ChibiOS/RT

5.0.0

Reference Manual

Tue May 1 2018 09:42:56

Contents

Chapter 1

ChibiOS/RT

1.1 Copyright

1.2 Introduction

This document is the Reference Manual for the ChibiOS/RT portable Kernel.

1.3 Related Documents

- ChibiOS/RT General Architecture

Chapter 2

Kernel Concepts

ChibiOS/RT Kernel Concepts

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 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.

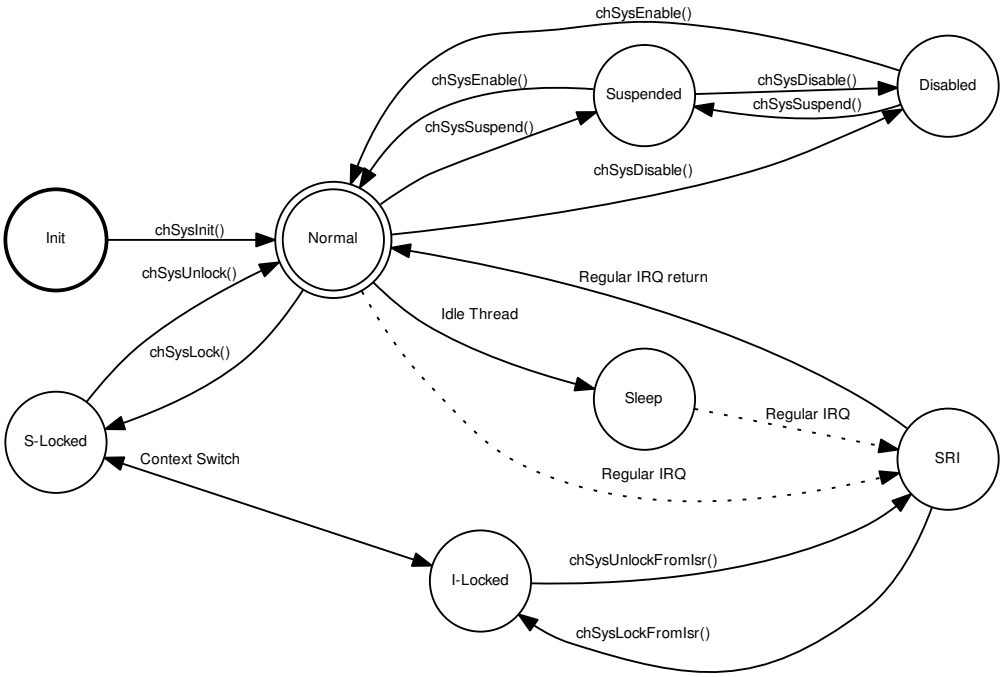
The mapping of the above logical classes into physical interrupts priorities is, of course, port dependent. See the documentation of the various ports for details.

2.4 System States

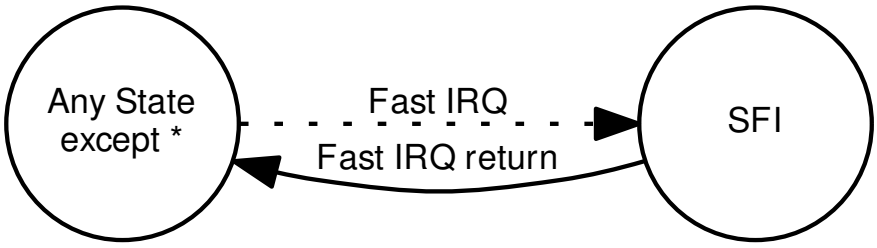
When using ChibiOS/RT the system can be in one of the following logical operating states:

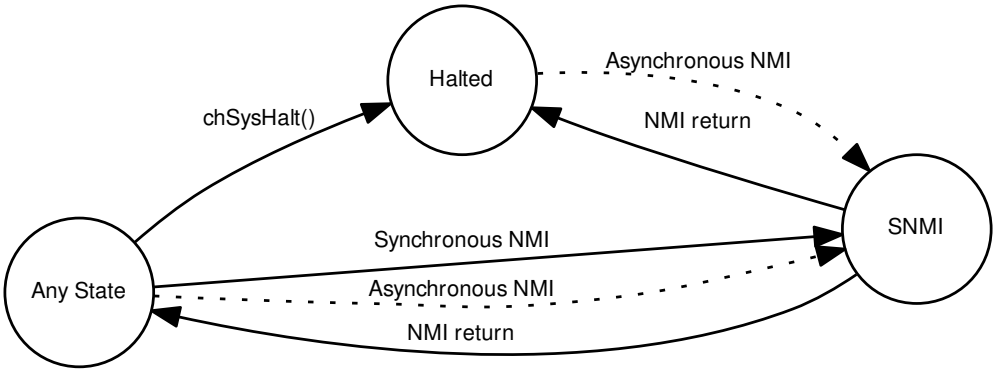
- **Init**. When the system is in this state all the maskable interrupt sources are disabled. In this state it is not possible to use any system API except `chSysInit()`. This state is entered after a physical reset.
- **Normal**. All the interrupt sources are enabled and the system APIs are accessible, 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:



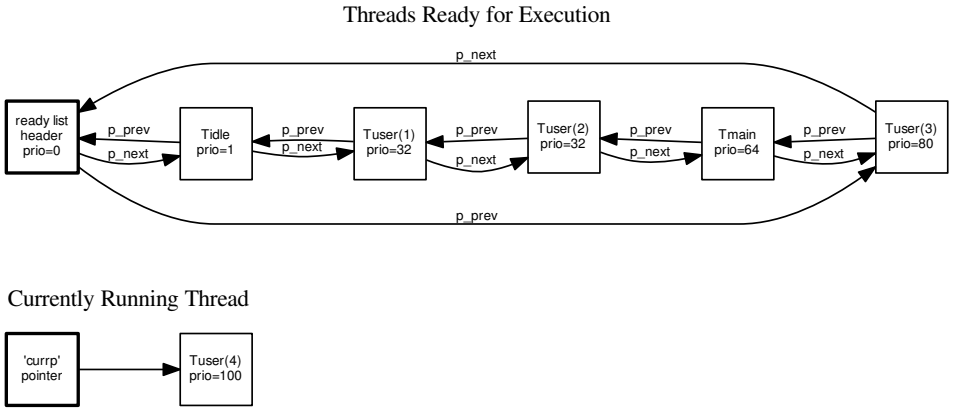


Attention

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

2.5 Scheduling

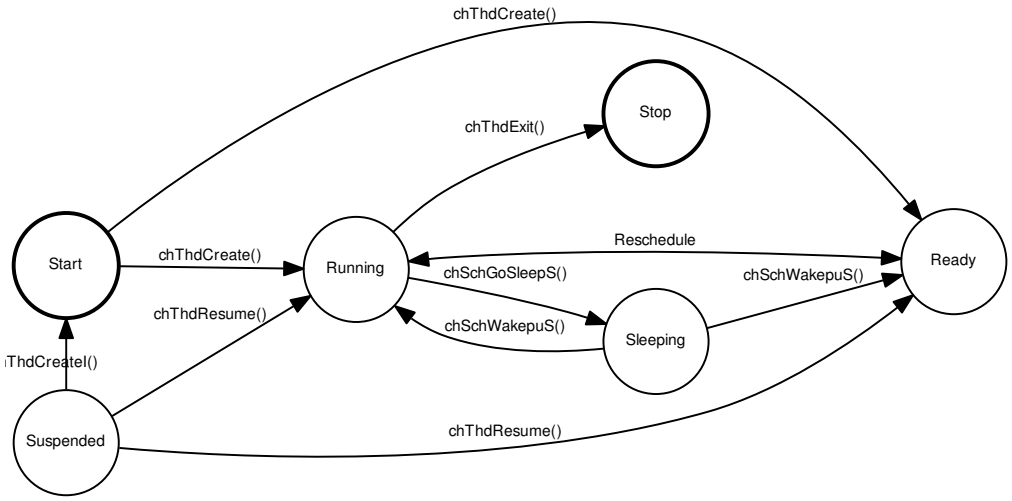
The strategy is very simple the currently ready thread with the highest priority is executed. If more than one thread with equal priority are eligible for execution then they are executed in a round-robin way, the CPU time slice constant is configurable. The ready list is a double linked list of threads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



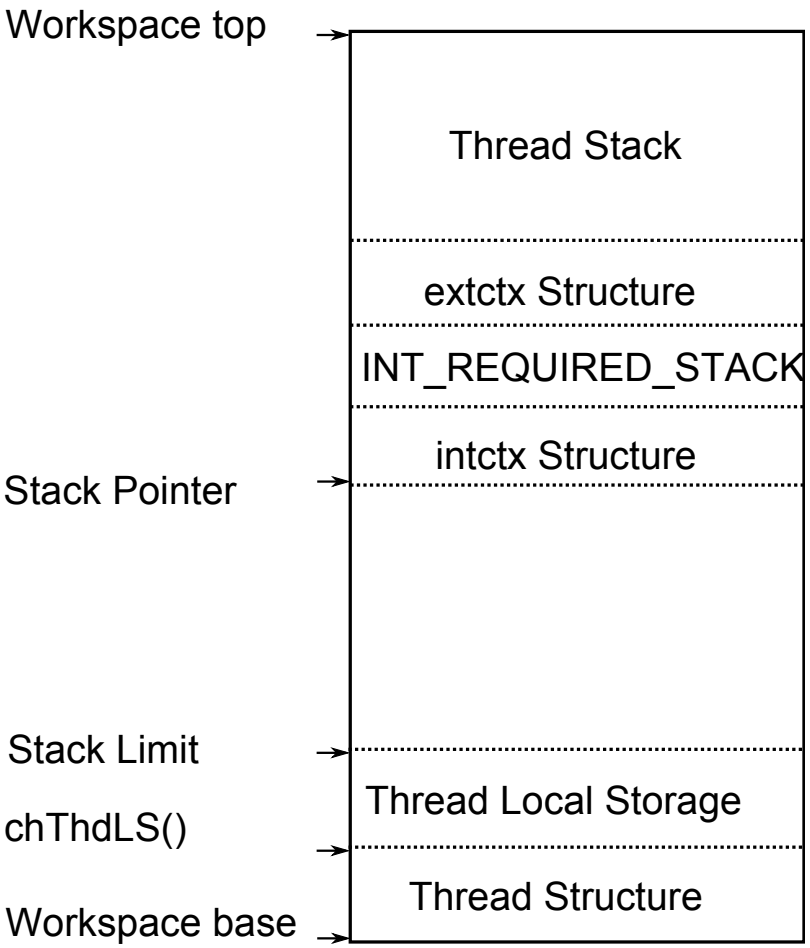
2.7 Priority Levels

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.
- `ABSPRO`, 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

Deprecated List

Global chMtxQueueNotEmptyS (mutex_t *mp)

Chapter 4

Module Index

4.1 Modules

Here is a list of all modules:

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

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

Chapter 6

Data Structure Index

6.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 7

File Index

7.1 File List

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

Chapter 8

Module Documentation

8.1 RT Kernel

8.1.1 Detailed Description

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

Modules

- Version Numbers and Identification
- Configuration
- License Checks
- Base Kernel Services
- Synchronization
- Memory Alignment
- Memory Management
- Registry
- Debug
- Trace
- Time Measurement
- Statistics
- Port Layer

8.2 Version Numbers and Identification

8.2.1 Detailed Description

Kernel related info.

Macros

- #define _CHIBIOS_RT_
  ChibiOS/RT identification macro.
- #define CH_KERNEL_STABLE 1
  Stable release flag.

ChibiOS/RT version identification

- #define CH_KERNEL_VERSION "5.0.0"
  Kernel version string.
- #define CH_KERNEL_MAJOR 5
  Kernel version major number.
- #define CH_KERNEL_MINOR 0
  Kernel version minor number.
- #define CH_KERNEL_PATCH 0
  Kernel version patch number.

Constants for configuration options

- #define FALSE 0
  Generic 'false' preprocessor boolean constant.
- #define TRUE 1
  Generic 'true' preprocessor boolean constant.

Functions

- void chSysHalt (const char ∗reason)
  Halts the system.

8.2.2 Macro Definition Documentation

8.2.2.1 #define _CHIBIOS_RT_

ChibiOS/RT identification macro.

8.2.2.2 #define CH_KERNEL_STABLE 1

Stable release flag.

8.2.2.3 #define CH_KERNEL_VERSION "5.0.0"

Kernel version string.

8.2.2.4 #define CH_KERNEL_MAJOR 5

Kernel version major number.

8.2.2.5 #define CH_KERNEL_MINOR 0

Kernel version minor number.

8.2.2.6 #define CH_KERNEL_PATCH 0

Kernel version patch number.

8.2.2.7 #define FALSE 0

Generic 'false' preprocessor boolean constant.

Note

It is meant to be used in configuration files as switch.

8.2.2.8 #define TRUE 1

Generic 'true' preprocessor boolean constant.

Note

It is meant to be used in configuration files as switch.

8.2.3 Function Documentation

8.2.3.1 void chSysHalt (const char * *reason*)

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.

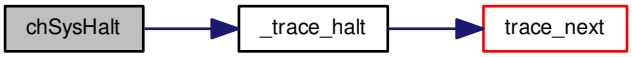
Parameters

in	*reason*	pointer to an error string

Function Class:

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

Here is the call graph for this function:



8.3 Configuration

8.3.1 Detailed Description

Kernel related settings and hooks.

System timers settings

- #define CH_CFG_ST_RESOLUTION 32

  System time counter resolution.
- #define CH_CFG_ST_FREQUENCY 10000

  System tick frequency.
- #define CH_CFG_INTERVALS_SIZE 32

  Time intervals data size.
- #define CH_CFG_TIME_TYPES_SIZE 32

  Time types data size.
- #define CH_CFG_ST_TIMEDELTA 2

  Time delta constant for the tick-less mode.

Kernel parameters and options

- #define CH_CFG_TIME_QUANTUM 0

  Round robin interval.
- #define CH_CFG_MEMCORE_SIZE 0

  Managed RAM size.
- #define CH_CFG_NO_IDLE_THREAD FALSE

  Idle thread automatic spawn suppression.

Performance options

- #define CH_CFG_OPTIMIZE_SPEED TRUE

  OS optimization.

Subsystem options

- #define CH_CFG_USE_TM TRUE

  Time Measurement APIs.
- #define CH_CFG_USE_REGISTRY TRUE

  Threads registry APIs.
- #define CH_CFG_USE_WAITEXIT TRUE

  Threads synchronization APIs.
- #define CH_CFG_USE_SEMAPHORES TRUE

  Semaphores APIs.
- #define CH_CFG_USE_SEMAPHORES_PRIORITY FALSE

  Semaphores queuing mode.
- #define CH_CFG_USE_MUTEXES TRUE

  Mutexes APIs.
- #define CH_CFG_USE_MUTEXES_RECURSIVE FALSE

  Enables recursive behavior on mutexes.
- #define CH_CFG_USE_CONDVARS TRUE

Conditional Variables APIs.

- #define CH_CFG_USE_CONDVARS_TIMEOUT TRUE

  Conditional Variables APIs with timeout.
- #define CH_CFG_USE_EVENTS TRUE

  Events Flags APIs.
- #define CH_CFG_USE_EVENTS_TIMEOUT TRUE

  Events Flags APIs with timeout.
- #define CH_CFG_USE_MESSAGES TRUE

  Synchronous Messages APIs.
- #define CH_CFG_USE_MESSAGES_PRIORITY FALSE

  Synchronous Messages queuing mode.
- #define CH_CFG_USE_MAILBOXES TRUE

  Mailboxes APIs.
- #define CH_CFG_USE_MEMCORE TRUE

  Core Memory Manager APIs.
- #define CH_CFG_USE_HEAP TRUE

  Heap Allocator APIs.
- #define CH_CFG_USE_MEMPOOLS TRUE

  Memory Pools Allocator APIs.
- #define CH_CFG_USE_OBJ_FIFOS TRUE

  Objects FIFOs APIs.
- #define CH_CFG_USE_DYNAMIC TRUE

  Dynamic Threads APIs.

Objects factory options

- #define CH_CFG_USE_FACTORY TRUE

  Objects Factory APIs.
- #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8

  Maximum length for object names.
- #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE

  Enables the registry of generic objects.
- #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE

  Enables factory for generic buffers.
- #define CH_CFG_FACTORY_SEMAPHORES TRUE

  Enables factory for semaphores.
- #define CH_CFG_FACTORY_MAILBOXES TRUE

  Enables factory for mailboxes.
- #define CH_CFG_FACTORY_OBJ_FIFOS TRUE

  Enables factory for objects FIFOs.

Debug options

- #define CH_DBG_STATISTICS FALSE

  Debug option, kernel statistics.
- #define CH_DBG_SYSTEM_STATE_CHECK TRUE

  Debug option, system state check.
- #define CH_DBG_ENABLE_CHECKS TRUE

  Debug option, parameters checks.
- #define CH_DBG_ENABLE_ASSERTS TRUE

Debug option, consistency checks.

- #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_ALL

  Debug option, trace buffer.
- #define CH_DBG_TRACE_BUFFER_SIZE 128

  Trace buffer entries.
- #define CH_DBG_ENABLE_STACK_CHECK TRUE

  Debug option, stack checks.
- #define CH_DBG_FILL_THREADS TRUE

  Debug option, stacks initialization.
- #define CH_DBG_THREADS_PROFILING FALSE

  Debug option, threads profiling.

