignalFlags](#) ([Thread](#) *tp, [eventmask_t](#) mask)
Adds (OR) a set of event flags on the specified [Thread](#).
- void [chEvtSignalFlagsI](#) ([Thread](#) *tp, [eventmask_t](#) mask)
Adds (OR) a set of event flags on the specified [Thread](#).
- void [chEvtBroadcastFlags](#) ([EventSource](#) *esp, [eventmask_t](#) mask)
Signals all the Event Listeners registered on the specified Event Source.
- void [chEvtBroadcastFlagsI](#) ([EventSource](#) *esp, [eventmask_t](#) mask)

- **Signals all the Event Listeners registered on the specified Event Source.**
- void **chEvtDispatch** (const **evhandler_t** *handlers, **eventmask_t** mask)
 - Invokes the event handlers associated to an event flags mask.*
- **eventmask_t chEvtWaitOneTimeout** (**eventmask_t** mask, **systime_t** time)
 - Waits for exactly one of the specified events.*
- **eventmask_t chEvtWaitAnyTimeout** (**eventmask_t** mask, **systime_t** time)
 - Waits for any of the specified events.*
- **eventmask_t chEvtWaitAllTimeout** (**eventmask_t** mask, **systime_t** time)
 - Waits for all the specified events.*

Defines

- **#define _EVENTSOURCE_DATA**(name) {(void *)(&name)}
 - Data part of a static event source initializer.*
- **#define EVENTSOURCE_DECL**(name) **EventSource** name = _EVENTSOURCE_DATA(name)
 - Static event source initializer.*
- **#define ALL_EVENTS** ((**eventmask_t**)-1)
 - All events allowed mask.*
- **#define EVENT_MASK**(eid) ((**eventmask_t**)(1 << (eid)))
 - Returns an event mask from an event identifier.*

Macro Functions

- **#define chEvtRegister**(esp, elp, eid) **chEvtRegisterMask**(esp, elp, **EVENT_MASK**(eid))
 - Registers an Event Listener on an Event Source.*
- **#define chEvtInit**(esp) ((esp)->es_next = (**EventListener** *)(void *)(esp))
 - Initializes an Event Source.*
- **#define chEvtIsListeningI**(esp) ((void *)(esp) != (void *)(esp)->es_next)
 - Verifies if there is at least one **EventListener** registered.*
- **#define chEvtBroadcast**(esp) **chEvtBroadcastFlags**(esp, 0)
 - Signals all the Event Listeners registered on the specified Event Source.*
- **#define chEvtBroadcastI**(esp) **chEvtBroadcastFlagsI**(esp, 0)
 - Signals all the Event Listeners registered on the specified Event Source.*

Typedefs

- **typedef struct EventSource EventSource**
 - Event Source structure.*
- **typedef void(* evhandler_t)(eventid_t)**
 - Event Handler callback function.*

9.17 chfiles.h File Reference

9.17.1 Detailed Description

Data files. This header defines abstract interfaces useful to access generic data files in a standardized way.

Data Structures

- **struct BaseFileStreamVMT**
 - BaseFileStream virtual methods table.*
- **struct BaseFileStream**
 - Base file stream class.*

Defines

- `#define FILE_OK 0`
No error return code.
- `#define FILE_ERROR 0xFFFFFFFFFUL`
Error code from the file stream methods.
- `#define _base_file_stream_methods`
BaseFileStream specific methods.
- `#define _base_file_stream_data _base_sequential_stream_data`
BaseFileStream specific data.

Macro Functions (BaseFileStream)

- `#define chFileStreamClose(ip) ((ip)->vmt->close(ip))`
Base file Stream close.
- `#define chFileStreamGetError(ip) ((ip)->vmt->geterror(ip))`
Returns an implementation dependent error code.
- `#define chFileStreamGetSize(ip) ((ip)->vmt->getposition(ip))`
Returns the current file size.
- `#define chFileStreamGetPosition(ip) ((ip)->vmt->getposition(ip))`
Returns the current file pointer position.
- `#define chFileStreamSeek(ip, offset) ((ip)->vmt->lseek(ip, offset))`
Moves the file current pointer to an absolute position.

Typedefs

- `typedef uint32_t fileoffset_t`
File offset type.

9.18 chheap.c File Reference

9.18.1 Detailed Description

Heaps code. `#include "ch.h"`

Functions

- `void _heap_init (void)`
Initializes the default heap.
- `void chHeapInit (MemoryHeap *heapp, void *buf, size_t size)`
Initializes a memory heap from a static memory area.
- `void * chHeapAlloc (MemoryHeap *heapp, size_t size)`
Allocates a block of memory from the heap by using the first-fit algorithm.
- `void chHeapFree (void *p)`
Frees a previously allocated memory block.
- `size_t chHeapStatus (MemoryHeap *heapp, size_t *sizep)`
Reports the heap status.

9.19 chheap.h File Reference

9.19.1 Detailed Description

Heaps macros and structures.

Data Structures

- union `heap_header`
Memory heap block header.
- struct `memory_heap`
Structure describing a memory heap.

Functions

- void `_heap_init` (void)
Initializes the default heap.
- void `chHeapInit` (`MemoryHeap` *heapp, void *buf, `size_t` size)
Initializes a memory heap from a static memory area.
- void * `chHeapAlloc` (`MemoryHeap` *heapp, `size_t` size)
Allocates a block of memory from the heap by using the first-fit algorithm.
- void `chHeapFree` (void *p)
Frees a previously allocated memory block.
- `size_t` `chHeapStatus` (`MemoryHeap` *heapp, `size_t` *sizep)
Reports the heap status.

9.20 chinline.h File Reference

9.20.1 Detailed Description

Kernel inlined functions. In this file there are a set of inlined functions if the `CH_OPTIMIZE_SPEED` is enabled.

9.21 chioch.h File Reference

9.21.1 Detailed Description

I/O channels. This header defines abstract interfaces useful to access generic I/O resources in a standardized way.

Data Structures

- struct `BaseChannelVMT`
`BaseChannel` virtual methods table.
- struct `BaseChannel`
Base channel class.
- struct `BaseAsynchronousChannelVMT`
`BaseAsynchronousChannel` virtual methods table.
- struct `BaseAsynchronousChannel`
Base asynchronous channel class.

Defines

- `#define _base_channel_methods`
`BaseChannel` specific methods.
- `#define _base_channel_data _base_sequential_stream_data`
`BaseChannel` specific data.

- `#define _base_asynchronous_channel_methods`
`BaseAsynchronousChannel` specific methods.
- `#define _base_asynchronous_channel_data`
`BaseAsynchronousChannel` specific data.
- `#define _ch_get_and_clear_flags_impl(ip)`
Default implementation of the `getflags` virtual method.

Macro Functions (BaseChannel)

- `#define chIOPutWouldBlock(ip) ((ip)->vmt->putwouldblock(ip))`
Channel output check.
- `#define chIOGetWouldBlock(ip) ((ip)->vmt->getwouldblock(ip))`
Channel input check.
- `#define chIOPut(ip, b) ((ip)->vmt->put(ip, b, TIME_INFINITE))`
Channel blocking byte write.
- `#define chIOPutTimeout(ip, b, time) ((ip)->vmt->put(ip, b, time))`
Channel blocking byte write with timeout.
- `#define chIOGet(ip) ((ip)->vmt->get(ip, TIME_INFINITE))`
Channel blocking byte read.
- `#define chIOGetTimeout(ip, time) ((ip)->vmt->get(ip, time))`
Channel blocking byte read with timeout.
- `#define chIOWriteTimeout(ip, bp, n, time) ((ip)->vmt->writet(ip, bp, n, time))`
Channel blocking write with timeout.
- `#define chIORReadTimeout(ip, bp, n, time) ((ip)->vmt->readt(ip, bp, n, time))`
Channel blocking read with timeout.

I/O status flags

- `#define IO_NO_ERROR 0`
No pending conditions.
- `#define IO_CONNECTED 1`
Connection happened.
- `#define IO_DISCONNECTED 2`
Disconnection happened.
- `#define IO_INPUT_AVAILABLE 4`
Data available in the input queue.
- `#define IO_OUTPUT_EMPTY 8`
Output queue empty.
- `#define IO_TRANSMISSION_END 16`
Transmission end.

Macro Functions (BaseAsynchronousChannel)

- `#define chIOGetEventSource(ip) (&((ip)->event))`
Returns the I/O condition event source.
- `#define chIOAddFlags1(ip, mask)`
Adds status flags to the channel's mask.
- `#define chIOGetAndClearFlags(ip) ((ip)->vmt->getflags(ip))`
Returns and clears the status flags associated to the channel.

Typedefs

- `typedef uint_fast16_t ioflags_t`
Type of an I/O condition flags mask.

9.22 chlists.c File Reference

9.22.1 Detailed Description

`Thread` queues/lists code. #include "ch.h"

Functions

- void `prio_insert` (`Thread` *tp, `ThreadsQueue` *tqp)
Inserts a thread into a priority ordered queue.
- void `queue_insert` (`Thread` *tp, `ThreadsQueue` *tqp)
Inserts a Thread into a queue.
- `Thread` * `fifo_remove` (`ThreadsQueue` *tqp)
Removes the first-out Thread from a queue and returns it.
- `Thread` * `lifo_remove` (`ThreadsQueue` *tqp)
Removes the last-out Thread from a queue and returns it.
- `Thread` * `dequeue` (`Thread` *tp)
Removes a Thread from a queue and returns it.
- void `list_insert` (`Thread` *tp, `ThreadsList` *tlp)
Pushes a Thread on top of a stack list.
- `Thread` * `list_remove` (`ThreadsList` *tlp)
Pops a Thread from the top of a stack list and returns it.

9.23 chlists.h File Reference

9.23.1 Detailed Description

`Thread` queues/lists macros and structures.

Note

All the macros present in this module, while public, are not an OS API and should not be directly used in the user applications code.

Data Structures

- struct `ThreadsQueue`
Generic threads bidirectional linked list header and element.
- struct `ThreadsList`
Generic threads single link list, it works like a stack.

Functions

- void `prio_insert` (`Thread` *tp, `ThreadsQueue` *tqp)
Inserts a thread into a priority ordered queue.
- void `queue_insert` (`Thread` *tp, `ThreadsQueue` *tqp)
Inserts a Thread into a queue.
- `Thread` * `fifo_remove` (`ThreadsQueue` *tqp)
Removes the first-out Thread from a queue and returns it.
- `Thread` * `lifo_remove` (`ThreadsQueue` *tqp)

- `Thread * dequeue (Thread *tp)`
Removes the last-out `Thread` from a queue and returns it.
- `void list_insert (Thread *tp, ThreadsList *tlp)`
Pushes a `Thread` on top of a stack list.
- `Thread * list_remove (ThreadsList *tlp)`
Pops a `Thread` from the top of a stack list and returns it.

Defines

- `#define queue_init(tqp) ((tqp)->p_next = (tqp)->p_prev = (Thread *)(tqp));`
Threads queue initialization.
- `#define list_init(tlp) ((tlp)->p_next = (Thread *)(tlp))`
Threads list initialization.
- `#define isempty(p) ((p)->p_next == (Thread *)(p))`
Evaluates to TRUE if the specified threads queue or list is empty.
- `#define notempty(p) ((p)->p_next != (Thread *)(p))`
Evaluates to TRUE if the specified threads queue or list is not empty.
- `#define _THREADSQUEUE_DATA(name) {(<code>Thread *</code>) &name, (<code>Thread *</code>) &name}`
Data part of a static threads queue initializer.
- `#define THREADSQUEUE_DECL(name) ThreadsQueue name = _THREADSQUEUE_DATA(name)`
Static threads queue initializer.