Kernel hooks

- #define CH_CFG_SYSTEM_EXTRA_FIELDS /* Add threads custom fields here.*/

  System structure extension.
- #define CH_CFG_SYSTEM_INIT_HOOK(tp)

  System initialization hook.
- #define CH_CFG_THREAD_EXTRA_FIELDS /* Add threads custom fields here.*/

  Threads descriptor structure extension.
- #define CH_CFG_THREAD_INIT_HOOK(tp)

  Threads initialization hook.
- #define CH_CFG_THREAD_EXIT_HOOK(tp)

  Threads finalization hook.
- #define CH_CFG_CONTEXT_SWITCH_HOOK(ntp, otp)

  Context switch hook.
- #define CH_CFG_IRQ_PROLOGUE_HOOK()

  ISR enter hook.
- #define CH_CFG_IRQ_EPILOGUE_HOOK()

  ISR exit hook.
- #define CH_CFG_IDLE_ENTER_HOOK()

  Idle thread enter hook.
- #define CH_CFG_IDLE_LEAVE_HOOK()

  Idle thread leave hook.
- #define CH_CFG_IDLE_LOOP_HOOK()

  Idle Loop hook.
- #define CH_CFG_SYSTEM_TICK_HOOK()

  System tick event hook.
- #define CH_CFG_SYSTEM_HALT_HOOK(reason)

  System halt hook.
- #define CH_CFG_TRACE_HOOK(tep)

  Trace hook.

8.3.2 Macro Definition Documentation

8.3.2.1 #define CH_CFG_ST_RESOLUTION 32

System time counter resolution.

Note

Allowed values are 16 or 32 bits.

8.3.2.2 #define CH_CFG_ST_FREQUENCY 10000

System tick frequency.

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

8.3.2.3 #define CH_CFG_INTERVALS_SIZE 32

Time intervals data size.

Note

Allowed values are 16, 32 or 64 bits.

8.3.2.4 #define CH_CFG_TIME_TYPES_SIZE 32

Time types data size.

Note

Allowed values are 16 or 32 bits.

8.3.2.5 #define CH_CFG_ST_TIMEDELTA 2

Time delta constant for the tick-less mode.

Note

If this value is zero then the system uses the classic periodic tick. This value represents the minimum number of ticks that is safe to specify in a timeout directive. The value one is not valid, timeouts are rounded up to this value.

8.3.2.6 #define CH_CFG_TIME_QUANTUM 0

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.
The round robin preemption is not supported in tickless mode and must be set to zero in that case.

8.3.2.7 #define CH_CFG_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_CFG_USE_MEMCORE.

8.3.2.8 #define CH_CFG_NO_IDLE_THREAD FALSE

Idle thread automatic spawn suppression.

When this option is activated the function `chSysInit()` does not spawn the idle thread. The application `main()` function becomes the idle thread and must implement an infinite loop.

8.3.2.9 #define CH_CFG_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`.

8.3.2.10 #define CH_CFG_USE_TM TRUE

Time Measurement APIs.

If enabled then the time measurement APIs are included in the kernel.

Note

The default is `TRUE`.

8.3.2.11 #define CH_CFG_USE_REGISTRY TRUE

Threads registry APIs.

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

Note

The default is `TRUE`.

8.3.2.12 #define CH_CFG_USE_WAITEXIT TRUE

Threads synchronization APIs.

If enabled then the `chThdWait()` function is included in the kernel.

Note

The default is `TRUE`.

8.3.2.13 #define CH_CFG_USE_SEMAPHORES TRUE

Semaphores APIs.

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

Note

The default is `TRUE`.

8.3.2.14 #define CH_CFG_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_CFG_USE_SEMAPHORES`.

8.3.2.15 #define CH_CFG_USE_MUTEXES TRUE

Mutexes APIs.

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

Note

The default is `TRUE`.

8.3.2.16 #define CH_CFG_USE_MUTEXES_RECURSIVE FALSE

Enables recursive behavior on mutexes.

Note

Recursive mutexes are heavier and have an increased memory footprint.
The default is `FALSE`.
Requires `CH_CFG_USE_MUTEXES`.

8.3.2.17 #define CH_CFG_USE_CONDVARS TRUE

Conditional Variables APIs.

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

Note

The default is `TRUE`.
Requires `CH_CFG_USE_MUTEXES`.

8.3.2.18 #define CH_CFG_USE_CONDVARS_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_CFG_USE_CONDVARS`.

8.3.2.19 #define CH_CFG_USE_EVENTS TRUE

Events Flags APIs.

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

Note

The default is `TRUE`.

8.3.2.20 #define CH_CFG_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_CFG_USE_EVENTS`.

8.3.2.21 #define CH_CFG_USE_MESSAGES TRUE

Synchronous Messages APIs.

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

Note

The default is `TRUE`.

8.3.2.22 #define CH_CFG_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_CFG_USE_MESSAGES`.

8.3.2.23 #define CH_CFG_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_CFG_USE_SEMAPHORES`.

8.3.2.24 #define CH_CFG_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`.

8.3.2.25 #define CH_CFG_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_CFG_USE_MEMCORE` and either `CH_CFG_USE_MUTEXES` or `CH_CFG_USE_SEMAPHO↩RES`.
Mutexes are recommended.

8.3.2.26 #define CH_CFG_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`.

8.3.2.27 #define CH_CFG_USE_OBJ_FIFOS TRUE

Objects FIFOs APIs.

If enabled then the objects FIFOs APIs are included in the kernel.

Note

The default is `TRUE`.

8.3.2.28 #define CH_CFG_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_CFG_USE_WAITEXIT`.
Requires `CH_CFG_USE_HEAP` and/or `CH_CFG_USE_MEMPOOLS`.

8.3.2.29 #define CH_CFG_USE_FACTORY TRUE

Objects Factory APIs.

If enabled then the objects factory APIs are included in the kernel.

Note

The default is `FALSE`.

8.3.2.30 #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8

Maximum length for object names.

If the specified length is zero then the name is stored by pointer but this could have unintended side effects.

8.3.2.31 #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE

Enables the registry of generic objects.

8.3.2.32 #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE

Enables factory for generic buffers.

8.3.2.33 #define CH_CFG_FACTORY_SEMAPHORES TRUE

Enables factory for semaphores.

8.3.2.34 #define CH_CFG_FACTORY_MAILBOXES TRUE

Enables factory for mailboxes.

8.3.2.35 #define CH_CFG_FACTORY_OBJ_FIFOS TRUE

Enables factory for objects FIFOs.

8.3.2.36 #define CH_DBG_STATISTICS FALSE

Debug option, kernel statistics.

Note

The default is `FALSE`.

8.3.2.37 #define CH_DBG_SYSTEM_STATE_CHECK TRUE

Debug option, system state check.

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

Note

The default is `FALSE`.

8.3.2.38 #define CH_DBG_ENABLE_CHECKS TRUE

Debug option, parameters checks.

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

Note

The default is `FALSE`.

8.3.2.39 #define CH_DBG_ENABLE_ASSERTS TRUE

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`.

8.3.2.40 #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_ALL

Debug option, trace buffer.

If enabled then the trace buffer is activated.

Note

The default is `CH_DBG_TRACE_MASK_DISABLED`.

8.3.2.41 #define CH_DBG_TRACE_BUFFER_SIZE 128

Trace buffer entries.

Note

The trace buffer is only allocated if `CH_DBG_TRACE_MASK` is different from `CH_DBG_TRACE_MASK_D↩ISABLED`.

8.3.2.42 #define CH_DBG_ENABLE_STACK_CHECK TRUE

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`.

8.3.2.43 #define CH_DBG_FILL_THREADS TRUE

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`.

8.3.2.44 #define CH_DBG_THREADS_PROFILING FALSE

Debug option, threads profiling.

If enabled then a field is added to the `thread_t` structure that counts the system ticks occurred while executing the thread.

Note

The default is `FALSE`.
This debug option is not currently compatible with the tickless mode.

8.3.2.45 #define CH_CFG_SYSTEM_EXTRA_FIELDS /* Add threads custom fields here.*/

System structure extension.

User fields added to the end of the `ch_system_t` structure.

8.3.2.46 #define CH_CFG_SYSTEM_INIT_HOOK(*tp*)

Value:

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

System initialization hook.

User initialization code added to the `chSysInit()` function just before interrupts are enabled globally.

8.3.2.47 #define CH_CFG_THREAD_EXTRA_FIELDS /* Add threads custom fields here.*/

Threads descriptor structure extension.

User fields added to the end of the `thread_t` structure.

8.3.2.48 #define CH_CFG_THREAD_INIT_HOOK(*tp*)

Value:

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

Threads initialization hook.

User initialization code added to the `_thread_init()` function.

Note

It is invoked from within `_thread_init()` and implicitly from all the threads creation APIs.

8.3.2.49 #define CH_CFG_THREAD_EXIT_HOOK(*tp*)

Value:

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

Threads finalization hook.

User finalization code added to the `chThdExit()` API.

8.3.2.50 #define CH_CFG_CONTEXT_SWITCH_HOOK(*ntp*, *otp*)

Value:

```
{                                   \
  /* Context switch code here.*/                                        \
}
```

Context switch hook.

This hook is invoked just before switching between threads.

8.3.2.51 #define CH_CFG_IRQ_PROLOGUE_HOOK( )

Value:

```
{                                       \
  /* IRQ prologue code here.*/                                          \
}
```

ISR enter hook.

8.3.2.52 #define CH_CFG_IRQ_EPILOGUE_HOOK( )

Value:

```
{                                       \
  /* IRQ epilogue code here.*/                                          \
}
```

ISR exit hook.

8.3.2.53 #define CH_CFG_IDLE_ENTER_HOOK( )

Value:

```
{                                         \
  /* Idle-enter code here.*/                                            \
}
```

Idle thread enter hook.

Note

This hook is invoked within a critical zone, no OS functions should be invoked from here.
This macro can be used to activate a power saving mode.

8.3.2.54 #define CH_CFG_IDLE_LEAVE_HOOK( )

Value:

```
{                                                   \
  /* Idle-leave code here.*/                                    \
}
```

Idle thread leave hook.

Note

This hook is invoked within a critical zone, no OS functions should be invoked from here.
This macro can be used to deactivate a power saving mode.

8.3.2.55 #define CH_CFG_IDLE_LOOP_HOOK( )

Value:

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

Idle Loop hook.

This hook is continuously invoked by the idle thread loop.

8.3.2.56 #define CH_CFG_SYSTEM_TICK_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.

8.3.2.57 #define CH_CFG_SYSTEM_HALT_HOOK(*reason*)

Value:

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

System halt hook.

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

8.3.2.58 #define CH_CFG_TRACE_HOOK(*tep*)

Value:

```
{                                                    \
  /* Trace code here.*/                                         \
}
```

Trace hook.

This hook is invoked each time a new record is written in the trace buffer.

8.4 License Checks

8.5 Base Kernel Services

8.5.1 Detailed Description

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

Modules

- System Management
- Scheduler
- Threads
- Time and Virtual Timers

8.6 System Management

8.6.1 Detailed Description

System related APIs and services:

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

Macros

- #define chSysGetRealtimeCounterX() (rtcnt_t)port_rt_get_counter_value()

  Returns the current value of the system real time counter.
- #define chSysSwitch(ntp, otp)

  Performs a context switch.

Masks of executable integrity checks.

- #define **CH_INTEGRITY_RLIST** 1U
- #define **CH_INTEGRITY_VTLIST** 2U
- #define **CH_INTEGRITY_REGISTRY** 4U
- #define **CH_INTEGRITY_PORT** 8U

ISRs abstraction macros

- #define CH_IRQ_IS_VALID_PRIORITY(prio) PORT_IRQ_IS_VALID_PRIORITY(prio)

  Priority level validation macro.
- #define CH_IRQ_IS_VALID_KERNEL_PRIORITY(prio) PORT_IRQ_IS_VALID_KERNEL_PRIORITY(prio)

  Priority level validation macro.
- #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.

Time conversion utilities for the realtime counter

- #define S2RTC(freq, sec) ((freq) ∗ (sec))

  Seconds to realtime counter.
- #define MS2RTC(freq, msec) (rtcnt_t)((((freq) + 999UL) / 1000UL) ∗ (msec))

  Milliseconds to realtime counter.
- #define US2RTC(freq, usec) (rtcnt_t)((((freq) + 999999UL) / 1000000UL) ∗ (usec))

  Microseconds to realtime counter.
- #define RTC2S(freq, n) ((((n) - 1UL) / (freq)) + 1UL)

  Realtime counter cycles to seconds.
- #define RTC2MS(freq, n) ((((n) - 1UL) / ((freq) / 1000UL)) + 1UL)

  Realtime counter cycles to milliseconds.
- #define RTC2US(freq, n) ((((n) - 1UL) / ((freq) / 1000000UL)) + 1UL)

  Realtime counter cycles to microseconds.

Functions

- THD_WORKING_AREA (ch_idle_thread_wa, PORT_IDLE_THREAD_STACK_SIZE)

  Idle thread working area.
- static void _idle_thread (void ∗p)

  This function implements the idle thread infinite loop.
- void chSysInit (void)

  ChibiOS/RT initialization.
- void chSysHalt (const char ∗reason)

  Halts the system.
- bool chSysIntegrityCheckI (unsigned testmask)

  System integrity check.
- void chSysTimerHandlerI (void)

  Handles time ticks for round robin preemption and timer increments.
- syssts_t chSysGetStatusAndLockX (void)

  Returns the execution status and enters a critical zone.
- void chSysRestoreStatusX (syssts_t sts)

  Restores the specified execution status and leaves a critical zone.
- bool chSysIsCounterWithinX (rtcnt_t cnt, rtcnt_t start, rtcnt_t end)

  Realtime window test.
- void chSysPolledDelayX (rtcnt_t cycles)

  Polled delay.
- static void chSysDisable (void)

  Raises the system interrupt priority mask to the maximum level.
- static void chSysSuspend (void)

  Raises the system interrupt priority mask to system level.
- static void chSysEnable (void)

  Lowers the system interrupt priority mask to user level.
- static void chSysLock (void)

  Enters the kernel lock state.
- static void chSysUnlock (void)

  Leaves the kernel lock state.
- static void chSysLockFromISR (void)

  Enters the kernel lock state from within an interrupt handler.
- static void chSysUnlockFromISR (void)

  Leaves the kernel lock state from within an interrupt handler.

- static void chSysUnconditionalLock (void)

  Unconditionally enters the kernel lock state.
- static void chSysUnconditionalUnlock (void)

  Unconditionally leaves the kernel lock state.
- static thread_t * chSysGetIdleThreadX (void)

  Returns a pointer to the idle thread.

8.6.2 Macro Definition Documentation

8.6.2.1 #define CH_IRQ_IS_VALID_PRIORITY(*prio*) PORT_IRQ_IS_VALID_PRIORITY(prio)

Priority level validation macro.

This macro determines if the passed value is a valid priority level for the underlying architecture.

Parameters

in	*prio*	the priority level

Returns

Priority range result.

Return values

false	if the priority is invalid or if the architecture does not support priorities.
true	if the priority is valid.

8.6.2.2 #define CH_IRQ_IS_VALID_KERNEL_PRIORITY(*prio*) PORT_IRQ_IS_VALID_KERNEL_PRIORITY(prio)

Priority level validation macro.

This macro determines if the passed value is a valid priority level that cannot preempt the kernel critical zone.

Parameters

in	*prio*	the priority level

Returns

Priority range result.

Return values

false	if the priority is invalid or if the architecture does not support priorities.
true	if the priority is valid.

8.6.2.3 #define CH_IRQ_PROLOGUE( )

Value:

```
PORT_IRQ_PROLOGUE();                                                    \
  CH_CFG_IRQ_PROLOGUE_HOOK();
      \
  _stats_increase_irq();
      \
  _trace_isr_enter(__func__);
      \
  _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.

8.6.2.4 #define CH_IRQ_EPILOGUE( )

Value:

```
_dbg_check_leave_isr();
      \
  _trace_isr_leave(__func__);
      \
  CH_CFG_IRQ_EPILOGUE_HOOK();                                           \
  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.

8.6.2.5 #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.

8.6.2.6 #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.

8.6.2.7 #define S2RTC(*freq, sec*) ((freq) * (sec))

Seconds to realtime counter.

Converts from seconds to realtime counter cycles.

Note

The macro assumes that `freq` >= `1`.

Parameters

`in`	*freq*	clock frequency, in Hz, of the realtime counter
`in`	*sec*	number of seconds

Returns

The number of cycles.

Function Class:

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

8.6.2.8 #define MS2RTC(*freq, msec*) (rtcnt_t)((((freq) + 999UL) / 1000UL) * (msec))

Milliseconds to realtime counter.

Converts from milliseconds to realtime counter cycles.

Note

The result is rounded upward to the next millisecond boundary.
The macro assumes that `freq` >= `1000`.

Parameters

`in`	*freq*	clock frequency, in Hz, of the realtime counter
`in`	*msec*	number of milliseconds

Returns

The number of cycles.

Function Class:

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

8.6.2.9 #define US2RTC(*freq, usec*) (rtcnt_t)((((freq) + 999999UL) / 1000000UL) * (usec))

Microseconds to realtime counter.

Converts from microseconds to realtime counter cycles.

Note

The result is rounded upward to the next microsecond boundary.
The macro assumes that `freq >= 1000000`.

Parameters

`in`	*freq*	clock frequency, in Hz, of the realtime counter
`in`	*usec*	number of microseconds

Returns

The number of cycles.

Function Class:

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

8.6.2.10 #define RTC2S(*freq, n*) ((((n) - 1UL) / (freq)) + 1UL)

Realtime counter cycles to seconds.

Converts from realtime counter cycles number to seconds.

Note

The result is rounded up to the next second boundary.
The macro assumes that `freq >= 1`.

Parameters

`in`	*freq*	clock frequency, in Hz, of the realtime counter
`in`	*n*	number of cycles

Returns

The number of seconds.

Function Class:

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

8.6.2.11 #define RTC2MS(*freq, n*) ((((n) - 1UL) / ((freq) / 1000UL)) + 1UL)

Realtime counter cycles to milliseconds.

Converts from realtime counter cycles number to milliseconds.

Note

The result is rounded up to the next millisecond boundary.
The macro assumes that `freq` >= `1000`.

Parameters

in	*freq*	clock frequency, in Hz, of the realtime counter
in	*n*	number of cycles

Returns

The number of milliseconds.

Function Class:

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

8.6.2.12 #define RTC2US(*freq, n*) ((((n) - 1UL) / ((freq) / 1000000UL)) + 1UL)

Realtime counter cycles to microseconds.

Converts from realtime counter cycles number to microseconds.

Note

The result is rounded up to the next microsecond boundary.
The macro assumes that `freq` >= `1000000`.

Parameters

in	*freq*	clock frequency, in Hz, of the realtime counter
in	*n*	number of cycles

Returns

The number of microseconds.

Function Class:

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

8.6.2.13 #define chSysGetRealtimeCounterX( ) (rtcnt_t)port_rt_get_counter_value()

Returns the current value of the system real time counter.

Note

This function is only available if the port layer supports the option `PORT_SUPPORTS_RT`.

Returns

The value of the system realtime counter of type rtcnt_t.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.6.2.14 #define chSysSwitch(*ntp, otp*)

Value:

```
{                                                   \
                                                                    \
  _trace_switch(ntp, otp);
      \
  _stats_ctxswc(ntp, otp);
      \
  CH_CFG_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	*ntp*	the thread to be switched in
in	*otp*	the thread to be switched out

Function Class:

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

8.6.3 Function Documentation

8.6.3.1 THD_WORKING_AREA (ch_idle_thread_wa , PORT_IDLE_THREAD_STACK_SIZE)

Idle thread working area.

8.6.3.2 static void _idle_thread (void * *p*) [static]

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

8.6.3.3 void chSysInit (void)

ChibiOS/RT initialization.

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

Precondition

Interrupts must disabled before invoking this function.

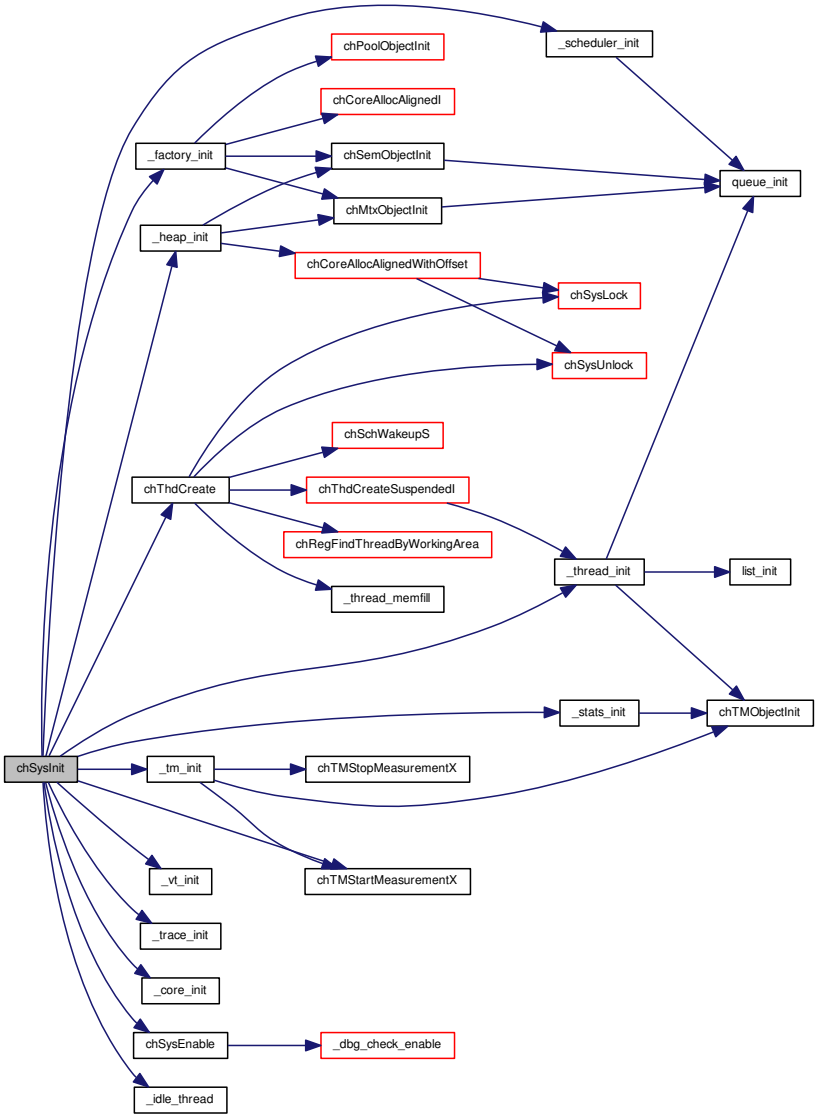
Postcondition

The main thread is created with priority `NORMALPRIO` and interrupts are enabled.

Function Class:

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

Here is the call graph for this function:



8.6.3.4 void chSysHalt (const char * *reason*)

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.

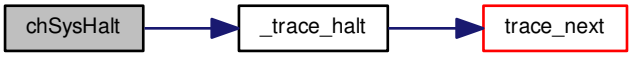
Parameters

in	*reason*	pointer to an error string

Function Class:

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

Here is the call graph for this function:



8.6.3.5 bool chSysIntegrityCheckI (unsigned *testmask*)

System integrity check.

Performs an integrity check of the important ChibiOS/RT data structures.

Note

The appropriate action in case of failure is to halt the system before releasing the critical zone.
If the system is corrupted then one possible outcome of this function is an exception caused by `NULL` or corrupted pointers in list elements. Exception vectors must be monitored as well.
This function is not used internally, it is up to the application to define if and where to perform system checking.
Performing all tests at once can be a slow operation and can degrade the system response time. It is suggested to execute one test at time and release the critical zone in between tests.

Parameters

in	*testmask*	Each bit in this mask is associated to a test to be performed.

Returns

The test result.

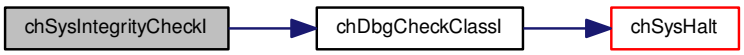
Return values

false	The test succeeded.
true	Test failed.

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:



8.6.3.6 void chSysTimerHandlerI (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.

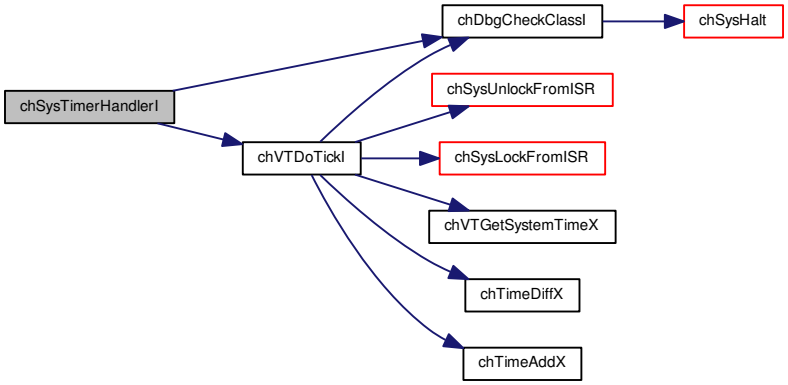
Note

The frequency of the timer determines the system tick granularity and, together with the `CH_CFG_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:



8.6.3.7 syssts_t chSysGetStatusAndLockX (void)

Returns the execution status and enters a critical zone.

This functions enters into a critical zone and can be called from any context. Because its flexibility it is less efficient than `chSysLock()` which is preferable when the calling context is known.

Postcondition

The system is in a critical zone.

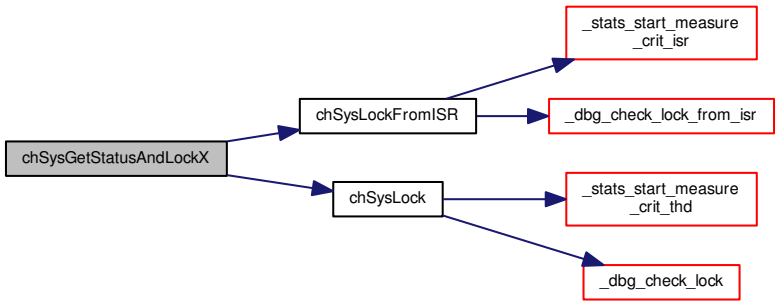
Returns

The previous system status, the encoding of this status word is architecture-dependent and opaque.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.6.3.8 void chSysRestoreStatusX (syssts_t *sts*)

Restores the specified execution status and leaves a critical zone.

Note

A call to `chSchRescheduleS()` is automatically performed if exiting the critical zone and if not in ISR context.

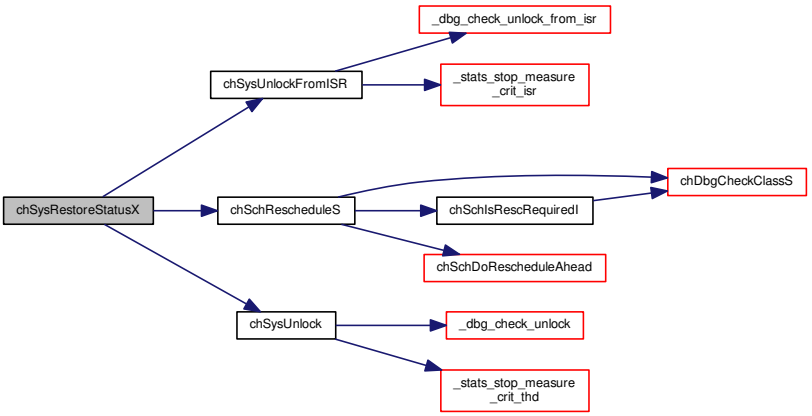
Parameters

in	*sts*	the system status to be restored.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.6.3.9 bool chSysIsCounterWithinX (rtcnt_t *cnt*, rtcnt_t *start*, rtcnt_t *end*)

Realtime window test.

This function verifies if the current realtime counter value lies within the specified range or not. The test takes care of the realtime counter wrapping to zero on overflow.

Note

When start==end then the function returns always true because the whole time range is specified.
This function is only available if the port layer supports the option `PORT_SUPPORTS_RT`.

Parameters

`in`	*cnt*	the counter value to be tested
`in`	*start*	the start of the time window (inclusive)
`in`	*end*	the end of the time window (non inclusive)

Return values

true	current time within the specified time window.
false	current time not within the specified time window.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.6.3.10 void chSysPolledDelayX (rtcnt_t *cycles*)

Polled delay.

Note

The real delay is always few cycles in excess of the specified value.
This function is only available if the port layer supports the option `PORT_SUPPORTS_RT`.

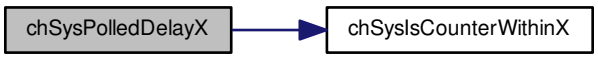
Parameters

`in`	*cycles*	number of cycles

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.6.3.11 static void chSysDisable (void) [inline], [static]

Raises the system interrupt priority mask to the maximum level.

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

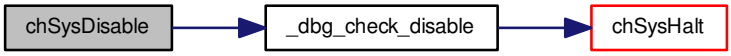
Note

Do not invoke this API from within a kernel lock.

Function Class:

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

Here is the call graph for this function:



8.6.3.12 static void chSysSuspend (void) [inline], [static]

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.

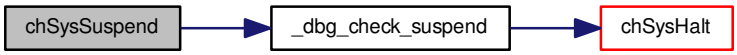
Note

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.

Here is the call graph for this function:



8.6.3.13 static void chSysEnable (void) [inline], [static]

Lowers the system interrupt priority mask to user level.

All the interrupt sources are enabled.

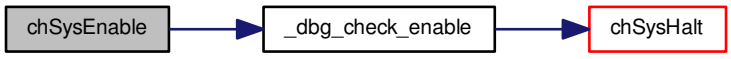
Note

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.

Here is the call graph for this function:



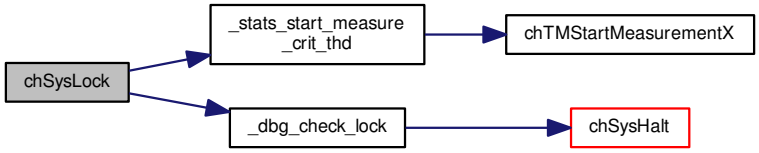
8.6.3.14 static void chSysLock (void) [inline], [static]

Enters the kernel lock state.

Function Class:

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

Here is the call graph for this function:



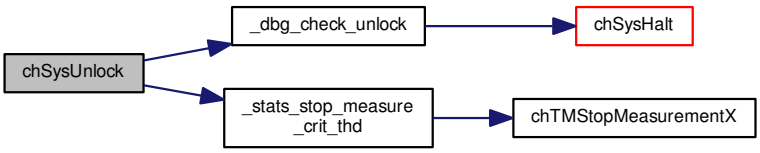
8.6.3.15 static void chSysUnlock (void) [inline], [static]

Leaves the kernel lock state.

Function Class:

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

Here is the call graph for this function:



8.6.3.16 static void chSysLockFromISR (void) [inline], [static]

Enters the kernel lock state from within an interrupt handler.

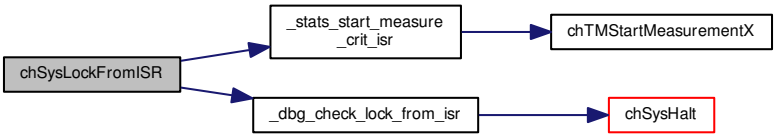
Note

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.

Here is the call graph for this function:



8.6.3.17 static void chSysUnlockFromISR (void) [inline], [static]

Leaves the kernel lock state from within an interrupt handler.

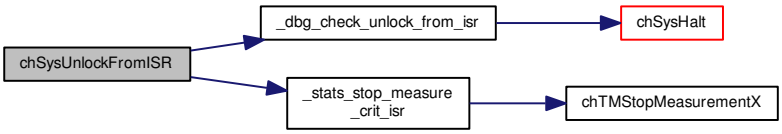
Note

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.

Here is the call graph for this function:



8.6.3.18 static void chSysUnconditionalLock (void) [inline], [static]

Unconditionally enters the kernel lock state.

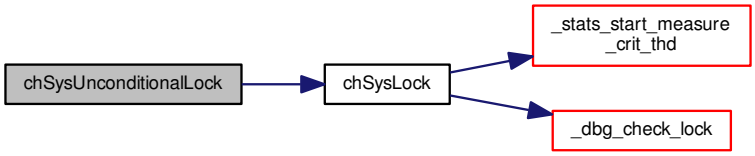
Note

Can be called without previous knowledge of the current lock state. The final state is "s-locked".

Function Class:

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

Here is the call graph for this function:



8.6.3.19 static void chSysUnconditionalUnlock (void) [inline], [static]

Unconditionally leaves the kernel lock state.

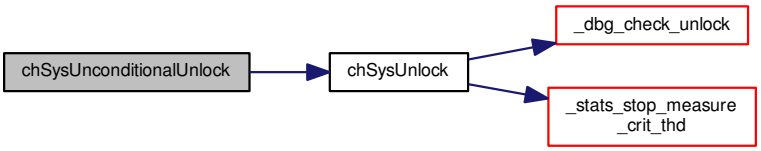
Note

Can be called without previous knowledge of the current lock state. The final state is "normal".

Function Class:

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

Here is the call graph for this function:



8.6.3.20 static thread_t* chSysGetIdleThreadX (void) [inline], [static]

Returns a pointer to the idle thread.

Precondition

In order to use this function the option `CH_CFG_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:

This is an **X-Class** API, this function can be invoked from any context.

8.7 Scheduler

8.7.1 Detailed Description

This module provides the default portable scheduler code.

Macros

- #define firstprio(rlp) ((rlp)->next->prio)

  Returns the priority of the first thread on the given ready list.
- #define currp ch.rlist.current

  Current thread pointer access macro.
- #define __CH_STRINGIFY(a) #a

  Utility to make the parameter a quoted string.

Wakeup status codes

- #define MSG_OK (msg_t)0

  Normal wakeup message.
- #define MSG_TIMEOUT (msg_t)-1

  Wakeup caused by a timeout condition.
- #define MSG_RESET (msg_t)-2

  Wakeup caused by a reset condition.

Priority constants

- #define NOPRIO (tprio_t)0

  Ready list header priority.
- #define IDLEPRIO (tprio_t)1

  Idle priority.
- #define LOWPRIO (tprio_t)2

  Lowest priority.
- #define NORMALPRIO (tprio_t)128

  Normal priority.
- #define HIGHPRIO (tprio_t)255

  Highest priority.

Thread states

- #define CH_STATE_READY (tstate_t)0

  Waiting on the ready list.
- #define CH_STATE_CURRENT (tstate_t)1

  Currently running.
- #define CH_STATE_WTSTART (tstate_t)2

  Just created.
- #define CH_STATE_SUSPENDED (tstate_t)3

  Suspended state.
- #define CH_STATE_QUEUED (tstate_t)4

  On an I/O queue.
- #define CH_STATE_WTSEM (tstate_t)5

On a semaphore.

- #define CH_STATE_WTMTX (tstate_t)6

  On a mutex.
- #define CH_STATE_WTCOND (tstate_t)7

  On a cond.variable.
- #define CH_STATE_SLEEPING (tstate_t)8

  Sleeping.
- #define CH_STATE_WTEXIT (tstate_t)9

  Waiting a thread.
- #define CH_STATE_WTOREVT (tstate_t)10

  One event.
- #define CH_STATE_WTANDEVT (tstate_t)11

  Several events.
- #define CH_STATE_SNDMSGQ (tstate_t)12

  Sending a message, in queue.
- #define CH_STATE_SNDMSG (tstate_t)13

  Sent a message, waiting answer.
- #define CH_STATE_WTMSG (tstate_t)14

  Waiting for a message.
- #define CH_STATE_FINAL (tstate_t)15

  Thread terminated.
- #define CH_STATE_NAMES

  Thread states as array of strings.

Thread flags and attributes

- #define CH_FLAG_MODE_MASK (tmode_t)3U

  Thread memory mode mask.
- #define CH_FLAG_MODE_STATIC (tmode_t)0U

  Static thread.
- #define CH_FLAG_MODE_HEAP (tmode_t)1U

  Thread allocated from a Memory Heap.
- #define CH_FLAG_MODE_MPOOL (tmode_t)2U

  Thread allocated from a Memory Pool.
- #define CH_FLAG_TERMINATE (tmode_t)4U

  Termination requested flag.

Typedefs

- typedef struct ch_thread thread_t

  Type of a thread structure.
- typedef thread_t * thread_reference_t

  Type of a thread reference.
- typedef struct ch_threads_list threads_list_t

  Type of a generic threads single link list, it works like a stack.
- typedef struct ch_threads_queue threads_queue_t

  Type of a generic threads bidirectional linked list header and element.
- typedef struct ch_ready_list ready_list_t

  Type of a ready list header.
- typedef void(* vtfunc_t) (void *p)

Type of a Virtual Timer callback function.

- typedef struct ch_virtual_timer virtual_timer_t

  Type of a Virtual Timer structure.
- typedef struct ch_virtual_timers_list virtual_timers_list_t

  Type of virtual timers list header.
- typedef struct ch_system_debug system_debug_t

  Type of a system debug structure.
- typedef struct ch_system ch_system_t

  Type of system data structure.

Data Structures

- struct ch_threads_list

  Generic threads single link list, it works like a stack.
- struct ch_threads_queue

  Generic threads bidirectional linked list header and element.
- struct ch_thread

  Structure representing a thread.
- struct ch_virtual_timer

  Virtual Timer descriptor structure.
- struct ch_virtual_timers_list

  Virtual timers list header.
- struct ch_system_debug

  System debug data structure.
- struct ch_system

  System data structure.

Functions

- void _scheduler_init (void)

  Scheduler initialization.
- void queue_prio_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a priority ordered queue.
- void queue_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a queue.
- thread_t * queue_fifo_remove (threads_queue_t *tqp)

  Removes the first-out thread from a queue and returns it.
- thread_t * queue_lifo_remove (threads_queue_t *tqp)

  Removes the last-out thread from a queue and returns it.
- thread_t * queue_dequeue (thread_t *tp)

  Removes a thread from a queue and returns it.
- void list_insert (thread_t *tp, threads_list_t *tlp)

  Pushes a thread_t on top of a stack list.
- thread_t * list_remove (threads_list_t *tlp)

  Pops a thread from the top of a stack list and returns it.
- thread_t * chSchReadyI (thread_t *tp)

  Inserts a thread in the Ready List placing it behind its peers.
- thread_t * chSchReadyAheadI (thread_t *tp)

  Inserts a thread in the Ready List placing it ahead its peers.
- void chSchGoSleepS (tstate_t newstate)

Puts the current thread to sleep into the specified state.

- msg_t chSchGoSleepTimeoutS (tstate_t newstate, sysinterval_t timeout)

  Puts the current thread to sleep into the specified state with timeout specification.
- void chSchWakeupS (thread_t *ntp, msg_t msg)

  Wakes up a thread.
- void chSchRescheduleS (void)

  Performs a reschedule if a higher priority thread is runnable.
- bool chSchIsPreemptionRequired (void)

  Evaluates if preemption is required.
- void chSchDoRescheduleBehind (void)

  Switches to the first thread on the runnable queue.
- void chSchDoRescheduleAhead (void)

  Switches to the first thread on the runnable queue.
- void chSchDoReschedule (void)

  Switches to the first thread on the runnable queue.
- static void list_init (threads_list_t *tlp)

  Threads list initialization.
- static bool list_isempty (threads_list_t *tlp)

  Evaluates to `true` if the specified threads list is empty.
- static bool list_notempty (threads_list_t *tlp)

  Evaluates to `true` if the specified threads list is not empty.
- static void queue_init (threads_queue_t *tqp)

  Threads queue initialization.
- static bool queue_isempty (const threads_queue_t *tqp)

  Evaluates to `true` if the specified threads queue is empty.
- static bool queue_notempty (const threads_queue_t *tqp)

  Evaluates to `true` if the specified threads queue is not empty.
- static bool chSchIsRescRequiredI (void)

  Determines if the current thread must reschedule.
- static bool chSchCanYieldS (void)

  Determines if yielding is possible.
- static void chSchDoYieldS (void)

  Yields the time slot.
- static void chSchPreemption (void)

  Inline-able preemption code.

Variables

- ch_system_t ch

  System data structures.

8.7.2 Macro Definition Documentation

8.7.2.1 #define MSG_OK (msg_t)0

Normal wakeup message.

8.7.2.2 #define MSG_TIMEOUT (msg_t)-1

Wakeup caused by a timeout condition.

8.7.2.3 #define MSG_RESET (msg_t)-2

Wakeup caused by a reset condition.

8.7.2.4 #define NOPRIO (tprio_t)0

Ready list header priority.

8.7.2.5 #define IDLEPRIO (tprio_t)1

Idle priority.

8.7.2.6 #define LOWPRIO (tprio_t)2

Lowest priority.

8.7.2.7 #define NORMALPRIO (tprio_t)128

Normal priority.

8.7.2.8 #define HIGHPRIO (tprio_t)255

Highest priority.

8.7.2.9 #define CH_STATE_READY (tstate_t)0

Waiting on the ready list.

8.7.2.10 #define CH_STATE_CURRENT (tstate_t)1

Currently running.

8.7.2.11 #define CH_STATE_WTSTART (tstate_t)2

Just created.

8.7.2.12 #define CH_STATE_SUSPENDED (tstate_t)3

Suspended state.

8.7.2.13 #define CH_STATE_QUEUED (tstate_t)4

On an I/O queue.

8.7.2.14 #define CH_STATE_WTSEM (tstate_t)5

On a semaphore.

8.7.2.15 #define CH_STATE_WTMTX (tstate_t)6

On a mutex.

8.7.2.16 #define CH_STATE_WTCOND (tstate_t)7

On a cond.variable.

8.7.2.17 #define CH_STATE_SLEEPING (tstate_t)8

Sleeping.

8.7.2.18 #define CH_STATE_WTEXIT (tstate_t)9

Waiting a thread.

8.7.2.19 #define CH_STATE_WTOREVT (tstate_t)10

One event.

8.7.2.20 #define CH_STATE_WTANDEVT (tstate_t)11

Several events.

8.7.2.21 #define CH_STATE_SNDMSGQ (tstate_t)12

Sending a message, in queue.

8.7.2.22 #define CH_STATE_SNDMSG (tstate_t)13

Sent a message, waiting answer.

8.7.2.23 #define CH_STATE_WTMSG (tstate_t)14

Waiting for a message.

8.7.2.24 #define CH_STATE_FINAL (tstate_t)15

Thread terminated.

8.7.2.25 #define CH_STATE_NAMES

Value:

```
"READY", "CURRENT", "WTSTART", "SUSPENDED", "QUEUED", "WTSEM", "WTMTX",  \
  "WTCOND", "SLEEPING", "WTEXIT", "WTOREVT", "WTANDEVT", "SNDMSGQ",       \
  "SNDMSG", "WTMSG", "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.

8.7.2.26 #define CH_FLAG_MODE_MASK (tmode_t)3U

Thread memory mode mask.

8.7.2.27 #define CH_FLAG_MODE_STATIC (tmode_t)0U

Static thread.

8.7.2.28 #define CH_FLAG_MODE_HEAP (tmode_t)1U

Thread allocated from a Memory Heap.

8.7.2.29 #define CH_FLAG_MODE_MPOOL (tmode_t)2U

Thread allocated from a Memory Pool.

8.7.2.30 #define CH_FLAG_TERMINATE (tmode_t)4U

Termination requested flag.

8.7.2.31 #define firstprio(*rlp*) ((rlp)->next->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.

8.7.2.32 #define currp ch.rlist.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 `chThdGetSelfX()` instead.

8.7.2.33 #define __CH_STRINGIFY(*a*) #a

Utility to make the parameter a quoted string.

8.7.3 Typedef Documentation

8.7.3.1 typedef struct ch_thread thread_t

Type of a thread structure.

8.7.3.2 typedef thread_t* thread_reference_t

Type of a thread reference.

8.7.3.3 typedef struct ch_threads_list threads_list_t

Type of a generic threads single link list, it works like a stack.

8.7.3.4 typedef struct ch_threads_queue threads_queue_t

Type of a generic threads bidirectional linked list header and element.

8.7.3.5 typedef struct ch_ready_list ready_list_t

Type of a ready list header.

8.7.3.6 typedef void(* vtfunc_t) (void *p)

Type of a Virtual Timer callback function.

8.7.3.7 typedef struct ch_virtual_timer virtual_timer_t

Type of a Virtual Timer structure.

8.7.3.8 typedef struct ch_virtual_timers_list virtual_timers_list_t

Type of virtual timers list header.

8.7.3.9 typedef struct ch_system_debug system_debug_t

Type of a system debug structure.

8.7.3.10 typedef struct ch_system ch_system_t

Type of system data structure.

8.7.4 Function Documentation

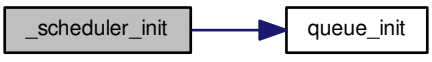
8.7.4.1 void _scheduler_init (void)

Scheduler initialization.

Function Class:

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

Here is the call graph for this function:



8.7.4.2 static void queue_prio_insert (thread_t * tp, threads_queue_t * tqp) [inline]

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	*tp*	the pointer to the thread to be inserted in the list
in	*tqp*	the pointer to the threads list header

Function Class:

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

8.7.4.3 static void queue_insert (thread_t * tp, threads_queue_t * tqp) [inline]

Inserts a thread into a queue.

Parameters

in	*tp*	the pointer to the thread to be inserted in the list
in	*tqp*	the pointer to the threads list header

Function Class:

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

8.7.4.4 static thread_t * queue_fifo_remove (threads_queue_t * tqp) [inline]

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.

8.7.4.5 static thread_t * queue_lifo_remove (threads_queue_t * tqp) [inline]

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.

8.7.4.6 static thread_t * queue_dequeue (thread_t * tp) [inline]

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.

8.7.4.7 static void list_insert (thread_t * tp, threads_list_t * tlp) [inline]

Pushes a thread_t 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.

8.7.4.8 static thread_t * list_remove (threads_list_t * *tlp*) [inline]

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.

8.7.4.9 thread_t * chSchReadyI (thread_t * *tp*)

Inserts a thread in the Ready List placing it behind its peers.

The thread is positioned behind all threads with higher or equal priority.

Precondition

The thread must not be already inserted in any list through its `next` and `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

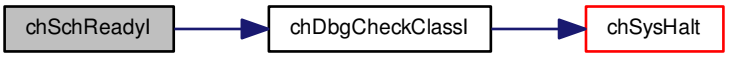
Returns

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.

Here is the call graph for this function:



8.7.4.10 thread_t * chSchReadyAheadI (thread_t * *tp*)

Inserts a thread in the Ready List placing it ahead its peers.

The thread is positioned ahead all threads with higher or equal priority.

Precondition

The thread must not be already inserted in any list through its `next` and `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

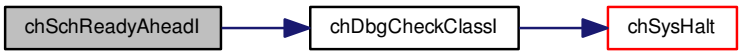
Returns

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.

Here is the call graph for this function:



8.7.4.11 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`.

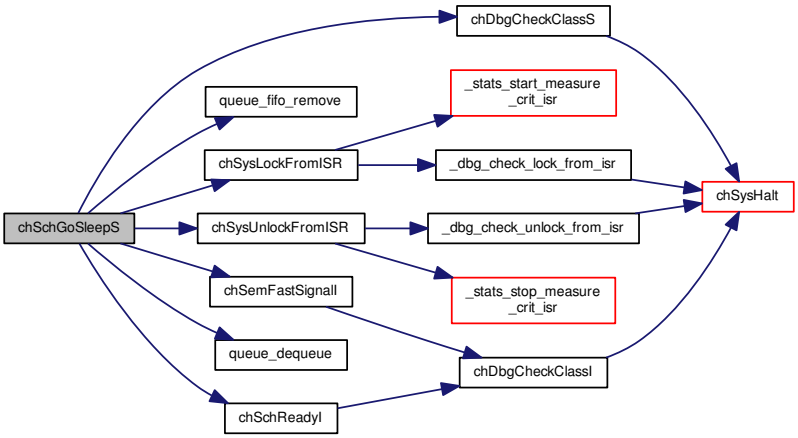
Parameters

`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:



8.7.4.12 msg_t chSchGoSleepTimeoutS (tstate_t *newstate*, sysinterval_t *timeout*)

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 `MSG_TIMEOUT` low level message. The possible Thread States are defined into `threads.h`.

Parameters

`in`	*newstate*	the new thread state
`in`	*timeout*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* the thread enters an infinite sleep state, this is equivalent to invoking `chSchGoSleepS()` but, of course, less efficient. • *TIME_IMMEDIATE* this value is not allowed.

Returns

The wakeup message.

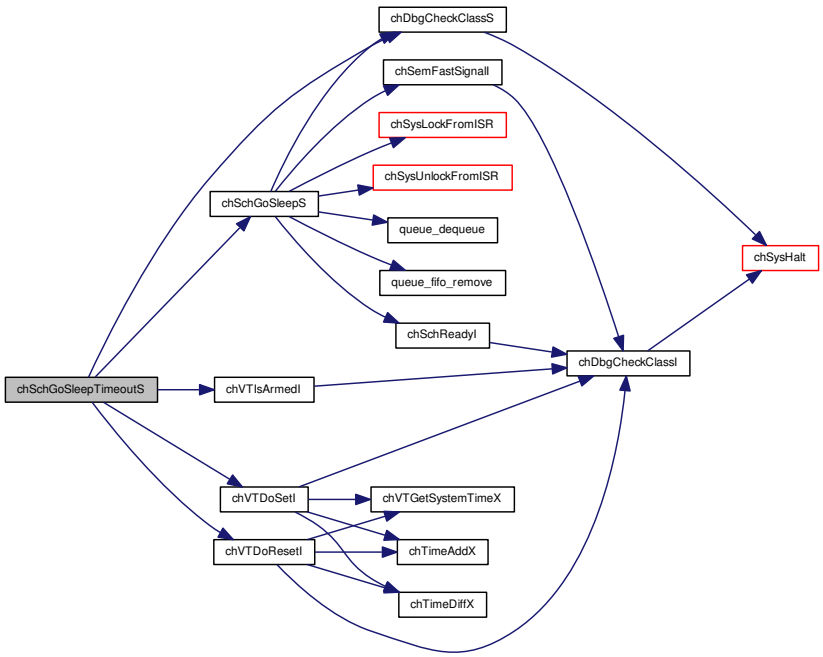
Return values

MSG_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:



8.7.4.13 void chSchWakeupS (thread_t * *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 `next` and `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.

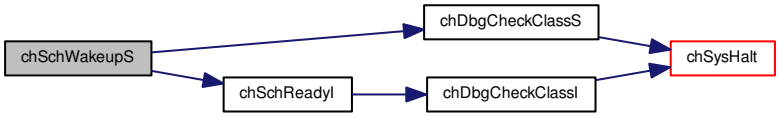
Parameters

`in`	*ntp*	the thread to be made ready
`in`	*msg*	the wakeup message

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:



8.7.4.14 void chSchRescheduleS (void)

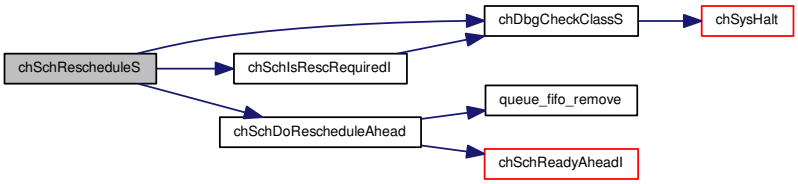
Performs a reschedule if a higher priority thread is runnable.

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 **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:



8.7.4.15 bool 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.

8.7.4.16 void chSchDoRescheduleBehind (void)

Switches to the first thread on the runnable queue.

The current thread is positioned in the ready list behind all threads having the same priority. The thread regains its time quantum.

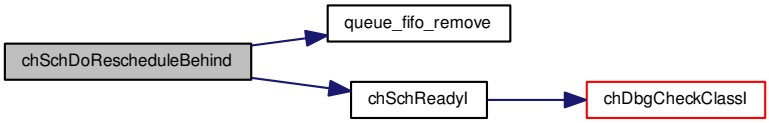
Note

Not a user function, it is meant to be invoked by the scheduler itself.

Function Class:

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

Here is the call graph for this function:



8.7.4.17 void chSchDoRescheduleAhead (void)

Switches to the first thread on the runnable queue.

The current thread is positioned in the ready list ahead of all threads having the same priority.

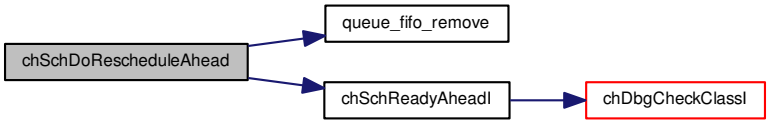
Note

Not a user function, it is meant to be invoked by the scheduler itself.

Function Class:

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

Here is the call graph for this function:



8.7.4.18 void chSchDoReschedule (void)

Switches to the first thread on the runnable queue.

The current thread is positioned in the ready list behind or ahead of all threads having the same priority depending on if it used its whole time slice.

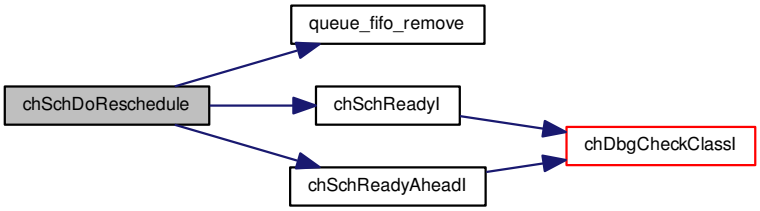
Note

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:



8.7.4.19 static void list_init (threads_list_t * *tlp*) [inline], [static]

Threads list initialization.

Parameters

in	*tlp*	pointer to the threads list object

Function Class:

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

8.7.4.20 static bool list_isempty (threads_list_t * *tlp*) [inline], [static]

Evaluates to `true` if the specified threads list is empty.

Parameters

in	*tlp*	pointer to the threads list object

Returns

The status of the list.