9.24 chmboxes.c File Reference

9.24.1 Detailed Description

Mailboxes code. `#include "ch.h"`

Functions

- `void chMBInit (Mailbox *mbp, msg_t *buf, cnt_t n)`
Initializes a `Mailbox` object.
- `void chMBReset (Mailbox *mbp)`
Resets a `Mailbox` object.
- `msg_t chMBPost (Mailbox *mbp, msg_t msg, systime_t time)`
Posts a message into a mailbox.
- `msg_t chMBPostS (Mailbox *mbp, msg_t msg, systime_t time)`
Posts a message into a mailbox.
- `msg_t chMBPostI (Mailbox *mbp, msg_t msg)`
Posts a message into a mailbox.
- `msg_t chMBPostAhead (Mailbox *mbp, msg_t msg, systime_t time)`
Posts an high priority message into a mailbox.
- `msg_t chMBPostAheadS (Mailbox *mbp, msg_t msg, systime_t time)`
Posts an high priority message into a mailbox.
- `msg_t chMBPostAheadI (Mailbox *mbp, msg_t msg)`
Posts an high priority message into a mailbox.
- `msg_t chMBFetch (Mailbox *mbp, msg_t *msgp, systime_t time)`
Retrieves a message from a mailbox.
- `msg_t chMBFetchS (Mailbox *mbp, msg_t *msgp, systime_t time)`

- `msg_t chMBFetchl (Mailbox *mbp, msg_t *msgp)`
Retrieves a message from a mailbox.

9.25 chmboxes.h File Reference

9.25.1 Detailed Description

Mailboxes macros and structures.

Data Structures

- struct `Mailbox`
Structure representing a mailbox object.

Functions

- `void chMBInit (Mailbox *mbp, msg_t *buf, cnt_t n)`
Initializes a `Mailbox` object.
- `void chMBReset (Mailbox *mbp)`
Resets a `Mailbox` object.
- `msg_t chMBPost (Mailbox *mbp, msg_t msg, systime_t time)`
Posts a message into a mailbox.
- `msg_t chMBPostS (Mailbox *mbp, msg_t msg, systime_t time)`
Posts a message into a mailbox.
- `msg_t chMBPostl (Mailbox *mbp, msg_t msg)`
Posts a message into a mailbox.
- `msg_t chMBPostAhead (Mailbox *mbp, msg_t msg, systime_t time)`
Posts an high priority message into a mailbox.
- `msg_t chMBPostAheadS (Mailbox *mbp, msg_t msg, systime_t time)`
Posts an high priority message into a mailbox.
- `msg_t chMBPostAheadl (Mailbox *mbp, msg_t msg)`
Posts an high priority message into a mailbox.
- `msg_t chMBFetch (Mailbox *mbp, msg_t *msgp, systime_t time)`
Retrieves a message from a mailbox.
- `msg_t chMBFetchS (Mailbox *mbp, msg_t *msgp, systime_t time)`
Retrieves a message from a mailbox.
- `msg_t chMBFetchl (Mailbox *mbp, msg_t *msgp)`
Retrieves a message from a mailbox.

Defines

- `#define _MAILBOX_DATA(name, buffer, size)`
Data part of a static mailbox initializer.
- `#define MAILBOX_DECL(name, buffer, size) Mailbox name = _MAILBOX_DATA(name, buffer, size)`
Static mailbox initializer.

Macro Functions

- `#define chMBSizeI(mbp) ((mbp)->mb_top - (mbp)->mb_buffer)`
Returns the mailbox buffer size.
- `#define chMBGetFreeCountI(mbp) chSemGetCounterI(&(mbp)->mb_emptysem)`
Returns the number of free message slots into a mailbox.
- `#define chMBGetUsedCountI(mbp) chSemGetCounterI(&(mbp)->mb_fullsem)`
Returns the number of used message slots into a mailbox.
- `#define chMBPeekI(mbp) (*(mbp)->mb_rdptr)`
Returns the next message in the queue without removing it.

9.26 chmemcore.c File Reference

9.26.1 Detailed Description

Core memory manager code. `#include "ch.h"`

Functions

- `void _core_init (void)`
Low level memory manager initialization.
- `void * chCoreAlloc (size_t size)`
Allocates a memory block.
- `void * chCoreAllocI (size_t size)`
Allocates a memory block.
- `size_t chCoreStatus (void)`
Core memory status.

9.27 chmemcore.h File Reference

9.27.1 Detailed Description

Core memory manager macros and structures.

Functions

- `void _core_init (void)`
Low level memory manager initialization.
- `void * chCoreAlloc (size_t size)`
Allocates a memory block.
- `void * chCoreAllocI (size_t size)`
Allocates a memory block.
- `size_t chCoreStatus (void)`
Core memory status.

Defines

Alignment support macros

- #define `MEM_ALIGN_SIZE` sizeof(`stkalign_t`)
Alignment size constant.
- #define `MEM_ALIGN_MASK` (`MEM_ALIGN_SIZE` - 1)
Alignment mask constant.
- #define `MEM_ALIGN_PREV`(`p`) ((`size_t`)(`p`) & ~`MEM_ALIGN_MASK`)
Alignment helper macro.
- #define `MEM_ALIGN_NEXT`(`p`) `MEM_ALIGN_PREV`((`size_t`)(`p`) + `MEM_ALIGN_MASK`)
Alignment helper macro.
- #define `MEM_IS_ALIGNED`(`p`) (((`size_t`)(`p`) & `MEM_ALIGN_MASK`) == 0)
Returns whatever a pointer or memory size is aligned to the type `align_t`.

Typedefs

- `typedef void *(*memgetfunc_t)(size_t size)`
Memory get function.

9.28 chmempools.c File Reference

9.28.1 Detailed Description

Memory Pools code. `#include "ch.h"`

Functions

- `void chPoolInit (MemoryPool *mp, size_t size, memgetfunc_t provider)`
Initializes an empty memory pool.
- `void * chPoolAlloc (MemoryPool *mp)`
Allocates an object from a memory pool.
- `void * chPoolAlloc (MemoryPool *mp)`
Allocates an object from a memory pool.
- `void chPoolFree (MemoryPool *mp, void *objp)`
Releases (or adds) an object into (to) a memory pool.
- `void chPoolFree (MemoryPool *mp, void *objp)`
Releases (or adds) an object into (to) a memory pool.

9.29 chmempools.h File Reference

9.29.1 Detailed Description

Memory Pools macros and structures.

Data Structures

- struct `pool_header`
Memory pool free object header.
- struct `MemoryPool`
Memory pool descriptor.

Functions

- void **chPoolInit** (MemoryPool *mp, size_t size, memgetfunc_t provider)
Initializes an empty memory pool.
- void * **chPoolAlloc** (MemoryPool *mp)
Allocates an object from a memory pool.
- void * **chPoolAlloc** (MemoryPool *mp)
Allocates an object from a memory pool.
- void **chPoolFree** (MemoryPool *mp, void *objp)
Releases (or adds) an object into (to) a memory pool.
- void **chPoolFree** (MemoryPool *mp, void *objp)
Releases (or adds) an object into (to) a memory pool.

Defines

- #define **_MEMORYPOOL_DATA**(name, size, provider) {NULL, MEM_ALIGN_NEXT(size), provider}
Data part of a static memory pool initializer.
- #define **MEMORYPOOL_DECL**(name, size, provider) MemoryPool name = **_MEMORYPOOL_DATA**(name, size, provider)
Static memory pool initializer in hungry mode.

9.30 chmsg.c File Reference

9.30.1 Detailed Description

Messages code. #include "ch.h"

Functions

- msg_t **chMsgSend** (Thread *tp, msg_t msg)
Sends a message to the specified thread.
- Thread * **chMsgWait** (void)
Suspends the thread and waits for an incoming message.
- void **chMsgRelease** (Thread *tp, msg_t msg)
Releases a sender thread specifying a response message.

9.31 chmsg.h File Reference

9.31.1 Detailed Description

Messages macros and structures.

Functions

- msg_t **chMsgSend** (Thread *tp, msg_t msg)
Sends a message to the specified thread.
- Thread * **chMsgWait** (void)
Suspends the thread and waits for an incoming message.
- void **chMsgRelease** (Thread *tp, msg_t msg)
Releases a sender thread specifying a response message.

Defines

Macro Functions

- `#define chMsgIsPendingI(tp) ((tp)->p_msgqueue.p_next != (Thread *)&(tp)->p_msgqueue)`
Evaluates to TRUE if the thread has pending messages.
- `#define chMsgGet(tp) ((tp)->p_msg)`
Returns the message carried by the specified thread.
- `#define chMsgGetS(tp) ((tp)->p_msg)`
Returns the message carried by the specified thread.
- `#define chMsgReleaseS(tp, msg) chSchWakeupS(tp, msg)`
Releases the thread waiting on top of the messages queue.

9.32 chmtx.c File Reference

9.32.1 Detailed Description

Mutexes code. `#include "ch.h"`

Functions

- `void chMtxInit (Mutex *mp)`
Initializes a Mutex structure.
- `void chMtxLock (Mutex *mp)`
Locks the specified mutex.
- `void chMtxLockS (Mutex *mp)`
Locks the specified mutex.
- `bool_t chMtxTryLock (Mutex *mp)`
Tries to lock a mutex.
- `bool_t chMtxTryLockS (Mutex *mp)`
Tries to lock a mutex.
- `Mutex * chMtxUnlock (void)`
Unlocks the next owned mutex in reverse lock order.
- `Mutex * chMtxUnlockS (void)`
Unlocks the next owned mutex in reverse lock order.
- `void chMtxUnlockAll (void)`
Unlocks all the mutexes owned by the invoking thread.

9.33 chmtx.h File Reference

9.33.1 Detailed Description

Mutexes macros and structures.

Data Structures

- `struct Mutex`
Mutex structure.

Functions

- void **chMtxInit** (**Mutex** *mp)
Initializes a Mutex structure.
- void **chMtxLock** (**Mutex** *mp)
Locks the specified mutex.
- void **chMtxLockS** (**Mutex** *mp)
Locks the specified mutex.
- **bool_t** **chMtxTryLock** (**Mutex** *mp)
Tries to lock a mutex.
- **bool_t** **chMtxTryLockS** (**Mutex** *mp)
Tries to lock a mutex.
- **Mutex** * **chMtxUnlock** (void)
Unlocks the next owned mutex in reverse lock order.
- **Mutex** * **chMtxUnlockS** (void)
Unlocks the next owned mutex in reverse lock order.
- void **chMtxUnlockAll** (void)
Unlocks all the mutexes owned by the invoking thread.

Defines

- #define **_MUTEX_DATA**(name) {_THREADSQUEUE_DATA(name.m_queue), NULL, NULL}
Data part of a static mutex initializer.
- #define **MUTEX_DECL**(name) **Mutex** name = **_MUTEX_DATA**(name)
Static mutex initializer.

Macro Functions

- #define **chMtxQueueNotEmptyS**(mp) notempty(&(mp)->m_queue)
Returns TRUE if the mutex queue contains at least a waiting thread.

TypeDefs

- typedef struct **Mutex** **Mutex**
Mutex structure.

9.34 chqueues.c File Reference

9.34.1 Detailed Description

I/O Queues code. #include "ch.h"

Functions

- void **chIQInit** (**InputQueue** *iqp, **uint8_t** *bp, **size_t** size, **qnotify_t** infy)
Initializes an input queue.
- void **chIQResetI** (**InputQueue** *iqp)
Resets an input queue.
- **msg_t** **chIQPutI** (**InputQueue** *iqp, **uint8_t** b)
Input queue write.

- `msg_t chIQGetTimeout (InputQueue *iqp, systime_t time)`
Input queue read with timeout.
- `size_t chIQReadTimeout (InputQueue *iqp, uint8_t *bp, size_t n, systime_t time)`
Input queue read with timeout.
- `void chOQInit (OutputQueue *oqp, uint8_t *bp, size_t size, qnotify_t onfy)`
Initializes an output queue.
- `void chOQResetI (OutputQueue *oqp)`
Resets an output queue.
- `msg_t chOQPutTimeout (OutputQueue *oqp, uint8_t b, systime_t time)`
Output queue write with timeout.
- `msg_t chOQGetI (OutputQueue *oqp)`
Output queue read.
- `size_t chOQWriteTimeout (OutputQueue *oqp, const uint8_t *bp, size_t n, systime_t time)`
Output queue write with timeout.

9.35 chqueues.h File Reference

9.35.1 Detailed Description

I/O Queues macros and structures.

Data Structures

- struct `GenericQueue`
Generic I/O queue structure.

Functions

- `void chIQInit (InputQueue *iqp, uint8_t *bp, size_t size, qnotify_t onfy)`
Initializes an input queue.
- `void chIQResetI (InputQueue *iqp)`
Resets an input queue.
- `msg_t chIQPutI (InputQueue *iqp, uint8_t b)`
Input queue write.
- `msg_t chIQGetTimeout (InputQueue *iqp, systime_t time)`
Input queue read with timeout.
- `size_t chIQReadTimeout (InputQueue *iqp, uint8_t *bp, size_t n, systime_t time)`
Input queue read with timeout.
- `void chOQInit (OutputQueue *oqp, uint8_t *bp, size_t size, qnotify_t onfy)`
Initializes an output queue.
- `void chOQResetI (OutputQueue *oqp)`
Resets an output queue.
- `msg_t chOQPutTimeout (OutputQueue *oqp, uint8_t b, systime_t time)`
Output queue write with timeout.
- `msg_t chOQGetI (OutputQueue *oqp)`
Output queue read.
- `size_t chOQWriteTimeout (OutputQueue *oqp, const uint8_t *bp, size_t n, systime_t time)`
Output queue write with timeout.

Defines

- `#define _INPUTQUEUE_DATA(name, buffer, size, inotify)`
Data part of a static input queue initializer.
- `#define INPUTQUEUE_DECL(name, buffer, size, inotify) InputQueue name = _INPUTQUEUE_DATA(name, buffer, size, inotify)`
Static input queue initializer.
- `#define _OUTPUTQUEUE_DATA(name, buffer, size, onotify)`
Data part of a static output queue initializer.
- `#define OUTPUTQUEUE_DECL(name, buffer, size, onotify) OutputQueue name = _OUTPUTQUEUE_DATA(name, buffer, size, onotify)`
Static output queue initializer.

Queue functions returned status value

- `#define Q_OK RDY_OK`
Operation successful.
- `#define Q_TIMEOUT RDY_TIMEOUT`
Timeout condition.
- `#define Q_RESET RDY_RESET`
Queue has been reset.
- `#define Q_EMPTY -3`
Queue empty.
- `#define Q_FULL -4`
Queue full.,

Macro Functions

- `#define chQSizeI(qp) ((size_t)((qp)->q_top - (qp)->q_buffer))`
Returns the queue's buffer size.
- `#define chQSpaceI(qp) ((qp)->q_counter)`
Queue space.
- `#define chIQGetFullI(iqp) chQSpaceI(iqp)`
Returns the filled space into an input queue.
- `#define chIQGetEmptyI(iqp) (chQSizeI(iqp) - chQSpaceI(iqp))`
Returns the empty space into an input queue.
- `#define chIQIsEmptyI(iqp) ((bool_t)(chQSpaceI(iqp) <= 0))`
Evaluates to TRUE if the specified input queue is empty.
- `#define chIQIsFullI(iqp)`
Evaluates to TRUE if the specified input queue is full.
- `#define chIQGet(iqp) chIQGetTimeout(iqp, TIME_INFINITE)`
Input queue read.
- `#define chOQGetFullI(oqp) (chQSizeI(oqp) - chQSpaceI(oqp))`
Returns the filled space into an output queue.
- `#define chOQGetEmptyI(iqp) chQSpaceI(oqp)`
Returns the empty space into an output queue.
- `#define chOQIsEmptyI(oqp)`
Evaluates to TRUE if the specified output queue is empty.
- `#define chOQIsFullI(oqp) ((bool_t)(chQSpaceI(oqp) <= 0))`
Evaluates to TRUE if the specified output queue is full.
- `#define chOQPut(oqp, b) chOQPutTimeout(oqp, b, TIME_INFINITE)`
Output queue write.

Typedefs

- **typedef struct GenericQueue GenericQueue**
Type of a generic I/O queue structure.
- **typedef void(* qnotify_t)(GenericQueue *qp)**
Queue notification callback type.
- **typedef GenericQueue InputQueue**
Type of an input queue structure.
- **typedef GenericQueue OutputQueue**
Type of an output queue structure.

9.36 chregistry.c File Reference

9.36.1 Detailed Description

Threads registry code. #include "ch.h"

Functions

- **Thread * chRegFirstThread (void)**
Returns the first thread in the system.
- **Thread * chRegNextThread (Thread *tp)**
Returns the thread next to the specified one.

9.37 chregistry.h File Reference

9.37.1 Detailed Description

Threads registry macros and structures.

Functions

- **Thread * chRegFirstThread (void)**
Returns the first thread in the system.
- **Thread * chRegNextThread (Thread *tp)**
Returns the thread next to the specified one.

Defines

- **#define REG_REMOVE(tp)**
Removes a thread from the registry list.
- **#define REG_INSERT(tp)**
Adds a thread to the registry list.

Macro Functions

- **#define chRegSetThreadName(p) (currp->p_name = (p))**
Sets the current thread name.
- **#define chRegGetThreadName(tp) ((tp)->p_name)**
Returns the name of the specified thread.

9.38 chschd.c File Reference

9.38.1 Detailed Description

Scheduler code. #include "ch.h"

Functions

- void `_scheduler_init` (void)
Scheduler initialization.
- `Thread * chSchReadyI (Thread *tp)`
Inserts a thread in the Ready List.
- void `chSchGoSleepS (tstate_t newstate)`
Puts the current thread to sleep into the specified state.
- `msg_t chSchGoSleepTimeoutS (tstate_t newstate, systime_t time)`
Puts the current thread to sleep into the specified state with timeout specification.
- void `chSchWakeUpS (Thread *ntp, msg_t msg)`
Wakes up a thread.
- void `chSchRescheduleS (void)`
Performs a reschedule if a higher priority thread is runnable.
- `bool_t chSchIsPreemptionRequired (void)`
Evaluates if preemption is required.
- void `chSchDoReschedule (void)`
Switches to the first thread on the runnable queue.

Variables

- `ReadyList rlist`
Ready list header.

9.39 chschd.h File Reference

9.39.1 Detailed Description

Scheduler macros and structures.

Data Structures

- struct `ReadyList`
Ready list header.

Functions

- void `_scheduler_init` (void)
Scheduler initialization.
- `Thread * chSchReadyI (Thread *tp)`
Inserts a thread in the Ready List.
- void `chSchGoSleepS (tstate_t newstate)`
Puts the current thread to sleep into the specified state.

- **msg_t chSchGoSleepTimeoutS (tstate_t newstate, systime_t time)**
Puts the current thread to sleep into the specified state with timeout specification.
- **void chSchWakeupS (Thread *ntp, msg_t msg)**
Wakes up a thread.
- **void chSchRescheduleS (void)**
Performs a reschedule if a higher priority thread is runnable.
- **bool_t chSchIsPreemptionRequired (void)**
Evaluates if preemption is required.
- **void chSchDoReschedule (void)**
Switches to the first thread on the runnable queue.

Defines

- **#define firstprio(rlp) ((rlp)->p_next->p_prio)**
Returns the priority of the first thread on the given ready list.
- **#define currp rlist.r_current**
Current thread pointer access macro.
- **#define setcurrp(tp) (currp = (tp))**
Current thread pointer change macro.

Wakeup status codes

- **#define RDY_OK 0**
Normal wakeup message.
- **#define RDY_TIMEOUT -1**
Wakeup caused by a timeout condition.
- **#define RDY_RESET -2**
Wakeup caused by a reset condition.

Priority constants

- **#define NOPRIO 0**
Ready list header priority.
- **#define IDLEPRIO 1**
Idle thread priority.
- **#define LOWPRIO 2**
Lowest user priority.
- **#define NORMALPRIO 64**
Normal user priority.
- **#define HIGHPRIO 127**
Highest user priority.
- **#define ABSPRIO 255**
Greatest possible priority.

Special time constants

- **#define TIME_IMMEDIATE ((systime_t)0)**
Zero time specification for some functions with a timeout specification.
- **#define TIME_INFINITE ((systime_t)-1)**
Infinite time specification for all functions with a timeout specification.

Macro Functions

- **#define chSchIsRescRequired() (firstprio(&rlist.r_queue) > currp->p_prio)**
Determines if the current thread must reschedule.
- **#define chSchCanYieldS() (firstprio(&rlist.r_queue) >= currp->p_prio)**
Determines if yielding is possible.
- **#define chSchDoYieldS()**
Yields the time slot.
- **#define chSchPreemption()**
Inlineable preemption code.

9.40 chsem.c File Reference

9.40.1 Detailed Description

Semaphores code. #include "ch.h"

Functions

- void **chSemInit** (*Semaphore* *sp, *cnt_t* n)
Initializes a semaphore with the specified counter value.
- void **chSemReset** (*Semaphore* *sp, *cnt_t* n)
Performs a reset operation on the semaphore.
- void **chSemResetl** (*Semaphore* *sp, *cnt_t* n)
Performs a reset operation on the semaphore.
- *msg_t* **chSemWait** (*Semaphore* *sp)
Performs a wait operation on a semaphore.
- *msg_t* **chSemWaitS** (*Semaphore* *sp)
Performs a wait operation on a semaphore.
- *msg_t* **chSemWaitTimeout** (*Semaphore* *sp, *systime_t* time)
Performs a wait operation on a semaphore with timeout specification.
- *msg_t* **chSemWaitTimeoutS** (*Semaphore* *sp, *systime_t* time)
Performs a wait operation on a semaphore with timeout specification.
- void **chSemSignal** (*Semaphore* *sp)
Performs a signal operation on a semaphore.
- void **chSemSignall** (*Semaphore* *sp)
Performs a signal operation on a semaphore.
- void **chSemAddCounterl** (*Semaphore* *sp, *cnt_t* n)
Adds the specified value to the semaphore counter.
- *msg_t* **chSemSignalWait** (*Semaphore* *sp, *Semaphore* *spw)
Performs atomic signal and wait operations on two semaphores.

9.41 chsem.h File Reference

9.41.1 Detailed Description

Semaphores macros and structures.

Data Structures

- struct **Semaphore**
Semaphore structure.

Functions

- void **chSemInit** (*Semaphore* *sp, *cnt_t* n)
Initializes a semaphore with the specified counter value.
- void **chSemReset** (*Semaphore* *sp, *cnt_t* n)
Performs a reset operation on the semaphore.
- void **chSemResetl** (*Semaphore* *sp, *cnt_t* n)

- `msg_t chSemWait (Semaphore *sp)`
Performs a wait operation on a semaphore.
- `msg_t chSemWaitS (Semaphore *sp)`
Performs a wait operation on a semaphore.
- `msg_t chSemWaitTimeout (Semaphore *sp, systime_t time)`
Performs a wait operation on a semaphore with timeout specification.
- `msg_t chSemWaitTimeoutS (Semaphore *sp, systime_t time)`
Performs a wait operation on a semaphore with timeout specification.
- `void chSemSignal (Semaphore *sp)`
Performs a signal operation on a semaphore.
- `void chSemSignall (Semaphore *sp)`
Performs a signal operation on a semaphore.
- `void chSemAddCounterl (Semaphore *sp, cnt_t n)`
Adds the specified value to the semaphore counter.
- `msg_t chSemSignalWait (Semaphore *sp, Semaphore *spw)`
Performs atomic signal and wait operations on two semaphores.