Function Class:

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

8.7.4.21 static bool list_notempty (threads_list_t * *tlp*) [inline], [static]

Evaluates to `true` if the specified threads list is not empty.

Parameters

in	*tlp*	pointer to the threads list object

Returns

The status of the list.

Function Class:

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

8.7.4.22 static void queue_init (threads_queue_t * *tqp*) [inline], [static]

Threads queue initialization.

Parameters

in	*tqp*	pointer to the threads queue object

Function Class:

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

8.7.4.23 static bool queue_isempty (const threads_queue_t * *tqp*) [inline], [static]

Evaluates to `true` if the specified threads queue is empty.

Parameters

in	*tqp*	pointer to the threads queue object

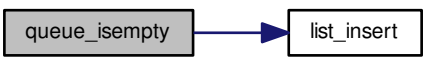
Returns

The status of the queue.

Function Class:

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

Here is the call graph for this function:



8.7.4.24 static bool queue_notempty (const threads_queue_t * *tqp*) [inline],[static]

Evaluates to `true` if the specified threads queue is not empty.

Parameters

`in`	*tqp*	pointer to the threads queue object

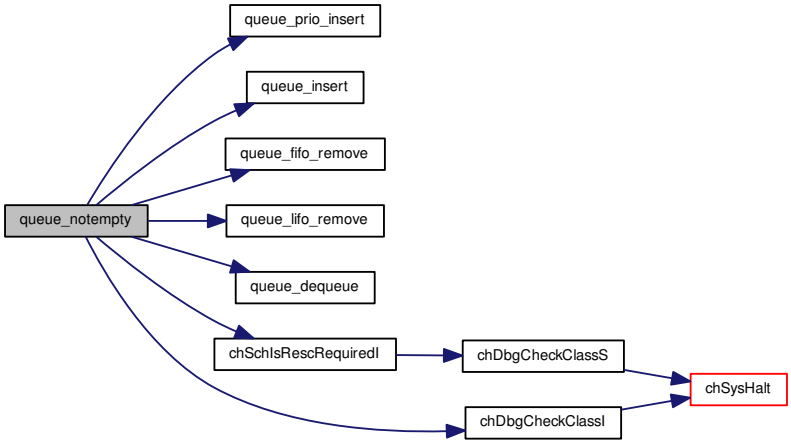
Returns

The status of the queue.

Function Class:

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

Here is the call graph for this function:



8.7.4.25 static bool chSchIsRescRequiredI (void) [inline],[static]

Determines if the current thread must reschedule.

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

Returns

The priorities situation.

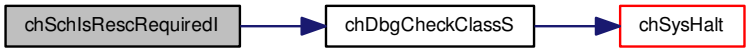
Return values

false	if rescheduling is not necessary.
true	if there is a ready thread at 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.

Here is the call graph for this function:



8.7.4.26 static bool chSchCanYieldS (void) [inline], [static]

Determines if yielding is possible.

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

Returns

The priorities situation.

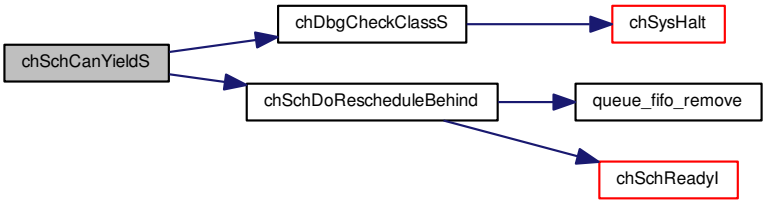
Return values

false	if yielding is not possible.
true	if there is a ready thread at 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.

Here is the call graph for this function:



8.7.4.27 static void chSchDoYieldS (void) [inline], [static]

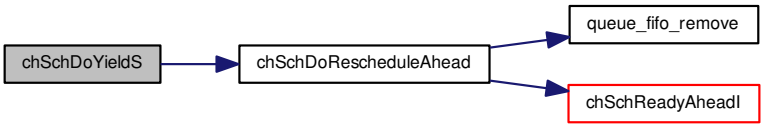
Yields the time slot.

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.

Here is the call graph for this function:



8.7.4.28 static void chSchPreemption (void) [inline], [static]

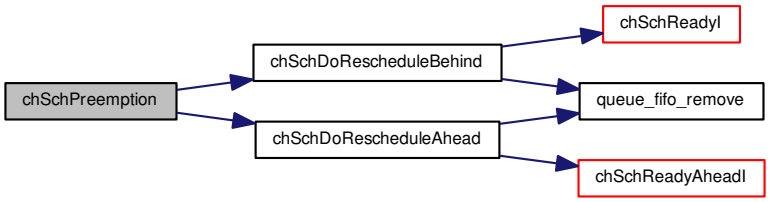
Inline-able preemption code.

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.

Here is the call graph for this function:



8.7.5 Variable Documentation

8.7.5.1 ch_system_t ch

System data structures.

8.8 Threads

8.8.1 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.

Threads queues

- #define _THREADS_QUEUE_DATA(name) {(thread_t *)&name, (thread_t *)&name}

  Data part of a static threads queue object initializer.
- #define _THREADS_QUEUE_DECL(name) threads_queue_t name = _THREADS_QUEUE_DATA(name)

  Static threads queue object initializer.

Working Areas

- #define THD_WORKING_AREA_SIZE(n) MEM_ALIGN_NEXT(sizeof(thread_t) + PORT_WA_SIZE(n), P↩ ORT_STACK_ALIGN)

  Calculates the total Working Area size.
- #define THD_WORKING_AREA(s, n) PORT_WORKING_AREA(s, n)

  Static working area allocation.
- #define THD_WORKING_AREA_BASE(s) ((stkalign_t *)(s))

  Base of a working area casted to the correct type.
- #define THD_WORKING_AREA_END(s)

  End of a working area casted to the correct type.

Threads abstraction macros

- #define THD_FUNCTION(tname, arg) PORT_THD_FUNCTION(tname, arg)

  Thread declaration macro.

Macro Functions

- #define chThdSleepSeconds(sec) chThdSleep(TIME_S2I(sec))

  Delays the invoking thread for the specified number of seconds.
- #define chThdSleepMilliseconds(msec) chThdSleep(TIME_MS2I(msec))

  Delays the invoking thread for the specified number of milliseconds.
- #define chThdSleepMicroseconds(usec) chThdSleep(TIME_US2I(usec))

  Delays the invoking thread for the specified number of microseconds.

Typedefs

- typedef void(∗ tfunc_t) (void ∗p)

  Thread function.

Data Structures

- struct thread_descriptor_t

  Type of a thread descriptor.

Functions

- thread_t ∗ _thread_init (thread_t ∗tp, const char ∗name, tprio_t prio)

  Initializes a thread structure.
- void _thread_memfill (uint8_t ∗startp, uint8_t ∗endp, uint8_t v)

  Memory fill utility.
- thread_t ∗ chThdCreateSuspendedI (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateSuspended (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateI (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreate (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateStatic (void ∗wsp, size_t size, tprio_t prio, tfunc_t pf, void ∗arg)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdStart (thread_t ∗tp)

  Resumes a thread created with `chThdCreateI()`.
- thread_t ∗ chThdAddRef (thread_t ∗tp)

  Adds a reference to a thread object.
- void chThdRelease (thread_t ∗tp)

  Releases a reference to a thread object.
- void chThdExit (msg_t msg)

  Terminates the current thread.
- void chThdExitS (msg_t msg)

  Terminates the current thread.
- msg_t chThdWait (thread_t ∗tp)

  Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.
- tprio_t chThdSetPriority (tprio_t newprio)

  Changes the running thread priority level then reschedules if necessary.
- void chThdTerminate (thread_t ∗tp)

Requests a thread termination.

- void chThdSleep (sysinterval_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.
- systime_t chThdSleepUntilWindowed (systime_t prev, systime_t next)

  Suspends the invoking thread until the system time arrives to the specified value.
- void chThdYield (void)

  Yields the time slot.
- msg_t chThdSuspendS (thread_reference_t *trp)

  Sends the current thread sleeping and sets a reference variable.
- msg_t chThdSuspendTimeoutS (thread_reference_t *trp, sysinterval_t timeout)

  Sends the current thread sleeping and sets a reference variable.
- void chThdResumeI (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- void chThdResumeS (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- void chThdResume (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- msg_t chThdEnqueueTimeoutS (threads_queue_t *tqp, sysinterval_t timeout)

  Enqueues the caller thread on a threads queue object.
- void chThdDequeueNextI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up one thread from the threads queue object, if any.
- void chThdDequeueAllI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up all threads from the threads queue object.
- static thread_t * chThdGetSelfX (void)

  Returns a pointer to the current `thread_t`.
- static tprio_t chThdGetPriorityX (void)

  Returns the current thread priority.
- static systime_t chThdGetTicksX (thread_t *tp)

  Returns the number of ticks consumed by the specified thread.
- static stkalign_t * chThdGetWorkingAreaX (thread_t *tp)

  Returns the working area base of the specified thread.
- static bool chThdTerminatedX (thread_t *tp)

  Verifies if the specified thread is in the `CH_STATE_FINAL` state.
- static bool chThdShouldTerminateX (void)

  Verifies if the current thread has a termination request pending.
- static thread_t * chThdStartI (thread_t *tp)

  Resumes a thread created with `chThdCreateI()`.
- static void chThdSleepS (sysinterval_t ticks)

  Suspends the invoking thread for the specified number of ticks.
- static void chThdQueueObjectInit (threads_queue_t *tqp)

  Initializes a threads queue object.
- static bool chThdQueueIsEmptyI (threads_queue_t *tqp)

  Evaluates to `true` if the specified queue is empty.
- static void chThdDoDequeueNextI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up one thread from the threads queue object.

8.8.2 Macro Definition Documentation

8.8.2.1 #define _THREADS_QUEUE_DATA(*name*) {(thread_t *)&name, (thread_t *)&name}

Data part of a static threads queue object 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

8.8.2.2 #define _THREADS_QUEUE_DECL(*name*) threads_queue_t name = _THREADS_QUEUE_DATA(name)

Static threads queue object initializer.

Statically initialized threads queues require no explicit initialization using `queue_init()`.

Parameters

in	*name*	the name of the threads queue variable

8.8.2.3 #define THD_WORKING_AREA_SIZE(*n*) MEM_ALIGN_NEXT(sizeof(thread_t) + PORT_WA_SIZE(n), PORT_STACK_ALIGN)

Calculates the total Working Area size.

Parameters

in	*n*	the stack size to be assigned to the thread

Returns

The total used memory in bytes.

Function Class:

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

8.8.2.4 #define THD_WORKING_AREA(*s, n*) PORT_WORKING_AREA(s, n)

Static working area allocation.

This macro is used to allocate a static thread working area aligned as both position and size.

Parameters

in	*s*	the name to be assigned to the stack array
in	*n*	the stack size to be assigned to the thread

Function Class:

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

8.8.2.5 #define THD_WORKING_AREA_BASE(*s*) ((stkalign_t *)(s))

Base of a working area casted to the correct type.

Parameters

in	*s*	name of the working area

8.8.2.6 #define THD_WORKING_AREA_END(*s*)

Value:

```
(THD_WORKING_AREA_BASE(s) +                 \
                                    (sizeof (s) / sizeof (stkalign_t)))
```

End of a working area casted to the correct type.

Parameters

in	*s*	name of the working area

8.8.2.7 #define THD_FUNCTION(*tname, arg*) PORT_THD_FUNCTION(tname, arg)

Thread declaration macro.

Note

Thread declarations should be performed using this macro because the port layer could define optimizations for thread functions.

8.8.2.8 #define chThdSleepSeconds(*sec*) chThdSleep(TIME_S2I(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 tick clock.
The maximum specifiable value is implementation dependent.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

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.

8.8.2.9 #define chThdSleepMilliseconds(*msec*) chThdSleep(TIME_MS2I(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 tick clock.
The maximum specifiable value is implementation dependent.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

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.

8.8.2.10 #define chThdSleepMicroseconds(*usec*) chThdSleep(TIME_US2I(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 tick clock.
The maximum specifiable value is implementation dependent.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

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.

8.8.3 Typedef Documentation

8.8.3.1 typedef void(* tfunc_t) (void *p)

Thread function.

8.8.4 Function Documentation

8.8.4.1 thread_t * _thread_init (thread_t * *tp,* const char * *name,* tprio_t *prio*)

Initializes a thread structure.

Note

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

Parameters

in	*tp*	pointer to the thread
in	*name*	thread name
in	*prio*	the priority level for the new thread

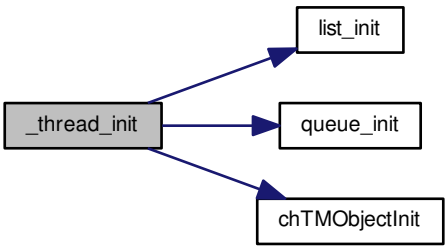
Returns

The same thread pointer passed as parameter.

Function Class:

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

Here is the call graph for this function:



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

Memory fill utility.

Parameters

in	*startp*	first address to fill
in	*endp*	last address to fill +1
in	*v*	filler value

Function Class:

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

8.8.4.3 thread_t * chThdCreateSuspendedI (const thread_descriptor_t * *tdp*)

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 `CH_STATE_WTSTART`.

Postcondition

The created thread has a reference counter set to one, it is caller responsibility to call `chThdRelease()` or `chthdWait()` in order to release the reference. The thread persists in the registry until its reference counter reaches zero.
The initialized thread can be subsequently started by invoking `chThdStart()`, `chThdStartI()` 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`	*tdp*	pointer to the thread descriptor

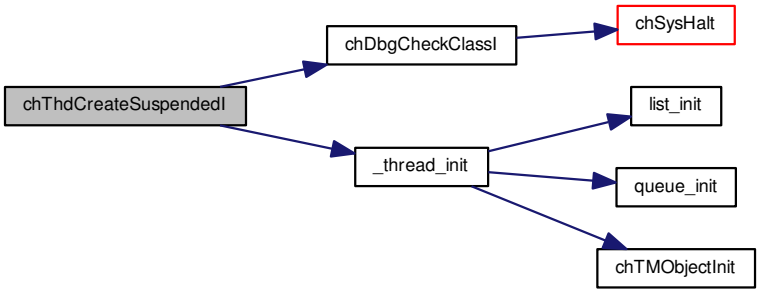
Returns

The pointer to the `thread_t` 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:



8.8.4.4 thread_t * chThdCreateSuspended (const thread_descriptor_t * *tdp*)

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 `CH_STATE_WTSTART`.

Postcondition

The created thread has a reference counter set to one, it is caller responsibility to call chThdRelease() or chthdWait() in order to release the reference. The thread persists in the registry until its reference counter reaches zero.
The initialized thread can be subsequently started by invoking chThdStart(), chThdStartI() or chSchWakeupS() depending on the execution context.

Note

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

Parameters

out	*tdp*	pointer to the thread descriptor

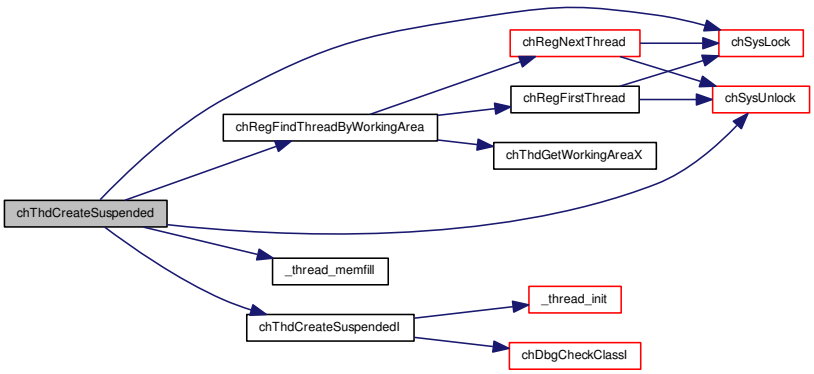
Returns

The pointer to the thread_t 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:



8.8.4.5 thread_t * chThdCreateI (const thread_descriptor_t * *tdp*)

Creates a new thread into a static memory area.

The new thread is initialized and make ready to execute.

Postcondition

The created thread has a reference counter set to one, it is caller responsibility to call chThdRelease() or chthdWait() in order to release the reference. The thread persists in the registry until its reference counter reaches zero.
The initialized thread can be subsequently started by invoking chThdStart(), chThdStartI() 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`	*tdp*	pointer to the thread descriptor

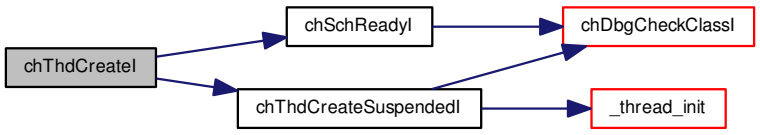
Returns

The pointer to the `thread_t` 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:



8.8.4.6 thread_t * chThdCreate (const thread_descriptor_t * *tdp*)

Creates a new thread into a static memory area.

The new thread is initialized and make ready to execute.

Postcondition

The created thread has a reference counter set to one, it is caller responsibility to call `chThdRelease()` or `chthdWait()` in order to release the reference. The thread persists in the registry until its reference counter reaches zero.

Note

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

Parameters

`out`	*tdp*	pointer to the thread descriptor

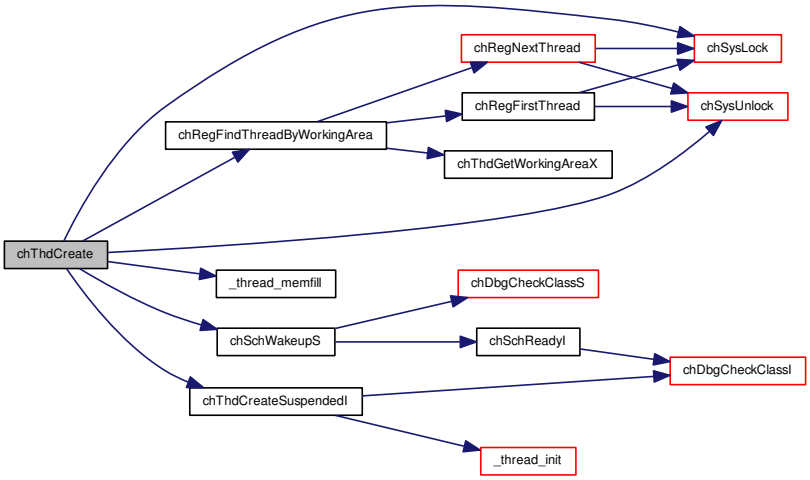
Returns

The pointer to the `thread_t` 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:



8.8.4.7 thread_t * chThdCreateStatic (void * *wsp*, size_t *size*, tprio_t *prio*, tfunc_t *pf*, void * *arg*)

Creates a new thread into a static memory area.