Defines

- `#define _SEMAPHORE_DATA(name, n) {_THREADSQUEUE_DATA(name.s_queue), n}`
Data part of a static semaphore initializer.
- `#define SEMAPHORE_DECL(name, n) Semaphore name = _SEMAPHORE_DATA(name, n)`
Static semaphore initializer.

Macro Functions

- `#define chSemFastWaitl(sp) ((sp)->s_cnt--)`
Decreases the semaphore counter.
- `#define chSemFastSignall(sp) ((sp)->s_cnt++)`
Increases the semaphore counter.
- `#define chSemGetCounterl(sp) ((sp)->s_cnt)`
Returns the semaphore counter current value.

Typedefs

- `typedef struct Semaphore Semaphore`
Semaphore structure.

9.42 chstreams.h File Reference

9.42.1 Detailed Description

Data streams. This header defines abstract interfaces useful to access generic data streams in a standardized way.

Data Structures

- `struct BaseSequentialStreamVMT`
BaseSequentialStream virtual methods table.
- `struct BaseSequentialStream`
Base stream class.

Defines

- `#define _base_sequential_stream_methods`
BaseSequentialStream specific methods.
- `#define _base_sequential_stream_data`
BaseSequentialStream specific data.

Macro Functions (BaseSequentialStream)

- `#define chSequentialStreamWrite(ip, bp, n) ((ip)->vmt->write(ip, bp, n))`
Sequential Stream write.
- `#define chSequentialStreamRead(ip, bp, n) ((ip)->vmt->read(ip, bp, n))`
Sequential Stream read.

9.43 chsys.c File Reference

9.43.1 Detailed Description

System related code. `#include "ch.h"`

Functions

- `void _idle_thread (void *p)`
This function implements the idle thread infinite loop.
- `void chSysInit (void)`
ChibiOS/RT initialization.
- `void chSysTimerHandlerl (void)`
Handles time ticks for round robin preemption and timer increments.

9.44 chsys.h File Reference

9.44.1 Detailed Description

System related macros and structures.

Functions

- `void chSysInit (void)`
ChibiOS/RT initialization.
- `void chSysTimerHandlerl (void)`
Handles time ticks for round robin preemption and timer increments.

Defines

Macro Functions

- `#define chSysGetIdleThread() ((Thread *)_idle_thread_wa)`
Returns a pointer to the idle thread.
- `#define chSysHalt() port_halt()`
Halts the system.
- `#define chSysSwitch(ntp, otp)`

- `#define chSysDisable()`
Performs a context switch.
- `#define chSysSuspend()`
Raises the system interrupt priority mask to the maximum level.
- `#define chSysEnable()`
Raises the system interrupt priority mask to system level.
- `#define chSysLock()`
Lowers the system interrupt priority mask to user level.
- `#define chSysUnlock()`
Enters the kernel lock mode.
- `#define chSysLockFromIsr()`
Leaves the kernel lock mode.
- `#define chSysLockFromIsr()`
Enters the kernel lock mode from within an interrupt handler.
- `#define chSysUnlockFromIsr()`
Leaves the kernel lock mode from within an interrupt handler.

ISRs abstraction macros

- `#define CH_IRQ_PROLOGUE()`
IRQ handler enter code.
- `#define CH_IRQ_EPILOGUE()`
IRQ handler exit code.
- `#define CH_IRQ_HANDLER(id) PORT_IRQ_HANDLER(id)`
Standard normal IRQ handler declaration.

Fast ISRs abstraction macros

- `#define CH_FAST_IRQ_HANDLER(id) PORT_FAST_IRQ_HANDLER(id)`
Standard fast IRQ handler declaration.

9.45 chthreads.c File Reference

9.45.1 Detailed Description

Threads code. `#include "ch.h"`

Functions

- `Thread * _thread_init (Thread *tp, tprio_t prio)`
Initializes a thread structure.
- `void _thread_memfill (uint8_t *startp, uint8_t *endp, uint8_t v)`
Memory fill utility.
- `Thread * chThdCreate1 (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `Thread * chThdCreateStatic (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `tprio_t chThdSetPriority (tprio_t newprio)`
Changes the running thread priority level then reschedules if necessary.
- `Thread * chThdResume (Thread *tp)`
Resumes a suspended thread.
- `void chThdTerminate (Thread *tp)`
Requests a thread termination.
- `void chThdSleep (systime_t time)`
Suspends the invoking thread for the specified time.

- void `chThdSleepUntil (systime_t time)`
Suspends the invoking thread until the system time arrives to the specified value.
- void `chThdYield (void)`
Yields the time slot.
- void `chThdExit (msg_t msg)`
Terminates the current thread.
- void `chThdExitS (msg_t msg)`
Terminates the current thread.
- `msg_t chThdWait (Thread *tp)`
Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.

9.46 chthreads.h File Reference

9.46.1 Detailed Description

Threads macros and structures.

Data Structures

- struct `Thread`
Structure representing a thread.

Functions

- `Thread * _thread_init (Thread *tp, tprio_t prio)`
Initializes a thread structure.
- void `_thread_memfill (uint8_t *startp, uint8_t *endp, uint8_t v)`
Memory fill utility.
- `Thread * chThdCreate1 (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `Thread * chThdCreateStatic (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)`
Creates a new thread into a static memory area.
- `tprio_t chThdSetPriority (tprio_t newprio)`
Changes the running thread priority level then reschedules if necessary.
- `Thread * chThdResume (Thread *tp)`
Resumes a suspended thread.
- void `chThdTerminate (Thread *tp)`
Requests a thread termination.
- void `chThdSleep (systime_t time)`
Suspends the invoking thread for the specified time.
- void `chThdSleepUntil (systime_t time)`
Suspends the invoking thread until the system time arrives to the specified value.
- void `chThdYield (void)`
Yields the time slot.
- void `chThdExit (msg_t msg)`
Terminates the current thread.
- void `chThdExitS (msg_t msg)`
Terminates the current thread.
- `msg_t chThdWait (Thread *tp)`
Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.

Defines

Thread states

- #define THD_STATE_READY 0
Waiting on the ready list.
- #define THD_STATE_CURRENT 1
Currently running.
- #define THD_STATE_SUSPENDED 2
Created in suspended state.
- #define THD_STATE_WTSEM 3
Waiting on a semaphore.
- #define THD_STATE_WTMTX 4
Waiting on a mutex.
- #define THD_STATE_WTCOND 5
Waiting on a condition variable.
- #define THD_STATE_SLEEPING 6
Waiting in chThdSleep() or chThdSleepUntil().
- #define THD_STATE_WTEXIT 7
Waiting in chThdWait().
- #define THD_STATE_WTOREVT 8
Waiting for an event.
- #define THD_STATE_WTANDEVT 9
Waiting for several events.
- #define THD_STATE SNDMSGQ 10
Sending a message, in queue.
- #define THD_STATE SNDMSG 11
Sent a message, waiting answer.
- #define THD_STATE_WTMSG 12
Waiting for a message.
- #define THD_STATE_WTQUEUE 13
Waiting on an I/O queue.
- #define THD_STATE_FINAL 14
Thread terminated.
- #define THD_STATE_NAMES
Thread states as array of strings.

Thread flags and attributes

- #define THD_MEM_MODE_MASK 3
Thread memory mode mask.
- #define THD_MEM_MODE_STATIC 0
Static thread.
- #define THD_MEM_MODE_HEAP 1
Thread allocated from a Memory Heap.
- #define THD_MEM_MODE_MEMPOOL 2
Thread allocated from a Memory Pool.
- #define THD_TERMINATE 4
Termination requested flag.

Macro Functions

- #define chThdSelf() currp
Returns a pointer to the current Thread.
- #define chThdGetPriority() (currp->p_prio)
Returns the current thread priority.
- #define chThdGetTicks(tp) ((tp)->p_time)
Returns the number of ticks consumed by the specified thread.
- #define chThdLSS() (void *) (currp + 1)
Returns the pointer to the Thread local storage area, if any.

- `#define chThdTerminated(tp) ((tp)->p_state == THD_STATE_FINAL)`
Verifies if the specified thread is in the THD_STATE_FINAL state.
- `#define chThdShouldTerminate() (currp->p_flags & THD_TERMINATE)`
Verifies if the current thread has a termination request pending.
- `#define chThdResumeI(tp) chSchReadyI(tp)`
Resumes a thread created with chThdCreateI().
- `#define chThdSleepS(time) chSchGoSleepTimeoutS(THD_STATE_SLEEPING, time)`
Suspends the invoking thread for the specified time.
- `#define chThdSleepSeconds(sec) chThdSleep(S2ST(sec))`
Delays the invoking thread for the specified number of seconds.
- `#define chThdSleepMilliseconds(msec) chThdSleep(MS2ST(msec))`
Delays the invoking thread for the specified number of milliseconds.
- `#define chThdSleepMicroseconds(usec) chThdSleep(US2ST(usec))`
Delays the invoking thread for the specified number of microseconds.

Typedefs

- `typedef msg_t(* tfunc_t)(void *)`
Thread function.

9.47 chtypes.h File Reference

9.47.1 Detailed Description

ARM7/9 architecture port system types.

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

Defines

- `#define INLINE inline`
Inline function modifier.
- `#define ROMCONST const`
ROM constant modifier.
- `#define PACK_STRUCT_STRUCT __attribute__((packed))`
Packed structure modifier (within).
- `#define PACK_STRUCT_BEGIN`
Packed structure modifier (before).
- `#define PACK_STRUCT_END`
Packed structure modifier (after).

Typedefs

- `typedef int32_t bool_t`
- `typedef uint8_t tmode_t`
- `typedef uint8_t tstate_t`
- `typedef uint8_t trefs_t`
- `typedef uint32_t tprio_t`
- `typedef int32_t msg_t`
- `typedef int32_t eventid_t`
- `typedef uint32_t eventmask_t`
- `typedef uint32_t systime_t`
- `typedef int32_t cnt_t`

9.48 chvt.c File Reference

9.48.1 Detailed Description

Time and Virtual Timers related code. #include "ch.h"

Functions

- void [_vt_init](#) (void)
Virtual Timers initialization.
- void [chVTSetl](#) ([VirtualTimer](#) *vtp, [systime_t](#) time, [vfunc_t](#) vfunc, void *par)
Enables a virtual timer.
- void [chVTResetl](#) ([VirtualTimer](#) *vtp)
Disables a Virtual Timer.
- [bool_t chTimelsWithin](#) ([systime_t](#) start, [systime_t](#) end)
Checks if the current system time is within the specified time window.

Variables

- [VTLlist vtlist](#)
Virtual timers delta list header.

9.49 chvt.h File Reference

9.49.1 Detailed Description

Time macros and structures.

Data Structures

- struct [VirtualTimer](#)
Virtual Timer descriptor structure.
- struct [VTLlist](#)
Virtual timers list header.

Functions

- void [_vt_init](#) (void)
Virtual Timers initialization.
- void [chVTSetl](#) ([VirtualTimer](#) *vtp, [systime_t](#) time, [vfunc_t](#) vfunc, void *par)
Enables a virtual timer.
- void [chVTResetl](#) ([VirtualTimer](#) *vtp)
Disables a Virtual Timer.
- [bool_t chTimelsWithin](#) ([systime_t](#) start, [systime_t](#) end)
Checks if the current system time is within the specified time window.

Variables

- [VTLlist vtlist](#)
Virtual timers delta list header.

Defines

Time conversion utilities

- `#define S2ST(sec) ((systime_t)((sec) * CH_FREQUENCY))`
Seconds to system ticks.
- `#define MS2ST(msec) ((systime_t)((((msec) - 1L) * CH_FREQUENCY) / 1000L) + 1L))`
Milliseconds to system ticks.
- `#define US2ST(usec) ((systime_t)((((usec) - 1L) * CH_FREQUENCY) / 1000000L) + 1L))`
Microseconds to system ticks.

Macro Functions

- `#define chVTDoTick()`
Virtual timers ticker.
- `#define chVTIsArmed(vtp) ((vtp)->vt_func != NULL)`
Returns TRUE if the specified timer is armed.
- `#define chTimeNow() (vtlist.vt_systime)`
Current system time.

Typedefs

- `typedef void(* vfunc_t)(void *)`
Virtual Timer callback function.
- `typedef struct VirtualTimer VirtualTimer`
Virtual Timer structure type.

9.50 crt0.s File Reference

9.50.1 Detailed Description

Generic ARM7/9 startup file for ChibiOS/RT.

9.51 vectors.s File Reference

9.51.1 Detailed Description

Interrupt vectors for the AT91SAM7 family.

Functions

- `void _unhandled_exception (void)`
Unhandled exceptions handler.

9.52 vectors.s File Reference

9.52.1 Detailed Description

Interrupt vectors for the LPC214x family.

Functions

- void [_unhandled_exception](#) (void)

Unhandled exceptions handler.

Index

_BSEMAPHORE_DATA
 Binary Semaphores, 73

_CHIBIOS_RT_
 Version Numbers and Identification, 15

_CONDVAR_DATA
 Condition Variables, 92

_EVENTSOURCE_DATA
 Event Flags, 103

_INPUTQUEUE_DATA
 I/O Queues, 158

_MAILBOX_DATA
 Mailboxes, 120

_MEMORYPOOL_DATA
 Memory Pools, 130

_MUTEX_DATA
 Mutexes, 83

_OUTPUTQUEUE_DATA
 I/O Queues, 161

_SEMAPHORE_DATA
 Counting Semaphores, 70

_THREADSQUEUE_DATA
 Internals, 177

_base_asynchronous_channel_data
 Abstract I/O Channels, 145

_base_asynchronous_channel_methods
 Abstract I/O Channels, 145

_base_channel_data
 Abstract I/O Channels, 141

_base_channel_methods
 Abstract I/O Channels, 141

_base_file_stream_data
 Abstract File Streams, 137

_base_file_stream_methods
 Abstract File Streams, 137

_base_sequential_stream_data
 Abstract Sequential Streams, 135

_base_sequential_stream_methods
 Abstract Sequential Streams, 135

_ch_get_and_clear_flags_impl
 Abstract I/O Channels, 147

_core_init
 Core Memory Manager, 122

_heap_init
 Heaps, 125

_idle_thread
 System Management, 28
 Version Numbers and Identification, 15

_scheduler_init
 Scheduler, 34

_thread_init

Threads, 44

_thread_memfill
 Threads, 44

_trace_init
 Debug, 167

_unhandled_exception
 AT91SAM7 Interrupt Vectors, 192
 LPC214x Interrupt Vectors, 193

_vt_init
 Time and Virtual Timers, 57

ABSPRIO
 Scheduler, 39

Abstract File Streams, 136

 _base_file_stream_data, 137

 _base_file_stream_methods, 137

 chFileStreamClose, 138

 chFileStreamGetError, 138

 chFileStreamGetPosition, 139

 chFileStreamGetSize, 138

 chFileStreamSeek, 139

 FILE_ERROR, 137

 FILE_OK, 137

 fileoffset_t, 139

Abstract I/O Channels, 139

 _base_asynchronous_channel_data, 145

 _base_asynchronous_channel_methods, 145

 _base_channel_data, 141

 _base_channel_methods, 141

 _ch_get_and_clear_flagsImpl, 147

 chIOAddFlagsI, 146

 chIOGet, 143

 chIOGetAndClearFlags, 146

 chIOGetEventSource, 146

 chIOGetTimeout, 143

 chIOGetWouldBlock, 142

 chIOPut, 142

 chIOPutTimeout, 142

 chIOPutWouldBlock, 141

 chIOReadTimeout, 144

 chIOWriteTimeout, 144

 IO_CONNECTED, 145

 IO_DISCONNECTED, 145

 IO_INPUT_AVAILABLE, 145

 IO_NO_ERROR, 145

 IO_OUTPUT_EMPTY, 145

 IO_TRANSMISSION_END, 145

 ioflags_t, 147

Abstract Sequential Streams, 134

 _base_sequential_stream_data, 135

_base_sequential_stream_methods, 135
chSequentialStreamRead, 136
chSequentialStreamWriter, 135
ALL_EVENTS
Event Flags, 103
ARM7/9, 178
ARM_CORE
AT91SAM7 Specific Parameters, 191
LPC214x Specific Parameters, 192
ARM_CORE_ARM7TDMI
Core Port Implementation, 183
ARM_CORE_ARM9
Core Port Implementation, 183
ARM_ENABLE_WFI_IDLE
Core Port Implementation, 183
arpparams.h, 249
AT91SAM7 Interrupt Vectors, 192
_unhandled_exception, 192
AT91SAM7 Specific Parameters, 191
ARM_CORE, 191
port_wait_for_interrupt, 191

Base Kernel Services, 25
BaseAsynchronousChannel, 194
vmt, 195
BaseAsynchronousChannelVMT, 195
BaseChannel, 197
vmt, 199
BaseChannelVMT, 199
BaseFileStream, 200
vmt, 202
BaseFileStreamVMT, 202
BaseSequentialStream, 203
vmt, 205
BaseSequentialStreamVMT, 205
Binary Semaphores, 71
_BSEMAPHORE_DATA, 73
BSEMAPHORE_DECL, 73
chBSemGetStatel, 76
chBSemInit, 73
chBSemReset, 75
chBSemResetl, 75
chBSemSignal, 76
chBSemSignall, 76
chBSemWait, 73
chBSemWaitS, 74
chBSemWaitTimeout, 74
chBSemWaitTimeoutS, 74
BinarySemaphore, 205
bool_t
Core Port Implementation, 189
BSEMAPHORE_DECL
Binary Semaphores, 73

c_queue
CondVar, 213
ch.h, 249
CH_ARCHITECTURE_ARM
Core Port Implementation, 183
CH_ARCHITECTURE_ARMx
Core Port Implementation, 183
CH_ARCHITECTURE_NAME
Core Port Implementation, 183
CH_COMPILER_NAME
Core Port Implementation, 184
CH_CORE_VARIANT_NAME
Core Port Implementation, 183
CH_DBG_ENABLE_ASSERTS
Configuration, 22
CH_DBG_ENABLE_CHECKS
Configuration, 22
CH_DBG_ENABLE_STACK_CHECK
Configuration, 23
CH_DBG_ENABLE_TRACE
Configuration, 23
CH_DBG_FILL_THREADS
Configuration, 23
CH_DBG_SYSTEM_STATE_CHECK
Configuration, 22
CH_DBG_THREADS_PROFILING
Configuration, 23
CH_FAST_IRQ_HANDLER
System Management, 32
CH_FREQUENCY
Configuration, 17
CH_IRQ_EPILOGUE
System Management, 32
CH_IRQ_HANDLER
System Management, 32
CH_IRQ_PROLOGUE
System Management, 31
CH_KERNEL_MAJOR
Version Numbers and Identification, 15
CH_KERNEL_MINOR
Version Numbers and Identification, 15
CH_KERNEL_PATCH
Version Numbers and Identification, 15
CH_KERNEL_VERSION
Version Numbers and Identification, 15
CH_MEMCORE_SIZE
Configuration, 18
CH_NO_IDLE_THREAD
Configuration, 18
CH_OPTIMIZE_SPEED
Configuration, 18
CH_PORT_INFO
Core Port Implementation, 184
CH_STACK_FILL_VALUE
Debug, 173
ch_swc_event_t, 207
se_state, 209
se_time, 209
se_tp, 209
se_wtobjp, 209
CH_THREAD_FILL_VALUE
Debug, 173
CH_TIME_QUANTUM
Configuration, 17

CH_TRACE_BUFFER_SIZE
 Debug, 173
ch_trace_buffer_t, 209
 tb_buffer, 211
 tb_ptr, 211
 tb_size, 211
CH_USE_CONDVARSL
 Configuration, 19
CH_USE_CONDVARSL_TIMEOUT
 Configuration, 20
CH_USE_DYNAMIC
 Configuration, 22
CH_USE_EVENTS
 Configuration, 20
CH_USE_EVENTS_TIMEOUT
 Configuration, 20
CH_USE_HEAP
 Configuration, 21
CH_USE_MAILBOXES
 Configuration, 21
CH_USE_MALLOC_HEAP
 Configuration, 21
CH_USE_MEMCORE
 Configuration, 21
CH_USE_MEMPOOLS
 Configuration, 22
CH_USE_MESSAGES
 Configuration, 20
CH_USE_MESSAGES_PRIORITY
 Configuration, 20
CH_USE_MUTEXES
 Configuration, 19
CH_USE_QUEUES
 Configuration, 21
CH_USE_REGISTRY
 Configuration, 18
CH_USE_SEMAPHORES
 Configuration, 19
CH_USE_SEMAPHORES_PRIORITY
 Configuration, 19
CH_USE_SEMSW
 Configuration, 19
CH_USE_WAITEXIT
 Configuration, 19
chbsem.h, 251
chBSemGetStatel
 Binary Semaphores, 76
chBSemInit
 Binary Semaphores, 73
chBSemReset
 Binary Semaphores, 75
chBSemResetl
 Binary Semaphores, 75
chBSemSignal
 Binary Semaphores, 76
chBSemSignall
 Binary Semaphores, 76
chBSemWait
 Binary Semaphores, 73
chBSemWaitS
 Binary Semaphores, 74
chBSemWaitTimeout
 Binary Semaphores, 74
chBSemWaitTimeoutS
 Binary Semaphores, 74
chcond.c, 251
chcond.h, 252
chCondBroadcast
 Condition Variables, 86
chCondBroadcastl
 Condition Variables, 87
chCondInit
 Condition Variables, 85
chCondSignal
 Condition Variables, 85
chCondSignall
 Condition Variables, 86
chCondWait
 Condition Variables, 88
chCondWaitS
 Condition Variables, 89
chCondWaitTimeout
 Condition Variables, 90
chCondWaitTimeoutS
 Condition Variables, 91
chconf.h, 253
chcore.c, 255
chcore.h, 255
chCoreAlloc
 Core Memory Manager, 122
chCoreAllocl
 Core Memory Manager, 122
chcoreasm.s, 256
chCoreStatus
 Core Memory Manager, 123
chDbgAssert
 Debug, 173
chDbgCheck
 Debug, 173
chDbgCheckClassl
 Debug, 171
chDbgCheckClassS
 Debug, 171
chDbgPanic
 Debug, 172
chdebug.c, 257
chdebug.h, 258
chdynamic.c, 258
chdynamic.h, 259
chevents.c, 259
chevents.h, 260
chEvtAddFlags
 Event Flags, 95
chEvtBroadcast
 Event Flags, 104
chEvtBroadcastFlags
 Event Flags, 97
chEvtBroadcastFlagsl