Postcondition

The created thread has a reference counter set to one, it is caller responsibility to call `chThdRelease()` or `chthdWait()` in order to release the reference. The thread persists in the registry until its reference counter reaches zero.

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`.

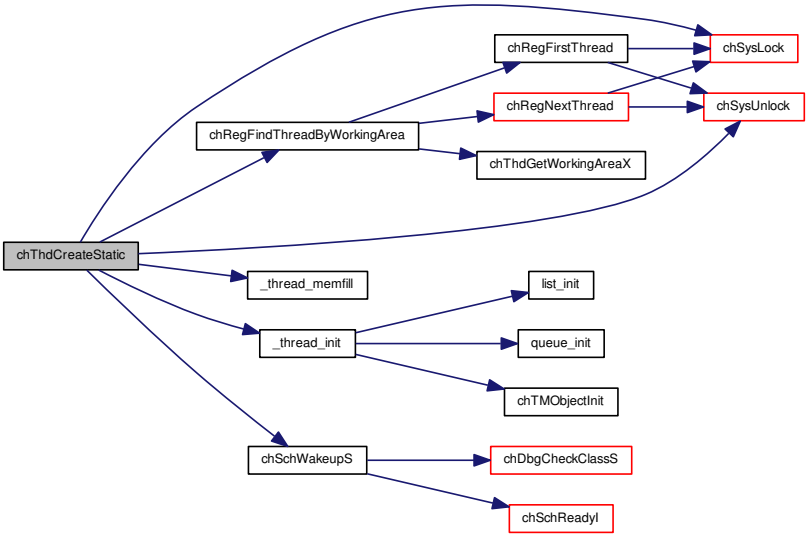
Returns

The pointer to the `thread_t` 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:



8.8.4.8 thread_t * chThdStart (thread_t * *tp*)

Resumes a thread created with `chThdCreateI()`.

Parameters

`in`	*tp*	pointer to the thread

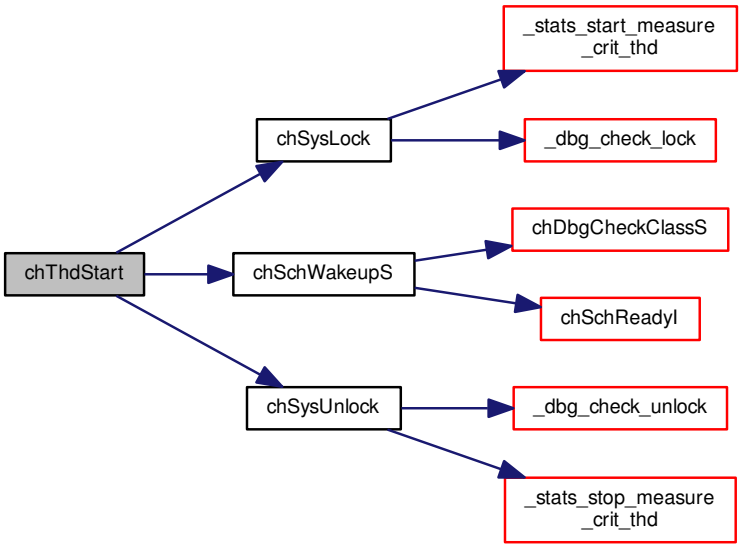
Returns

The pointer to the `thread_t` 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:



8.8.4.9 thread_t * chThdAddRef (thread_t * *tp*)

Adds a reference to a thread object.

Precondition

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

Parameters

`in`	*tp*	pointer to the thread

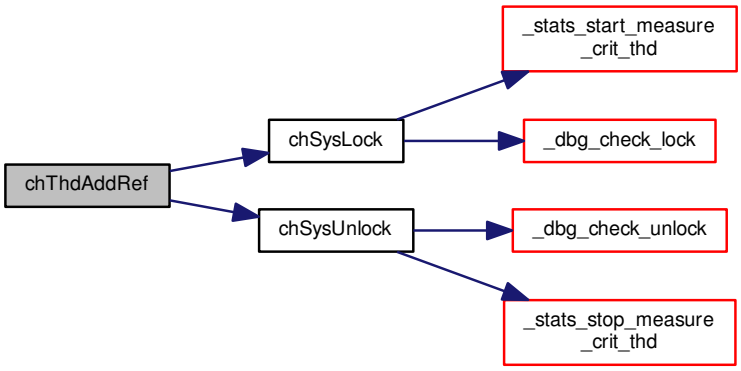
Returns

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.

Here is the call graph for this function:



8.8.4.10 void chThdRelease (thread_t * *tp*)

Releases a reference to a thread object.

If the references counter reaches zero **and** the thread is in the `CH_STATE_FINAL` state then the thread's memory is returned to the proper allocator and the thread is removed from the registry.
Threads whose counter reaches zero and are still active become "detached" and will be removed from registry on termination.

Precondition

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

Note

Static threads are not affected.

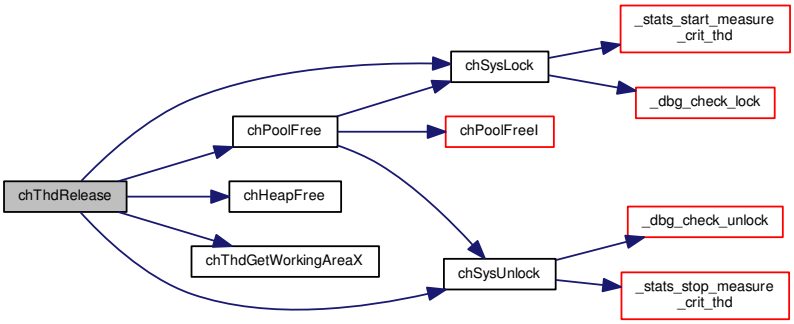
Parameters

`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:



8.8.4.11 void chThdExit (msg_t *msg*)

Terminates the current thread.

The thread goes in the `CH_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.

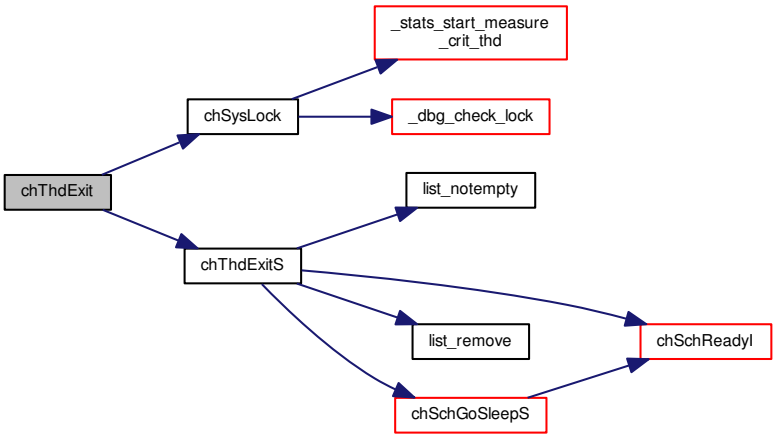
Parameters

`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:



8.8.4.12 void chThdExitS (msg_t *msg*)

Terminates the current thread.

The thread goes in the `CH_STATE_FINAL` state holding the specified exit status code, other threads can retrieve the exit status code by invoking the function `chThdWait()`.

Postcondition

Exiting a non-static thread that does not have references (detached) causes the thread to remain in the registry. It can only be removed by performing a registry scan operation.
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.

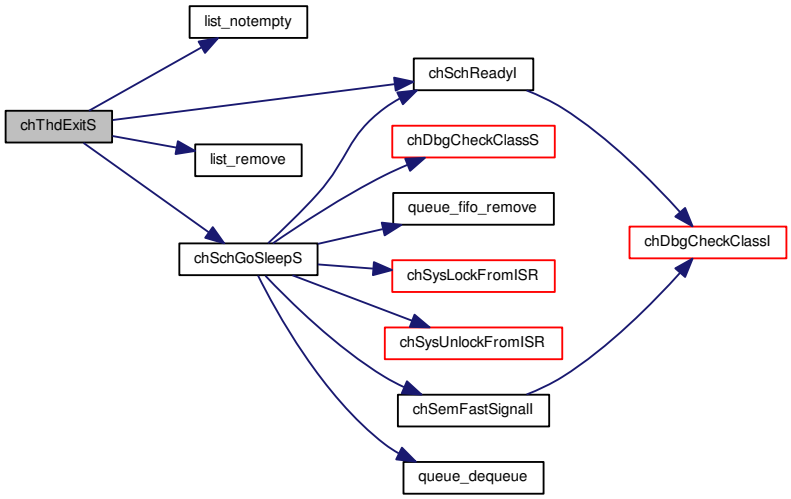
Parameters

`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:



8.8.4.13 msg_t chThdWait (thread_t * *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 and the thread is removed from registry.

Precondition

The configuration option `CH_CFG_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_t` structure.

Note

If `CH_CFG_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

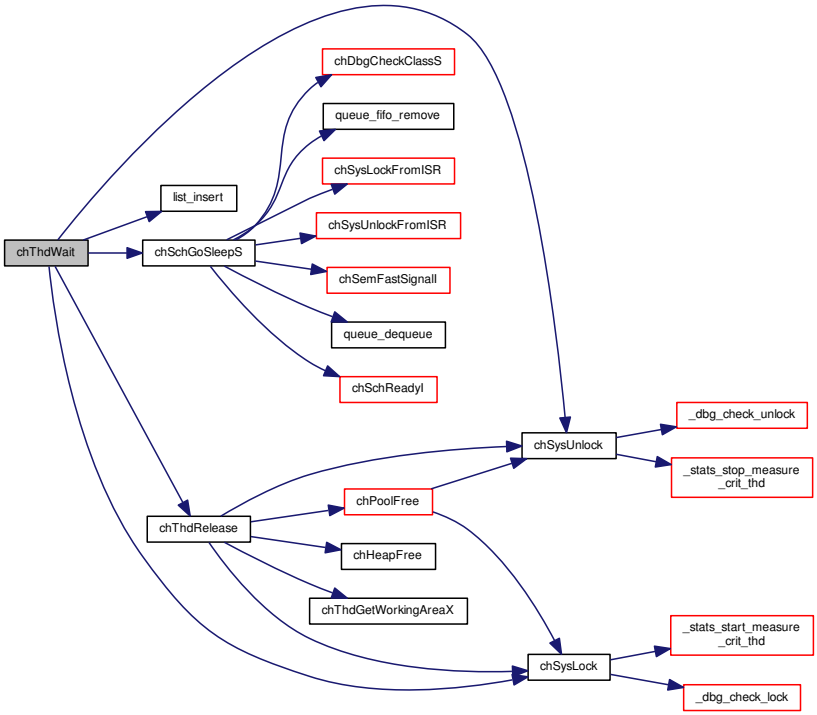
Returns

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:



8.8.4.14 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

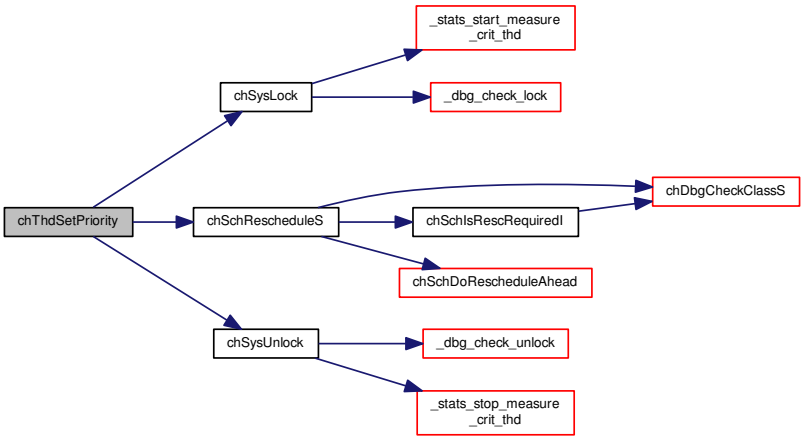
Returns

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:



8.8.4.15 void chThdTerminate (thread_t * *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.

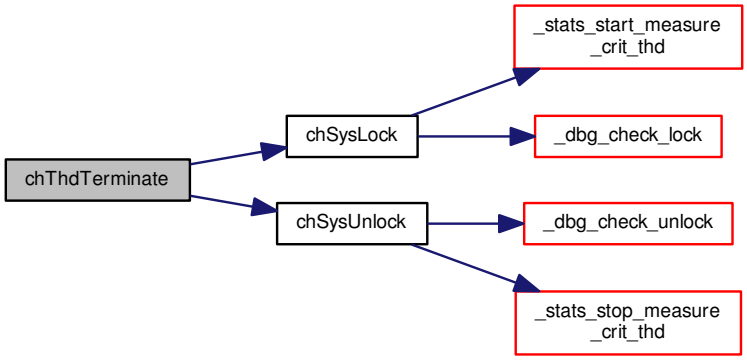
Parameters

`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.

Here is the call graph for this function:



8.8.4.16 void chThdSleep (sysinterval_t *time*)

Suspends the invoking thread for the specified time.

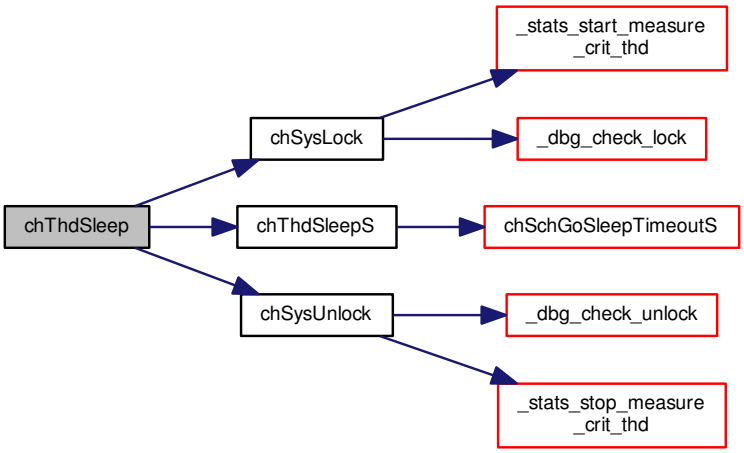
Parameters

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.

Here is the call graph for this function:



8.8.4.17 void chThdSleepUntil (systime_t *time*)

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

Note

The function has no concept of "past", all specifiable times are in the future, this means that if you call this function exceeding your calculated intervals then the function will return in a far future time, not immediately.

See also

chThdSleepUntilWindowed()

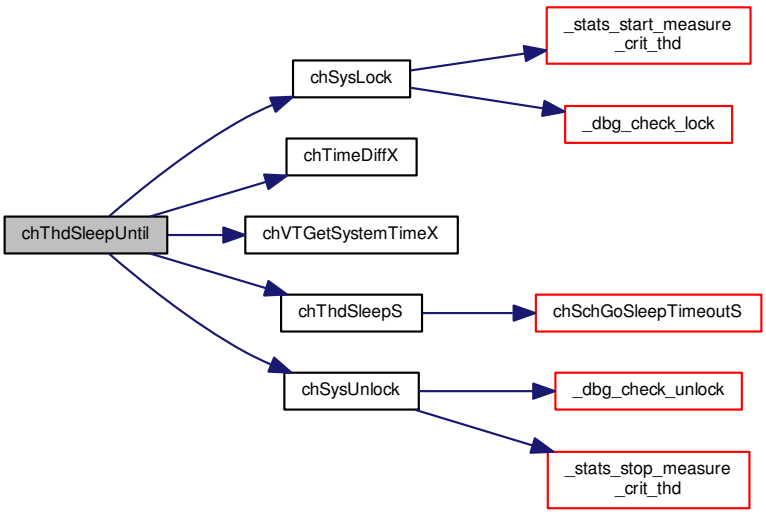
Parameters

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.

Here is the call graph for this function:



8.8.4.18 systime_t chThdSleepUntilWindowed (systime_t *prev*, systime_t *next*)

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

Note

The system time is assumed to be between `prev` and `time` else the call is assumed to have been called outside the allowed time interval, in this case no sleep is performed.