Event Flags, 97
chEvtBroadcastl
 Event Flags, 104
chEvtClearFlags
 Event Flags, 95
chEvtDispatch
 Event Flags, 98
chEvtInit
 Event Flags, 104
chEvtIsListeningl
 Event Flags, 104
chEvtRegister
 Event Flags, 103
chEvtRegisterMask
 Event Flags, 95
chEvtSignalFlags
 Event Flags, 96
chEvtSignalFlagsl
 Event Flags, 96
chEvtUnregister
 Event Flags, 95
chEvtWaitAll
 Event Flags, 102
chEvtWaitAllTimeout
 Event Flags, 100
chEvtWaitAny
 Event Flags, 101
chEvtWaitAnyTimeout
 Event Flags, 99
chEvtWaitOne
 Event Flags, 101
chEvtWaitOneTimeout
 Event Flags, 98
chfiles.h, 261
chFileStreamClose
 Abstract File Streams, 138
chFileStreamGetError
 Abstract File Streams, 138
chFileStreamGetPosition
 Abstract File Streams, 139
chFileStreamGetSize
 Abstract File Streams, 138
chFileStreamSeek
 Abstract File Streams, 139
chheap.c, 262
chheap.h, 262
chHeapAlloc
 Heaps, 126
chHeapFree
 Heaps, 126
chHeapInit
 Heaps, 125
chHeapStatus
 Heaps, 126
chinline.h, 263
chIOAddFlagsl
 Abstract I/O Channels, 146
chioch.h, 263
chIOGet
 Abstract I/O Channels, 143
chIOGetAndClearFlags
 Abstract I/O Channels, 146
chIOGetEventSource
 Abstract I/O Channels, 146
chIOGetTimeout
 Abstract I/O Channels, 143
chIOGetWouldBlock
 Abstract I/O Channels, 142
chIOPut
 Abstract I/O Channels, 142
chIOPutTimeout
 Abstract I/O Channels, 142
chIOPutWouldBlock
 Abstract I/O Channels, 141
chIOReadTimeout
 Abstract I/O Channels, 144
chIOWriteTimeout
 Abstract I/O Channels, 144
chIQGet
 I/O Queues, 158
chIQGetEmptyl
 I/O Queues, 157
chIQGetFulll
 I/O Queues, 156
chIQGetTimeout
 I/O Queues, 151
chIQInit
 I/O Queues, 150
chIQIsEmptyl
 I/O Queues, 157
chIQIsFulll
 I/O Queues, 157
chIQPutl
 I/O Queues, 151
chIQReadTimeout
 I/O Queues, 152
chIQResetl
 I/O Queues, 150
chlists.c, 265
chlists.h, 265
chMBFetch
 Mailboxes, 116
chMBFetchl
 Mailboxes, 118
chMBFetchS
 Mailboxes, 117
chMBGetFreeCountl
 Mailboxes, 119
chMBGetUsedCountl
 Mailboxes, 119
chMBInit
 Mailboxes, 110
chmboxes.c, 266
chmboxes.h, 267
chMBPeekl
 Mailboxes, 119
chMBPost
 Mailboxes, 111

chMBPostAhead
 Mailboxes, 114
chMBPostAheadI
 Mailboxes, 115
chMBPostAheadS
 Mailboxes, 114
chMBPostI
 Mailboxes, 113
chMBPostS
 Mailboxes, 112
chMBReset
 Mailboxes, 110
chMBSizel
 Mailboxes, 119
chmemcore.c, 268
chmemcore.h, 268
chmempools.c, 269
chmempools.h, 269
chmsg.c, 270
chmsg.h, 270
chMsgGet
 Synchronous Messages, 108
chMsgGetS
 Synchronous Messages, 108
chMsgIsPendingI
 Synchronous Messages, 107
chMsgRelease
 Synchronous Messages, 107
chMsgReleaseS
 Synchronous Messages, 108
chMsgSend
 Synchronous Messages, 106
chMsgWait
 Synchronous Messages, 106
chmtx.c, 271
chmtx.h, 271
chMtxInit
 Mutexes, 78
chMtxLock
 Mutexes, 79
chMtxLockS
 Mutexes, 79
chMtxQueueNotEmptyS
 Mutexes, 84
chMtxTryLock
 Mutexes, 80
chMtxTryLockS
 Mutexes, 81
chMtxUnlock
 Mutexes, 81
chMtxUnlockAll
 Mutexes, 83
chMtxUnlockS
 Mutexes, 82
chOQGetEmptyI
 I/O Queues, 159
chOQGetFullI
 I/O Queues, 159
chOQGetI
 I/O Queues, 154
chOQInit
 I/O Queues, 152
chOQIsEmptyI
 I/O Queues, 160
chOQIsFullI
 I/O Queues, 160
chOQPut
 I/O Queues, 160
chOQPutTimeout
 I/O Queues, 153
chOQResetI
 I/O Queues, 153
chOQWriteTimeout
 I/O Queues, 155
chPoolAlloc
 Memory Pools, 129
chPoolAllocI
 Memory Pools, 128
chPoolFree
 Memory Pools, 130
chPoolFreeI
 Memory Pools, 129
chPoolInit
 Memory Pools, 128
chQsizel
 I/O Queues, 156
chQSpaceI
 I/O Queues, 156
chqueues.c, 272
chqueues.h, 273
chRegFirstThread
 Registry, 163
chRegGetThreadName
 Registry, 164
chregistry.c, 275
chregistry.h, 275
chRegNextThread
 Registry, 163
chRegSetThreadName
 Registry, 164
chSchCanYieldS
 Scheduler, 40
chsched.c, 276
chsched.h, 276
chSchDoReschedule
 Scheduler, 38
chSchDoYieldS
 Scheduler, 40
chSchGoSleepS
 Scheduler, 35
chSchGoSleepTimeoutS
 Scheduler, 35
chSchIsPreemptionRequired
 Scheduler, 37
chSchIsRescRequiredI
 Scheduler, 40
chSchPreemption
 Scheduler, 40

chSchReadyI
 Scheduler, 34
chSchRescheduleS
 Scheduler, 37
chSchWakeupS
 Scheduler, 36
chsem.c, 278
chsem.h, 278
chSemAddCounterI
 Counting Semaphores, 69
chSemFastSignall
 Counting Semaphores, 71
chSemFastWaitI
 Counting Semaphores, 71
chSemGetCounterI
 Counting Semaphores, 71
chSemInit
 Counting Semaphores, 63
chSemReset
 Counting Semaphores, 63
chSemResetI
 Counting Semaphores, 64
chSemSignal
 Counting Semaphores, 67
chSemSignall
 Counting Semaphores, 68
chSemSignalWait
 Counting Semaphores, 69
chSemWait
 Counting Semaphores, 65
chSemWaitS
 Counting Semaphores, 65
chSemWaitTimeout
 Counting Semaphores, 66
chSemWaitTimeoutS
 Counting Semaphores, 67
chSequentialStreamRead
 Abstract Sequential Streams, 136
chSequentialStreamWriter
 Abstract Sequential Streams, 135
chstreams.h, 279
chsys.c, 280
chsys.h, 280
chSysDisable
 System Management, 29
chSysEnable
 System Management, 30
chSysGetIdleThread
 System Management, 28
chSysHalt
 System Management, 28
chSysInit
 System Management, 27
chSysLock
 System Management, 30
chSysLockFromIlsr
 System Management, 31
chSysSuspend
 System Management, 29
chSysSwitch
 System Management, 29
chSysTimerHandlerI
 System Management, 27
chSysUnlock
 System Management, 30
chSysUnlockFromIlsr
 System Management, 31
chThdAddRef
 Dynamic Threads, 131
chThdCreateFromHeap
 Dynamic Threads, 132
chThdCreateFromMemoryPool
 Dynamic Threads, 133
chThdCreateI
 Threads, 44
chThdCreateStatic
 Threads, 45
chThdExit
 Threads, 48
chThdExitS
 Threads, 49
chThdGetPriority
 Threads, 53
chThdGetTicks
 Threads, 53
chThdLS
 Threads, 53
chThdRelease
 Dynamic Threads, 132
chThdResume
 Threads, 46
chThdResumel
 Threads, 54
chThdSelf
 Threads, 53
chThdSetPriority
 Threads, 46
chThdShouldTerminate
 Threads, 54
chThdSleep
 Threads, 48
chThdSleepMicroseconds
 Threads, 55
chThdSleepMilliseconds
 Threads, 55
chThdSleepS
 Threads, 54
chThdSleepSeconds
 Threads, 55
chThdSleepUntil
 Threads, 48
chThdTerminate
 Threads, 47
chThdTerminated
 Threads, 54
chThdWait
 Threads, 50
chThdYield

Threads, 48
chthreads.c, 281
chthreads.h, 282
chTimelsWithin
 Time and Virtual Timers, 58
chTimeNow
 Time and Virtual Timers, 60
ctypes.h, 284
chvt.c, 285
chvt.h, 285
chVTDotickl
 Time and Virtual Timers, 60
chVTIsArmedl
 Time and Virtual Timers, 60
chVTResetl
 Time and Virtual Timers, 58
chVTSel
 Time and Virtual Timers, 57
cnt_t
 Core Port Implementation, 190
Condition Variables, 84
 _CONDVAR_DATA, 92
 chCondBroadcast, 86
 chCondBroadcastl, 87
 chCondlInit, 85
 chCondSignal, 85
 chCondSignall, 86
 chCondWait, 88
 chCondWaitS, 89
 chCondWaitTimeout, 90
 chCondWaitTimeoutS, 91
 CondVar, 92
 CONDVAR_DECL, 92
CondVar, 211
 c_queue, 213
 Condition Variables, 92
CONDVAR_DECL
 Condition Variables, 92
Configuration, 15
 CH_DBG_ENABLE_ASSERTS, 22
 CH_DBG_ENABLE_CHECKS, 22
 CH_DBG_ENABLE_STACK_CHECK, 23
 CH_DBG_ENABLE_TRACE, 23
 CH_DBG_FILL_THREADS, 23
 CH_DBG_SYSTEM_STATE_CHECK, 22
 CH_DBG_THREADS_PROFILING, 23
 CH_FREQUENCY, 17
 CH_MEMCORE_SIZE, 18
 CH_NO_IDLE_THREAD, 18
 CH_OPTIMIZE_SPEED, 18
 CH_TIME_QUANTUM, 17
 CH_USE_CONDVARs, 19
 CH_USE_CONDVARs_TIMEOUT, 20
 CH_USE_DYNAMIC, 22
 CH_USE_EVENTS, 20
 CH_USE_EVENTS_TIMEOUT, 20
 CH_USE_HEAP, 21
 CH_USE_MAILBOXES, 21
 CH_USE_MALLOC_HEAP, 21
 CH_USE_MEMCORE, 21
 CH_USE_MEMPOOLS, 22
 CH_USE_MESSAGES, 20
 CH_USE_MESSAGES_PRIORITY, 20
 CH_USE_MUTEXES, 19
 CH_USE_QUEUES, 21
 CH_USE_REGISTRY, 18
 CH_USE_SEMAPHORES, 19
 CH_USE_SEMAPHORES_PRIORITY, 19
 CH_USE_SEMSW, 19
 CH_USE_WAITEXIT, 19
 IDLE_LOOP_HOOK, 24
 SYSTEM_HALT_HOOK, 25
 SYSTEM_TICK_EVENT_HOOK, 24
 THREAD_CONTEXT_SWITCH_HOOK, 24
 THREAD_EXT_EXIT_HOOK, 24
 THREAD_EXT_FIELDS, 23
 THREAD_EXT_INIT_HOOK, 24
Configuration Options, 180
context, 213
Core Memory Manager, 121
 _core_init, 122
 chCoreAlloc, 122
 chCoreAllocl, 122
 chCoreStatus, 123
 MEM_ALIGN_MASK, 123
 MEM_ALIGN_NEXT, 124
 MEM_ALIGN_PREV, 123
 MEM_ALIGN_SIZE, 123
 MEM_IS_ALIGNED, 124
 memgetfunc_t, 124
Core Port Implementation, 181
 ARM_CORE_ARM7TDMI, 183
 ARM_CORE_ARM9, 183
 ARM_ENABLE_WFI_IDLE, 183
 bool_t, 189
 CH_ARCHITECTURE_ARM, 183
 CH_ARCHITECTURE_ARMx, 183
 CH_ARCHITECTURE_NAME, 183
 CH_COMPILER_NAME, 184
 CH_CORE_VARIANT_NAME, 183
 CH_PORT_INFO, 184
 cnt_t, 190
 eventid_t, 189
 eventmask_t, 189
 INLINE, 188
 msg_t, 189
 PACK_STRUCT_BEGIN, 188
 PACK_STRUCT_END, 188
 PACK_STRUCT_STRUCT, 188
 port_disable, 187
 port_enable, 187
 PORT_FAST_IRQ_HANDLER, 186
 port_halt, 183
 PORT_IDLE_THREAD_STACK_SIZE, 184
 port_init, 186
 PORT_INT_REQUIRED_STACK, 185
 PORT_IRQ_EPILOGUE, 185
 PORT_IRQ_HANDLER, 186

PORT_IRQ_PROLOGUE, 185
port_lock, 186
port_lock_from_isr, 186
port_suspend, 187
port_switch, 187
port_unlock, 186
port_unlock_from_isr, 187
regarm_t, 189
ROMCONST, 188
SETUP_CONTEXT, 184
STACK_ALIGN, 185
stkalign_t, 189
systime_t, 189
THD_WA_SIZE, 185
tmode_t, 189
tprio_t, 189
trefs_t, 189
tstate_t, 189
WORKING_AREA, 185
Counting Semaphores, 61
 _SEMAPHORE_DATA, 70
 chSemAddCounterl, 69
 chSemFastSignall, 71
 chSemFastWaitl, 71
 chSemGetCounterl, 71
 chSemInit, 63
 chSemReset, 63
 chSemResetl, 64
 chSemSignal, 67
 chSemSignall, 68
 chSemSignalWait, 69
 chSemWait, 65
 chSemWaitS, 65
 chSemWaitTimeout, 66
 chSemWaitTimeoutS, 67
Semaphore, 71
 SEMAPHORE_DECL, 70
crt0.s, 286
currp
 Scheduler, 39

dbg_check_disable
 Debug, 167
dbg_check_enable
 Debug, 168
dbg_check_enter_isr
 Debug, 170
dbg_check_leave_isr
 Debug, 171
dbg_check_lock
 Debug, 169
dbg_check_lock_from_isr
 Debug, 169
dbg_check_suspend
 Debug, 168
dbg_check_unlock
 Debug, 169
dbg_check_unlock_from_isr
 Debug, 170

dbg_isr_cnt
 Debug, 172
dbg_lock_cnt
 Debug, 172
dbg_panic_msg
 Debug, 172
dbg_trace
 Debug, 167
dbg_trace_buffer
 Debug, 172
Debug, 165
 _trace_init, 167
 CH_STACK_FILL_VALUE, 173
 CH_THREAD_FILL_VALUE, 173
 CH_TRACE_BUFFER_SIZE, 173
 chDbgAssert, 173
 chDbgCheck, 173
 chDbgCheckClassl, 171
 chDbgCheckClassS, 171
 chDbgPanic, 172
 dbg_check_disable, 167
 dbg_check_enable, 168
 dbg_check_enter_isr, 170
 dbg_check_leave_isr, 171
 dbg_check_lock, 169
 dbg_check_lock_from_isr, 169
 dbg_check_suspend, 168
 dbg_check_unlock, 169
 dbg_check_unlock_from_isr, 170
 dbg_isr_cnt, 172
 dbg_lock_cnt, 172
 dbg_panic_msg, 172
 dbg_trace, 167
 dbg_trace_buffer, 172
dequeue
 Internals, 176
Dynamic Threads, 131
 chThdAddRef, 131
 chThdCreateFromHeap, 132
 chThdCreateFromMemoryPool, 133
 chThdRelease, 132

el_listener
 EventListener, 215
el_mask
 EventListener, 215
el_next
 EventListener, 215
es_next
 EventSource, 217
Event Flags, 93
 _EVENTSOURCE_DATA, 103
 ALL_EVENTS, 103
 chEvtAddFlags, 95
 chEvtBroadcast, 104
 chEvtBroadcastFlags, 97
 chEvtBroadcastFlagsl, 97
 chEvtBroadcastl, 104
 chEvtClearFlags, 95

chEvtDispatch, 98
chEvtInit, 104
chEvtIsListeningI, 104
chEvtRegister, 103
chEvtRegisterMask, 95
chEvtSignalFlags, 96
chEvtSignalFlagsI, 96
chEvtUnregister, 95
chEvtWaitAll, 102
chEvtWaitAllTimeout, 100
chEvtWaitAny, 101
chEvtWaitAnyTimeout, 99
chEvtWaitOne, 101
chEvtWaitOneTimeout, 98
EVENT_MASK, 103
EventSource, 105
EVENTSOURCE DECL, 103
evhandler_t, 105
EVENT_MASK
 Event Flags, 103
eventid_t
 Core Port Implementation, 189
EventListener, 213
 el_listener, 215
 el_mask, 215
 el_next, 215
eventmask_t
 Core Port Implementation, 189
EventSource, 215
 es_next, 217
 Event Flags, 105
EVENTSOURCE DECL
 Event Flags, 103
evhandler_t
 Event Flags, 105
ewmask
 Thread, 238
exitcode
 Thread, 238
extctx, 217

fifo_remove
 Internals, 175
FILE_ERROR
 Abstract File Streams, 137
FILE_OK
 Abstract File Streams, 137
fileoffset_t
 Abstract File Streams, 139
firstprio
 Scheduler, 39

GenericQueue, 217
 I/O Queues, 162
 q_buffer, 219
 q_counter, 219
 q_notify, 219
 q_rptr, 219
 q_top, 219

 q_waiting, 219
 q_wptr, 219

h_free
 memory_heap, 224
h_mtx
 memory_heap, 224
h_provider
 memory_heap, 224
heap
 heap_header, 220
heap_header, 219
 heap, 220
 next, 220
 size, 220
 u, 220
Heaps, 124
 _heap_init, 125
 chHeapAlloc, 126
 chHeapFree, 126
 chHeapInit, 125
 chHeapStatus, 126
HIGHPRIO
 Scheduler, 39

I/O Queues, 147
 _INPUTQUEUE_DATA, 158
 _OUTPUTQUEUE_DATA, 161
 chIQGet, 158
 chIQGetEmptyI, 157
 chIQGetFullI, 156
 chIQGetTimeout, 151
 chIQInit, 150
 chIQIsEmptyI, 157
 chIQIsFullI, 157
 chIQPutI, 151
 chIQReadTimeout, 152
 chIQResetI, 150
 chOQGetEmptyI, 159
 chOQGetFullI, 159
 chOQGetI, 154
 chOQInit, 152
 chOQIsEmptyI, 160
 chOQIsFullI, 160
 chOQPut, 160
 chOQPutTimeout, 153
 chOQResetI, 153
 chOQWriteTimeout, 155
 chQSizel, 156
 chQSpacel, 156
 GenericQueue, 162
 InputQueue, 162
 INPUTQUEUE DECL, 159
 OutputQueue, 162
 OUTPUTQUEUE DECL, 161
 Q_EMPTY, 156
 Q_FULL, 156
 Q_OK, 155
 Q_RESET, 156

Q_TIMEOUT, 155
qnotify_t, 162
I/O Support, 134
IDLE_LOOP_HOOK
 Configuration, 24
IDLEPRIO
 Scheduler, 39
INLINE
 Core Port Implementation, 188
InputQueue
 I/O Queues, 162
INPUTQUEUE_DECL
 I/O Queues, 159
intctx, 220
Internals, 174
 _THREADSQUEUE_DATA, 177
 dequeue, 176
 fifo_remove, 175
 isempty, 177
 lifo_remove, 176
 list_init, 177
 list_insert, 176
 list_remove, 177
 notempty, 177
 prio_insert, 175
 queue_init, 177
 queue_insert, 175
 _THREADSQUEUE_DECL, 178
IO_CONNECTED
 Abstract I/O Channels, 145
IO_DISCONNECTED
 Abstract I/O Channels, 145
IO_INPUT_AVAILABLE
 Abstract I/O Channels, 145
IO_NO_ERROR
 Abstract I/O Channels, 145
IO_OUTPUT_EMPTY
 Abstract I/O Channels, 145
IO_TRANSMISSION_END
 Abstract I/O Channels, 145
ioflags_t
 Abstract I/O Channels, 147
isempty
 Internals, 177
Kernel, 14
lifo_remove
 Internals, 176
list_init
 Internals, 177
list_insert
 Internals, 176
list_remove
 Internals, 177
LOWPRIO
 Scheduler, 39
LPC214x Interrupt Vectors, 193
 _unhandled_exception, 193
LPC214x Specific Parameters, 192
ARM_CORE, 192
port_wait_for_interrupt, 192
m_next
 Mutex, 227
m_owner
 Mutex, 227
m_queue
 Mutex, 227
Mailbox, 221
 mb_buffer, 222
 mb_emptysem, 222
 mb_fullsem, 222
 mb_rptr, 222
 mb_top, 222
 mb_wptr, 222
MAILBOX_DECL
 Mailboxes, 120
Mailboxes, 109
 _MAILBOX_DATA, 120
 chMBFetch, 116
 chMBFetchl, 118
 chMBFetchS, 117
 chMBGetFreeCountl, 119
 chMBGetUsedCountl, 119
 chMBInit, 110
 chMBPeekl, 119
 chMBPost, 111
 chMBPostAhead, 114
 chMBPostAheadl, 115
 chMBPostAheadS, 114
 chMBPostl, 113
 chMBPostS, 112
 chMBReset, 110
 chMBSizel, 119
 MAILBOX_DECL, 120
mb_buffer
 Mailbox, 222
mb_emptysem
 Mailbox, 222
mb_fullsem
 Mailbox, 222
mb_rptr
 Mailbox, 222
mb_top
 Mailbox, 222
mb_wptr
 Mailbox, 222
MEM_ALIGN_MASK
 Core Memory Manager, 123
MEM_ALIGN_NEXT
 Core Memory Manager, 124
MEM_ALIGN_PREV
 Core Memory Manager, 123
MEM_ALIGN_SIZE
 Core Memory Manager, 123
MEM_IS_ALIGNED
 Core Memory Manager, 124

memgetfunc_t
 Core Memory Manager, 124

Memory Management, 120

Memory Pools, 127

- _MEMORYPOOL_DATA, 130
- chPoolAlloc, 129
- chPoolAllocI, 128
- chPoolFree, 130
- chPoolFreeI, 129
- chPoolInit, 128
- MEMORYPOOL_DECL, 131

memory_heap, 222

- h_free, 224
- h_mtx, 224
- h_provider, 224

MemoryPool, 224

- mp_next, 225
- mp_object_size, 225
- mp_provider, 225

MEMORYPOOL_DECL
 Memory Pools, 131

mp_next
 MemoryPool, 225

mp_object_size
 MemoryPool, 225

mp_provider
 MemoryPool, 225

MS2ST
 Time and Virtual Timers, 59

msg_t
 Core Port Implementation, 189

Mutex, 225

- m_next, 227
- m_owner, 227
- m_queue, 227
- Mutexes, 84

MUTEX_DECL
 Mutexes, 84

Mutexes, 77

- _MUTEX_DATA, 83
- chMtxInit, 78
- chMtxLock, 79
- chMtxLockS, 79
- chMtxQueueNotEmptyS, 84
- chMtxTryLock, 80
- chMtxTryLockS, 81
- chMtxUnlock, 81
- chMtxUnlockAll, 83
- chMtxUnlockS, 82
- Mutex, 84
- MUTEX_DECL, 84