See also

chThdSleepUntil()

Parameters

`in`	*prev*	absolute system time of the previous deadline
`in`	*next*	absolute system time of the next deadline

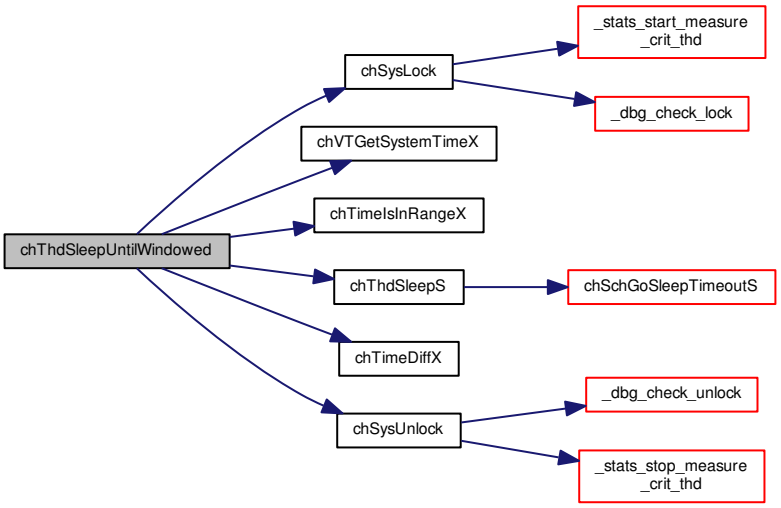
Returns

the `next` parameter

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:



8.8.4.19 void chThdYield (void)

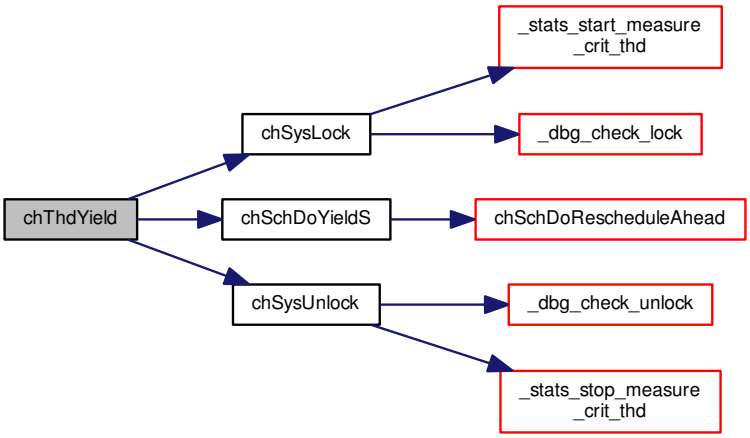
Yields the time slot.

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.

Here is the call graph for this function:



8.8.4.20 msg_t chThdSuspendS (thread_reference_t * *trp*)

Sends the current thread sleeping and sets a reference variable.

Note

This function must reschedule, it can only be called from thread context.

Parameters

in	*trp*	a pointer to a thread reference object

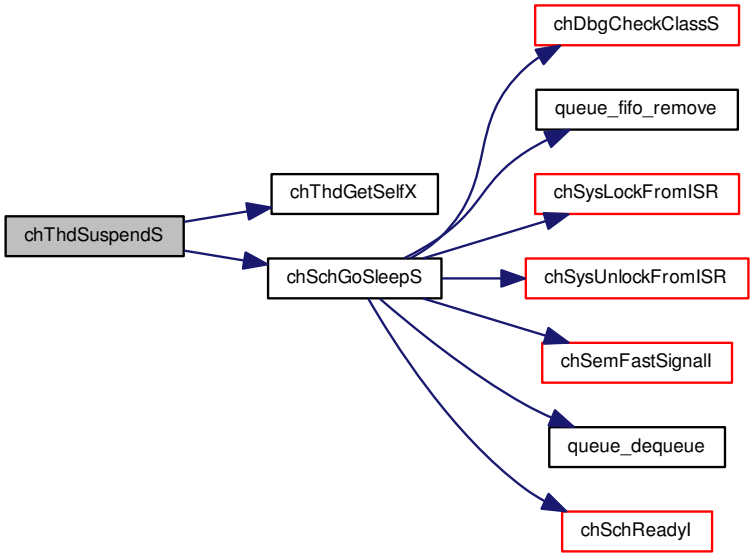
Returns

The wake up message.

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:



8.8.4.21 msg_t chThdSuspendTimeoutS (thread_reference_t * *trp*, sysinterval_t *timeout*)

Sends the current thread sleeping and sets a reference variable.

Note

This function must reschedule, it can only be called from thread context.

Parameters

in	*trp*	a pointer to a thread reference object
in	*timeout*	the timeout in system ticks, the special values are handled as follow: • *TIME_INFINITE* the thread enters an infinite sleep state. • *TIME_IMMEDIATE* the thread is not enqueued and the function returns `MSG_TIMEOUT` as if a timeout occurred.

Returns

The wake up message.

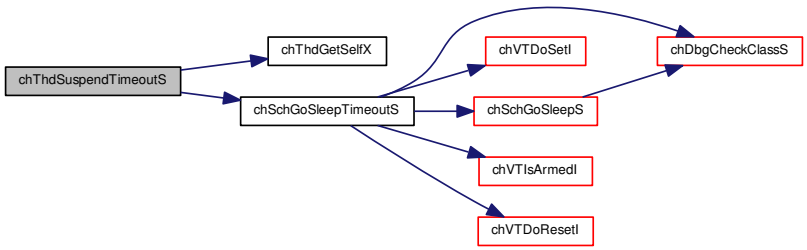
Return values

MSG_TIMEOUT	if the operation 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:



8.8.4.22 void chThdResumeI (thread_reference_t * *trp*, msg_t *msg*)

Wakes up a thread waiting on a thread reference object.

Note

This function must not reschedule because it can be called from ISR context.

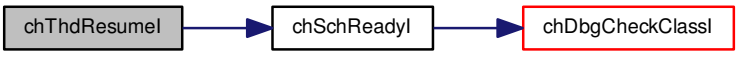
Parameters

in	*trp*	a pointer to a thread reference object
in	*msg*	the message code

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:



8.8.4.23 void chThdResumeS (thread_reference_t * *trp*, msg_t *msg*)

Wakes up a thread waiting on a thread reference object.

Note

This function must reschedule, it can only be called from thread context.

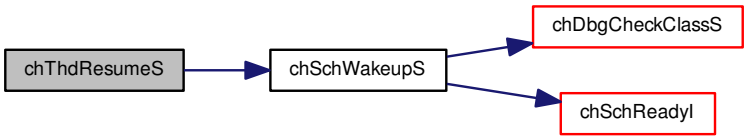
Parameters

in	*trp*	a pointer to a thread reference object
in	*msg*	the message code

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:



8.8.4.24 void chThdResume (thread_reference_t * *trp*, msg_t *msg*)

Wakes up a thread waiting on a thread reference object.

Note

This function must reschedule, it can only be called from thread context.

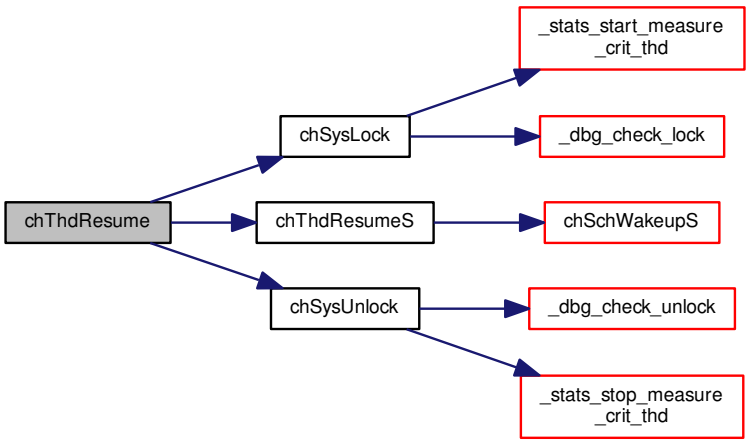
Parameters

in	*trp*	a pointer to a thread reference object
in	*msg*	the message 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:



8.8.4.25 msg_t chThdEnqueueTimeoutS (threads_queue_t * *tqp*, sysinterval_t *timeout*)

Enqueues the caller thread on a threads queue object.

The caller thread is enqueued and put to sleep until it is dequeued or the specified timeouts expires.

Parameters

in	*tqp*	pointer to the threads queue object
in	*timeout*	the timeout in system ticks, the special values are handled as follow: • *TIME_INFINITE* the thread enters an infinite sleep state. • *TIME_IMMEDIATE* the thread is not enqueued and the function returns `MSG_TIMEOUT` as if a timeout occurred.

Returns

The message from `osalQueueWakeupOneI()` or `osalQueueWakeupAllI()` functions.

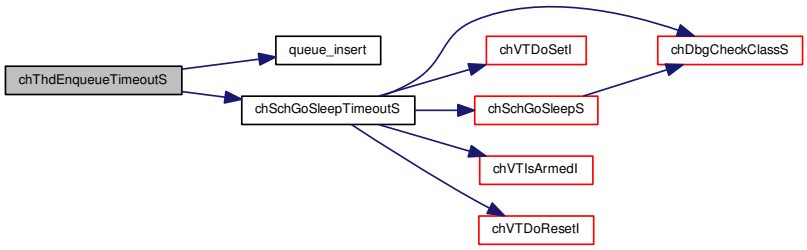
Return values

MSG_TIMEOUT	if the thread has not been dequeued within the specified timeout or if the function has been invoked with `TIME_IMMEDIATE` as timeout specification.

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:



8.8.4.26 void chThdDequeueNextI (threads_queue_t * *tqp*, msg_t *msg*)

Dequeues and wakes up one thread from the threads queue object, if any.

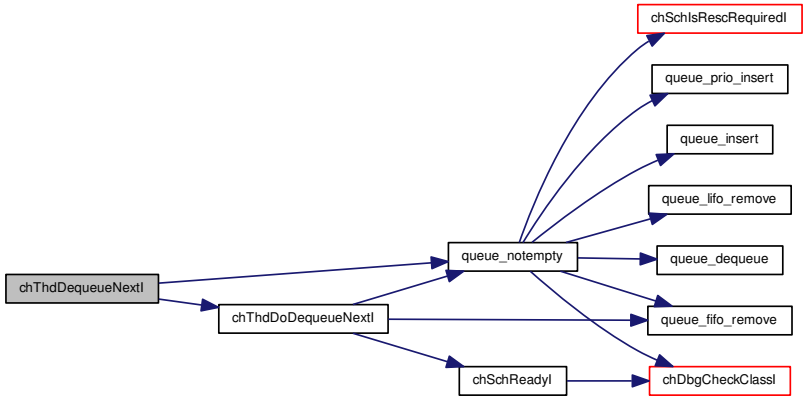
Parameters

in	*tqp*	pointer to the threads queue object
in	*msg*	the message code

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:



8.8.4.27 void chThdDequeueAllI (threads_queue_t * *tqp*, msg_t *msg*)

Dequeues and wakes up all threads from the threads queue object.

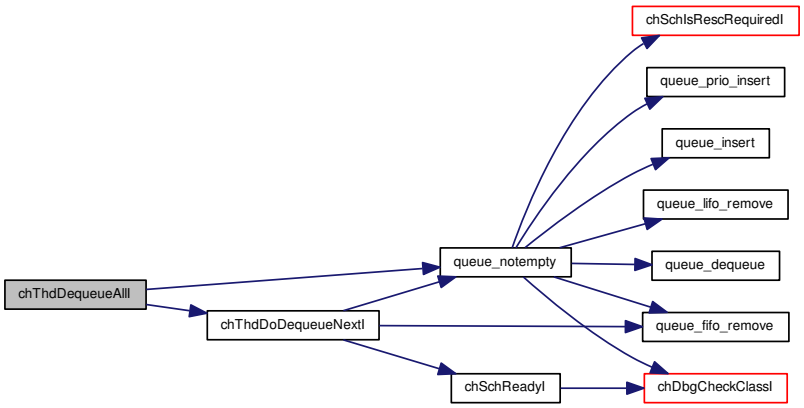
Parameters

in	*tqp*	pointer to the threads queue object
in	*msg*	the message code

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:



8.8.4.28 static thread_t* chThdGetSelfX (void) [inline], [static]

Returns a pointer to the current `thread_t`.

Returns

A pointer to the current thread.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.8.4.29 static tprio_t chThdGetPriorityX (void) [inline], [static]

Returns the current thread priority.

Note

Can be invoked in any context.

Returns

The current thread priority.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:

8.8.4.30 static systime_t chThdGetTicksX (thread_t * *tp*) [inline], [static]

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

Returns

The number of consumed system ticks.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.8.4.31 static stkalign_t* chThdGetWorkingAreaX (thread_t * *tp*) [inline], [static]

Returns the working area base of the specified thread.

Parameters

`in`	*tp*	pointer to the thread

Returns

The working area base pointer.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.8.4.32 static bool chThdTerminatedX (thread_t * tp) [inline], [static]

Verifies if the specified thread is in the CH_STATE_FINAL state.

Parameters

in	*tp*	pointer to the thread

Return values

true	thread terminated.
false	thread not terminated.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.8.4.33 static bool chThdShouldTerminateX (void) [inline], [static]

Verifies if the current thread has a termination request pending.

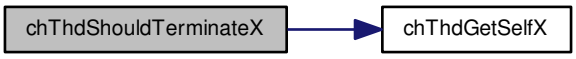
Return values

true	termination request pending.
false	termination request not pending.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.8.4.34 static thread_t* chThdStartI (thread_t * tp) [inline], [static]

Resumes a thread created with chThdCreateI().

Parameters

in	*tp*	pointer to the thread

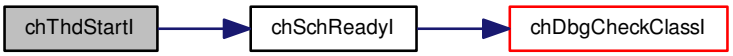
Returns

The pointer to the `thread_t` 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:



8.8.4.35 static void chThdSleepS (sysinterval_t *ticks*) [inline], [static]

Suspends the invoking thread for the specified number of ticks.

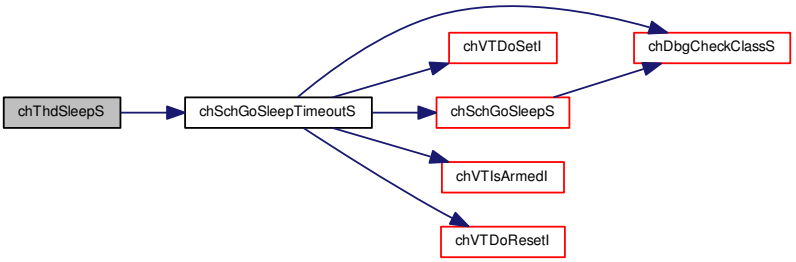
Parameters

in	*ticks*	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.

Here is the call graph for this function:



8.8.4.36 static void chThdQueueObjectInit (threads_queue_t * tqp) [inline], [static]

Initializes a threads queue object.

Parameters

out	*tqp*	pointer to the threads queue object

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:

8.8.4.37 static bool chThdQueueIsEmptyI (threads_queue_t * tqp) [inline], [static]

Evaluates to `true` if the specified queue is empty.

Parameters

out	*tqp*	pointer to the threads queue object

Returns

The queue status.

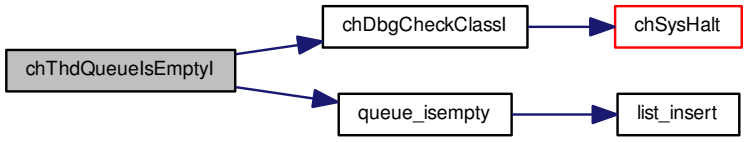
Return values

false	if the queue is not empty.
true	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:



8.8.4.38 static void chThdDoDequeueNextI (threads_queue_t * *tqp*, msg_t *msg*) [inline], [static]

Dequeues and wakes up one thread from the threads queue object.

Dequeues one thread from the queue without checking if the queue is empty.

Precondition

The queue must contain at least an object.

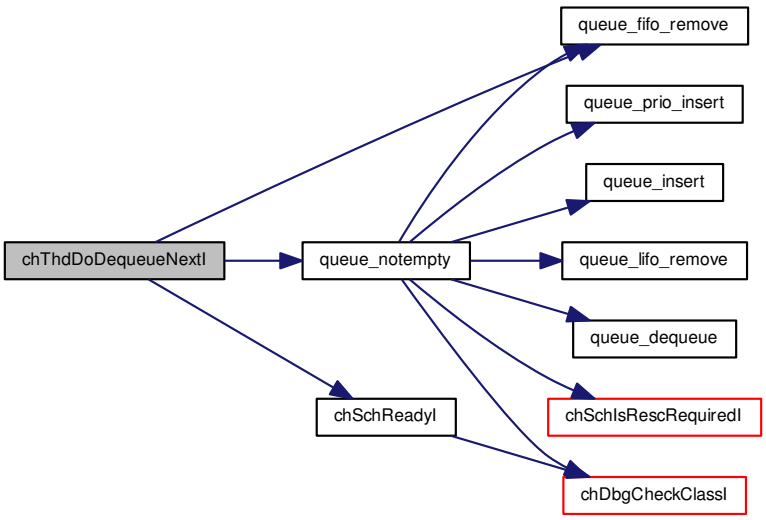
Parameters

in	*tqp*	pointer to the threads queue object
in	*msg*	the message code

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:



8.9 Time and Virtual Timers

8.9.1 Detailed Description

Time and Virtual Timers related APIs and services.

Functions

- void _vt_init (void)

  Virtual Timers initialization.
- void chVTDoSetI (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)

  Enables a virtual timer.
- void chVTDoResetI (virtual_timer_t *vtp)

  Disables a Virtual Timer.
- static void chVTObjectInit (virtual_timer_t *vtp)

  Initializes a `virtual_timer_t` object.
- static systime_t chVTGetSystemTimeX (void)

  Current system time.
- static systime_t chVTGetSystemTime (void)

  Current system time.
- static sysinterval_t chVTTimeElapsedSinceX (systime_t start)

  Returns the elapsed time since the specified start time.
- static bool chVTIsSystemTimeWithinX (systime_t start, systime_t end)

  Checks if the current system time is within the specified time window.
- static bool chVTIsSystemTimeWithin (systime_t start, systime_t end)

  Checks if the current system time is within the specified time window.
- static bool chVTGetTimersStateI (sysinterval_t *timep)

  Returns the time interval until the next timer event.
- static bool chVTIsArmedI (const virtual_timer_t *vtp)

  Returns `true` if the specified timer is armed.
- static bool chVTIsArmed (const virtual_timer_t *vtp)

  Returns `true` if the specified timer is armed.
- static void chVTResetI (virtual_timer_t *vtp)

  Disables a Virtual Timer.
- static void chVTReset (virtual_timer_t *vtp)

  Disables a Virtual Timer.
- static void chVTSetI (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)

  Enables a virtual timer.
- static void chVTSet (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)

  Enables a virtual timer.
- static void chVTDoTickI (void)

  Virtual timers ticker.

8.9.2 Function Documentation

8.9.2.1 void _vt_init (void)

Virtual Timers initialization.

Note

Internal use only.