next
 heap_header, 220

NOPRIO
 Scheduler, 39

NORMALPRIO
 Scheduler, 39

notempty

Internals, 177

OutputQueue
 I/O Queues, 162

OUTPUTQUEUE_DECL
 I/O Queues, 161

p_ctx
 Thread, 237

p_epending
 Thread, 238

p_flags
 Thread, 237

p_mpool
 Thread, 239

p_msg
 Thread, 238

p_msgqueue
 Thread, 238

p_mtxlist
 Thread, 238

p_name
 Thread, 237

p_newer
 Thread, 237

p_next
 Thread, 236

ThreadsList, 242

ThreadsQueue, 245

p_older
 Thread, 237

p_prev
 Thread, 236

ThreadsQueue, 245

p_prio
 Thread, 237

p_realprio
 Thread, 239

p_refs
 Thread, 237

p_state
 Thread, 237

p_stklimit
 Thread, 237

p_time
 Thread, 237

p_waiting
 Thread, 238

PACK_STRUCT_BEGIN
 Core Port Implementation, 188

PACK_STRUCT_END
 Core Port Implementation, 188

PACK_STRUCT_STRUCT
 Core Port Implementation, 188

ph_next
 pool_header, 227

pool_header, 227

 ph_next, 227

port_disable

Core Port Implementation, 187
port_enable
 Core Port Implementation, 187
PORT_FAST_IRQ_HANDLER
 Core Port Implementation, 186
port_halt
 Core Port Implementation, 183
PORT_IDLE_THREAD_STACK_SIZE
 Core Port Implementation, 184
port_init
 Core Port Implementation, 186
PORT_INT_REQUIRED_STACK
 Core Port Implementation, 185
PORT_IRQ_EPILOGUE
 Core Port Implementation, 185
PORT_IRQ_HANDLER
 Core Port Implementation, 186
PORT_IRQ_PROLOGUE
 Core Port Implementation, 185
port_lock
 Core Port Implementation, 186
port_lock_from_isr
 Core Port Implementation, 186
port_suspend
 Core Port Implementation, 187
port_switch
 Core Port Implementation, 187
port_unlock
 Core Port Implementation, 186
port_unlock_from_isr
 Core Port Implementation, 187
port_wait_for_interrupt
 AT91SAM7 Specific Parameters, 191
 LPC214x Specific Parameters, 192
prio_insert
 Internals, 175

q_buffer
 GenericQueue, 219
q_counter
 GenericQueue, 219
Q_EMPTY
 I/O Queues, 156
Q_FULL
 I/O Queues, 156
q_notify
 GenericQueue, 219
Q_OK
 I/O Queues, 155
q_rdptr
 GenericQueue, 219
Q_RESET
 I/O Queues, 156
Q_TIMEOUT
 I/O Queues, 155
q_top
 GenericQueue, 219
q_waiting
 GenericQueue, 219

q_wptr
 GenericQueue, 219
qnotify_t
 I/O Queues, 162
queue_init
 Internals, 177
queue_insert
 Internals, 175

r_ctx
 ReadyList, 230
r_current
 ReadyList, 230
r_newer
 ReadyList, 230
r_older
 ReadyList, 230
r_preempt
 ReadyList, 230
r_prio
 ReadyList, 230
r_queue
 ReadyList, 230
RDY_OK
 Scheduler, 38
RDY_RESET
 Scheduler, 39
RDY_TIMEOUT
 Scheduler, 38
rdymsg
 Thread, 237
ReadyList, 228
 r_ctx, 230
 r_current, 230
 r_newer, 230
 r_older, 230
 r_preempt, 230
 r_prio, 230
 r_queue, 230
REG_INSERT
 Registry, 165
REG_REMOVE
 Registry, 165
regarm_t
 Core Port Implementation, 189
Registry, 162
 chRegFirstThread, 163
 chRegGetThreadName, 164
 chRegNextThread, 163
 chRegSetThreadName, 164
 REG_INSERT, 165
 REG_REMOVE, 165
rlist
 Scheduler, 38
ROMCONST
 Core Port Implementation, 188

S2ST
 Time and Virtual Timers, 59

s_cnt
 Semaphore, 233

s_queue
 Semaphore, 233

Scheduler, 33
 _scheduler_init, 34
 ABSPRIO, 39
 chSchCanYieldS, 40
 chSchDoReschedule, 38
 chSchDoYieldS, 40
 chSchGoSleepS, 35
 chSchGoSleepTimeoutS, 35
 chSchIsPreemptionRequired, 37
 chSchIsRescRequiredl, 40
 chSchPreemption, 40
 chSchReadyI, 34
 chSchRescheduleS, 37
 chSchWakeups, 36
 currp, 39
 firstprio, 39
 HIGHPRIO, 39
 IDLEPRIO, 39
 LOWPRIO, 39
 NOPRIO, 39
 NORMALPRIO, 39
 RDY_OK, 38
 RDY_RESET, 39
 RDY_TIMEOUT, 38
 rlist, 38
 setcurrp, 40
 TIME_IMMEDIATE, 39
 TIME_INFINITE, 39

se_state
 ch_swc_event_t, 209

se_time
 ch_swc_event_t, 209

se_tp
 ch_swc_event_t, 209

se_wtobjp
 ch_swc_event_t, 209

Semaphore, 230
 Counting Semaphores, 71
 s_cnt, 233
 s_queue, 233

SEMAPHORE_DECL
 Counting Semaphores, 70

setcurrp
 Scheduler, 40

SETUP_CONTEXT
 Core Port Implementation, 184

size
 heap_header, 220

Specific Implementations, 191

STACK_ALIGN
 Core Port Implementation, 185

Startup Support, 190

stkalign_t
 Core Port Implementation, 189

Synchronization, 61

Synchronous Messages, 105
 chMsgGet, 108
 chMsgGetS, 108
 chMsgIsPendingI, 107
 chMsgRelease, 107
 chMsgReleaseS, 108
 chMsgSend, 106
 chMsgWait, 106

System Management, 25
 _idle_thread, 28
 CH_FAST_IRQ_HANDLER, 32
 CH_IRQ_EPILOGUE, 32
 CH_IRQ_HANDLER, 32
 CH_IRQ_PROLOGUE, 31
 chSysDisable, 29
 chSysEnable, 30
 chSysGetIdleThread, 28
 chSysHalt, 28
 chSysInit, 27
 chSysLock, 30
 chSysLockFromIsr, 31
 chSysSuspend, 29
 chSysSwitch, 29
 chSysTimerHandlerI, 27
 chSysUnlock, 30
 chSysUnlockFromIsr, 31

SYSTEM_HALT_HOOK
 Configuration, 25

SYSTEM_TICK_EVENT_HOOK
 Configuration, 24

system_t
 Core Port Implementation, 189

tb_buffer
 ch_trace_buffer_t, 211

tb_ptr
 ch_trace_buffer_t, 211

tb_size
 ch_trace_buffer_t, 211

tfunc_t
 Threads, 55

THD_MEM_MODE_HEAP
 Threads, 53

THD_MEM_MODE_MASK
 Threads, 52

THD_MEM_MODE_MEMPOOL
 Threads, 53

THD_MEM_MODE_STATIC
 Threads, 52

THD_STATE_CURRENT
 Threads, 51

THD_STATE_FINAL
 Threads, 52

THD_STATE_NAMES
 Threads, 52

THD_STATE_READY
 Threads, 51

THD_STATE_SLEEPING
 Threads, 51

THD_STATE_SNDSMSG
 Threads, 52

THD_STATE_SNDSMSGQ
 Threads, 52

THD_STATE_SUSPENDED
 Threads, 51

THD_STATE_WTANDEVT
 Threads, 52

THD_STATE_WTCOND
 Threads, 51

THD_STATE_WTEXIT
 Threads, 52

THD_STATE_WTMSG
 Threads, 52

THD_STATE_WTMTX
 Threads, 51

THD_STATE_WTOREVT
 Threads, 52

THD_STATE_WTQUEUE
 Threads, 52

THD_STATE_WTSEM
 Threads, 51

THD_TERMINATE
 Threads, 53

THD_WA_SIZE
 Core Port Implementation, 185

Thread, 233

- ewmask, 238
- exitcode, 238
- p_ctx, 237
- p_epending, 238
- p_flags, 237
- p_mpool, 239
- p_msg, 238
- p_msgqueue, 238
- p_mtxlist, 238
- p_name, 237
- p_newer, 237
- p_next, 236
- p_older, 237
- p_prev, 236
- p_prio, 237
- p_realprio, 239
- p_refs, 237
- p_state, 237
- p_stklimit, 237
- p_time, 237
- p_waiting, 238
- rdymsg, 237
- wtobjp, 238

THREAD_CONTEXT_SWITCH_HOOK
 Configuration, 24

THREAD_EXT_EXIT_HOOK
 Configuration, 24

THREAD_EXT_FIELDS
 Configuration, 23

THREAD_EXT_INIT_HOOK
 Configuration, 24

Threads, 41

 _thread_init, 44

 _thread_memfill, 44

 chThdCreateI, 44

 chThdCreateStatic, 45

 chThdExit, 48

 chThdExitS, 49

 chThdGetPriority, 53

 chThdGetTicks, 53

 chThdLS, 53

 chThdResume, 46

 chThdResumel, 54

 chThdSelf, 53

 chThdSetPriority, 46

 chThdShouldTerminate, 54

 chThdSleep, 48

 chThdSleepMicroseconds, 55

 chThdSleepMilliseconds, 55

 chThdSleepS, 54

 chThdSleepSeconds, 55

 chThdSleepUntil, 48

 chThdTerminate, 47

 chThdTerminated, 54

 chThdWait, 50

 chThdYield, 48

 tfunc_t, 55

THD_MEM_MODE_HEAP, 53

THD_MEM_MODE_MASK, 52

THD_MEM_MODE_MEMPOOL, 53

THD_MEM_MODE_STATIC, 52

THD_STATE_CURRENT, 51

THD_STATE_FINAL, 52

THD_STATE_NAMES, 52

THD_STATE_READY, 51

THD_STATE_SLEEPING, 51

THD_STATE_SNDSMSG, 52

THD_STATE_SNDSMSGQ, 52

THD_STATE_SUSPENDED, 51

THD_STATE_WTANDEVT, 52

THD_STATE_WTCOND, 51

THD_STATE_WTEXIT, 52

THD_STATE_WTMSG, 52

THD_STATE_WTMTX, 51

THD_STATE_WTOREVT, 52

THD_STATE_WTQUEUE, 52

THD_STATE_WTSEM, 51

THD_TERMINATE, 53

ThreadsList, 239

- p_next, 242

ThreadsQueue, 242

- p_next, 245
- p_prev, 245

THREADSQUEUE_DECL
 Internals, 178

Time and Virtual Timers, 56

- _vt_init, 57
- chTimeIsWithin, 58
- chTimeNow, 60
- chVTDotickI, 60
- chVTIsArmedI, 60

chVTResetl, 58
chVTSell, 57
MS2ST, 59
S2ST, 59
US2ST, 59
VirtualTimer, 61
vfunc_t, 61
vtlist, 59
TIME_IMMEDIATE
 Scheduler, 39
TIME_INFINITE
 Scheduler, 39
tmode_t
 Core Port Implementation, 189
tprio_t
 Core Port Implementation, 189
trefs_t
 Core Port Implementation, 189
tstate_t
 Core Port Implementation, 189
Types, 25

u
 heap_header, 220

US2ST
 Time and Virtual Timers, 59

vectors.s, 286
Version Numbers and Identification, 14
 _CHIBIOS_RT_, 15
 _idle_thread, 15
 CH_KERNEL_MAJOR, 15
 CH_KERNEL_MINOR, 15
 CH_KERNEL_PATCH, 15
 CH_KERNEL_VERSION, 15

VirtualTimer, 245
 Time and Virtual Timers, 61
 vt_func, 246
 vt_next, 246
 vt_par, 246
 vt_prev, 246
 vt_time, 246

vmt
 BaseAsynchronousChannel, 195
 BaseChannel, 199
 BaseFileStream, 202
 BaseSequentialStream, 205

vt_func
 VirtualTimer, 246

vt_next
 VirtualTimer, 246
 VTLList, 247

vt_par
 VirtualTimer, 246

vt_prev
 VirtualTimer, 246
 VTLList, 247

vt_systime
 VTLList, 248

vt_time
 VirtualTimer, 246
 VTLList, 247

vfunc_t
 Time and Virtual Timers, 61

VTLList, 246
 vt_next, 247
 vt_prev, 247
 vt_systime, 248
 vt_time, 247

vtlist
 Time and Virtual Timers, 59

WORKING_AREA
 Core Port Implementation, 185

wtobjp
 Thread, 238