Function Class:

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

8.9.2.2 void chVTDoSetI (virtual_timer_t * *vtp*, sysinterval_t *delay*, vtfunc_t *vtfunc*, void * *par*)

Enables a virtual timer.

The timer is enabled and programmed to trigger after the delay specified as parameter.

Precondition

The timer must not be already armed before calling this function.

Note

The callback function is invoked from interrupt context.

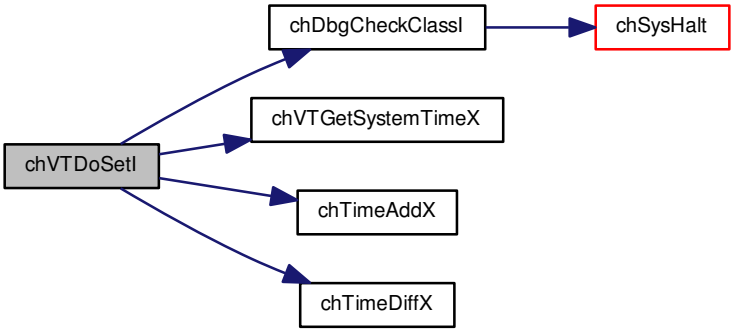
Parameters

out	*vtp*	the `virtual_timer_t` structure pointer
in	*delay*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* is allowed but interpreted as a normal time specification. • *TIME_IMMEDIATE* this value is not allowed.
in	*vtfunc*	the timer callback function. After invoking the callback the timer is disabled and the structure can be disposed or reused.
in	*par*	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:



8.9.2.3 void chVTDoResetI (virtual_timer_t * *vtp*)

Disables a Virtual Timer.

Precondition

The timer must be in armed state before calling this function.

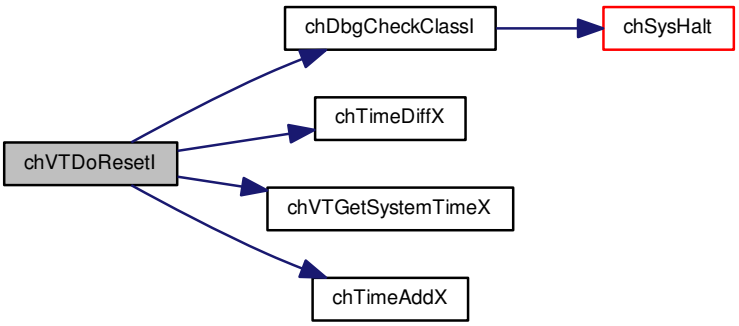
Parameters

in	*vtp*	the `virtual_timer_t` 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:



8.9.2.4 static void chVTObjectInit (virtual_timer_t * *vtp*) [inline], [static]

Initializes a `virtual_timer_t` object.

Note

Initializing a timer object is not strictly required because the function `chVTSetI()` initializes the object too. This function is only useful if you need to perform a `chVTIsArmed()` check before calling `chVTSetI()`.

Parameters

`out`	*vtp*	the `virtual_timer_t` structure pointer

Function Class:

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

8.9.2.5 static systime_t chVTGetSystemTimeX (void) [inline], [static]

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 can be called from any context but its atomicity is not guaranteed on architectures whose word size is less than `systime_t` size.

Returns

The system time in ticks.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.9.2.6 static systime_t chVTGetSystemTime (void) [inline], [static]

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.

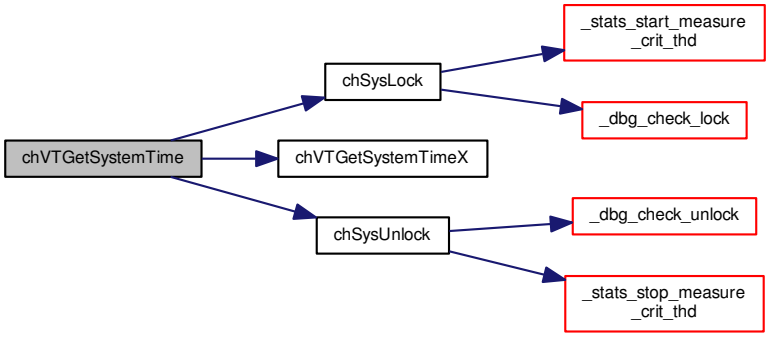
Returns

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.

Here is the call graph for this function:



8.9.2.7 static sysinterval_t chVTTimeElapsedSinceX (systime_t *start*) [inline], [static]

Returns the elapsed time since the specified start time.

Parameters

in	*start*	start time

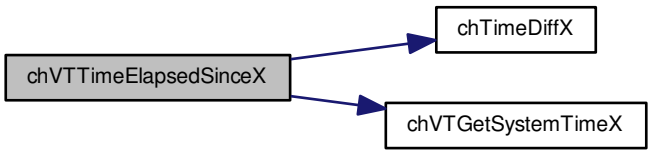
Returns

The elapsed time.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.9.2.8 static bool chVTIsSystemTimeWithinX (systime_t *start*, systime_t *end*) [inline],[static]

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	*start*	the start of the time window (inclusive)
in	*end*	the end of the time window (non inclusive)

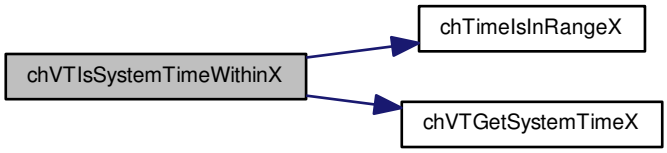
Return values

true	current time within the specified time window.
false	current time not within the specified time window.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

Here is the call graph for this function:



8.9.2.9 static bool chVTIsSystemTimeWithin (systime_t *start*, systime_t *end*) [inline], [static]

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	*start*	the start of the time window (inclusive)
in	*end*	the end of the time window (non inclusive)

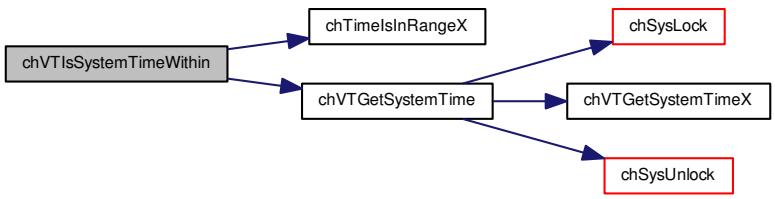
Return values

true	current time within the specified time window.
false	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.

Here is the call graph for this function:



8.9.2.10 static bool chVTGetTimersStateI (sysinterval_t * timep) [inline], [static]

Returns the time interval until the next timer event.

Note

The return value is not perfectly accurate and can report values in excess of CH_CFG_ST_TIMEDELTA ticks.
The interval returned by this function is only meaningful if more timers are not added to the list until the returned time.

Parameters

out	*timep*	pointer to a variable that will contain the time interval until the next timer elapses. This pointer can be NULL if the information is not required.

Returns

The time, in ticks, until next time event.

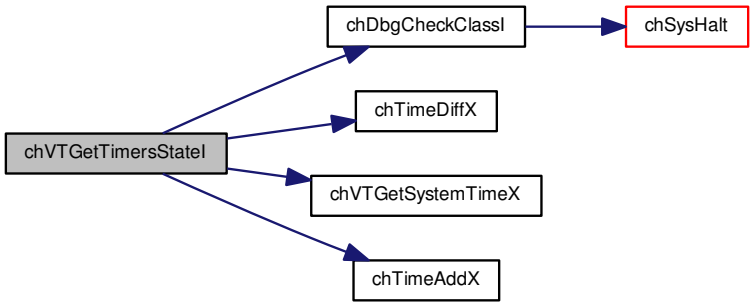
Return values

false	if the timers list is empty.
true	if the timers list contains at least one timer.

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:



8.9.2.11 static bool chVTIsArmedI (const virtual_timer_t * vtp) [inline], [static]

Returns true if the specified timer is armed.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

Parameters

`in`	*vtp*	the `virtual_timer_t` structure pointer

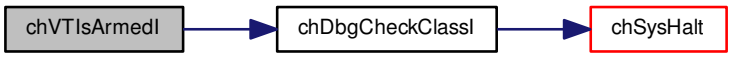
Returns

true if the 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.

Here is the call graph for this function:



8.9.2.12 static bool chVTIsArmed (const virtual_timer_t * *vtp*) [inline], [static]

Returns `true` if the specified timer is armed.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

Parameters

`in`	*vtp*	the `virtual_timer_t` structure pointer

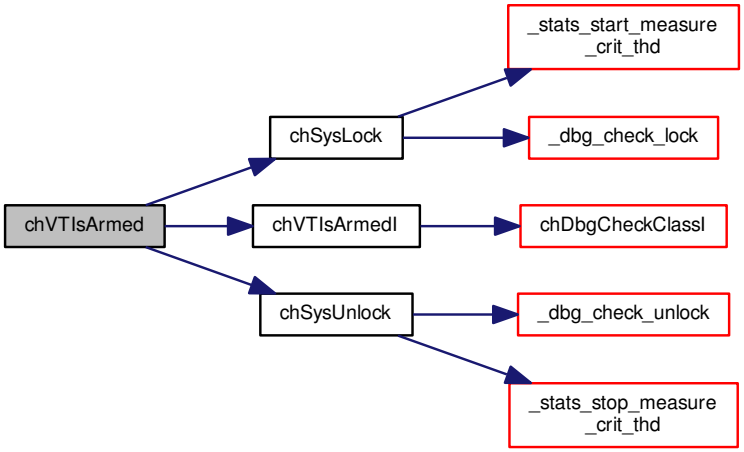
Returns

true if the timer is armed.

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:



8.9.2.13 static void chVTResetI (virtual_timer_t * *vtp*) [inline], [static]

Disables a Virtual Timer.

Note

The timer is first checked and disabled only if armed.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

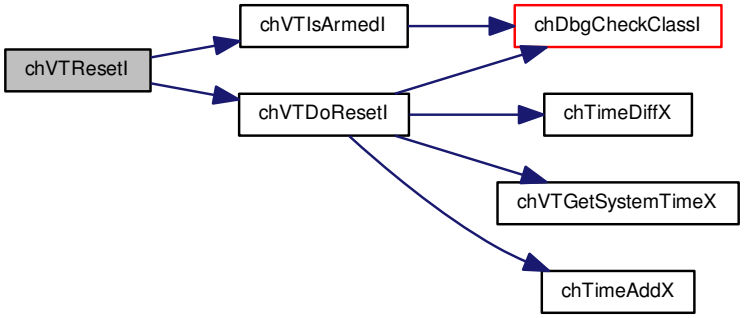
Parameters

in	*vtp*	the `virtual_timer_t` 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:



8.9.2.14 static void chVTReset (virtual_timer_t * *vtp*) [inline], [static]

Disables a Virtual Timer.

Note

The timer is first checked and disabled only if armed.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

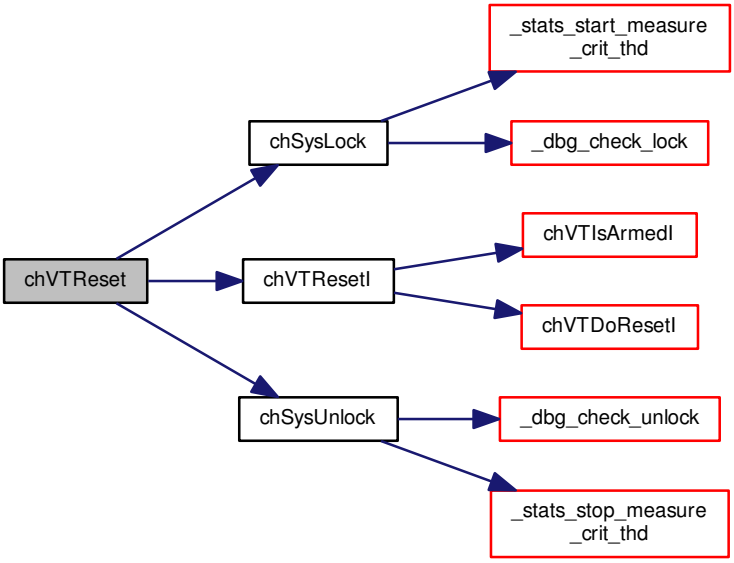
Parameters

in	*vtp*	the `virtual_timer_t` structure pointer

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:



8.9.2.15 static void chVTSetI (virtual_timer_t * *vtp*, sysinterval_t *delay*, vtfunc_t *vtfunc*, void * *par*) [inline], [static]

Enables a virtual timer.

If the virtual timer was already enabled then it is re-enabled using the new parameters.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

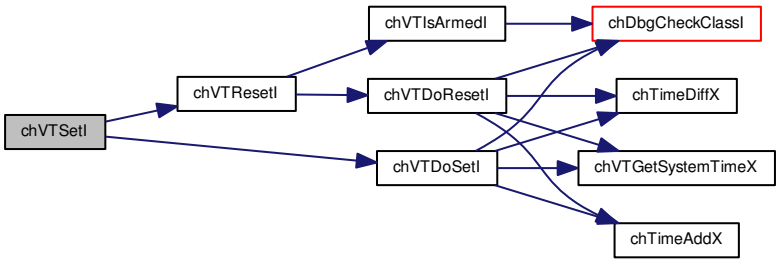
Parameters

in	*vtp*	the `virtual_timer_t` structure pointer
in	*delay*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* is allowed but interpreted as a normal time specification. • *TIME_IMMEDIATE* this value is not allowed.
in	*vtfunc*	the timer callback function. After invoking the callback the timer is disabled and the structure can be disposed or reused.
in	*par*	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:



8.9.2.16 static void chVTSet (virtual_timer_t * *vtp*, sysinterval_t *delay*, vtfunc_t *vtfunc*, void * *par*) [inline], [static]

Enables a virtual timer.

If the virtual timer was already enabled then it is re-enabled using the new parameters.

Precondition

The timer must have been initialized using `chVTObjectInit()` or `chVTDoSetI()`.

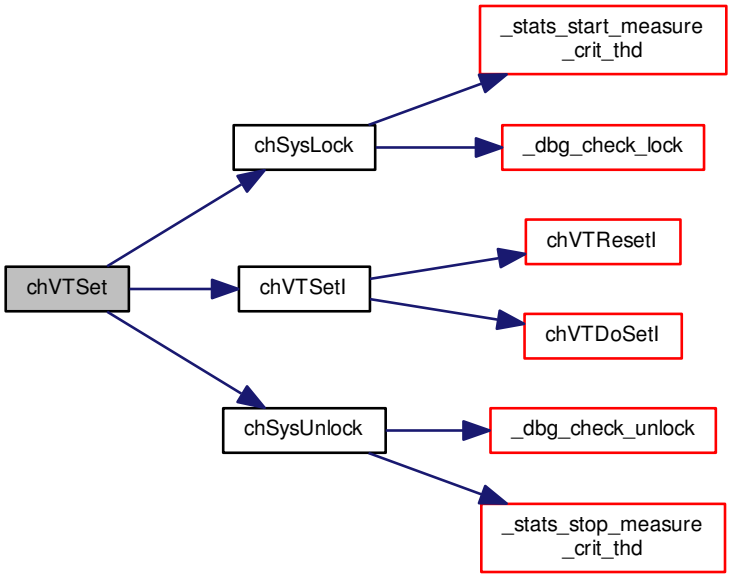
Parameters

in	*vtp*	the `virtual_timer_t` structure pointer
in	*delay*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* is allowed but interpreted as a normal time specification. • *TIME_IMMEDIATE* this value is not allowed.
in	*vtfunc*	the timer callback function. After invoking the callback the timer is disabled and the structure can be disposed or reused.
in	*par*	a parameter that will be passed to the callback function

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:



8.9.2.17 static void chVTDoTickI (void) [inline], [static]

Virtual timers ticker.

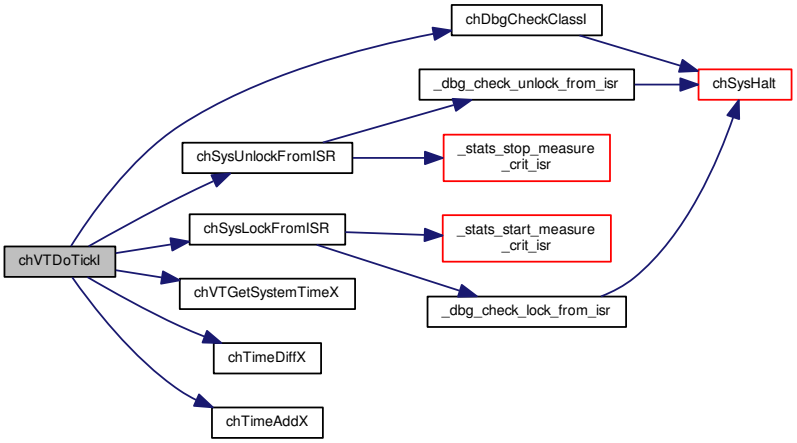
Note

The system lock is released before entering the callback and re-acquired immediately after. It is callback's responsibility to acquire the lock if needed. This is done in order to reduce interrupts jitter when many timers are in use.

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:



8.10 Synchronization

8.10.1 Detailed Description

Synchronization services.

Modules

- Counting Semaphores
- Binary Semaphores
- Mutexes
- Condition Variables
- Event Flags
- Synchronous Messages
- Mailboxes

8.11 Counting Semaphores

8.11.1 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_CFG_USE_SEMAPHORES_PRIORITY` in `chconf.h`.

Precondition

In order to use the semaphore APIs the `CH_CFG_USE_SEMAPHORES` option must be enabled in `chconf.h`.

Macros

- #define _SEMAPHORE_DATA(name, n) {_THREADS_QUEUE_DATA(name.queue), n}

  Data part of a static semaphore initializer.
- #define SEMAPHORE_DECL(name, n) semaphore_t name = _SEMAPHORE_DATA(name, n)

  Static semaphore initializer.

Typedefs

- typedef struct ch_semaphore semaphore_t

  Semaphore structure.

Data Structures

- struct ch_semaphore

  Semaphore structure.

Functions

- void chSemObjectInit (semaphore_t *sp, cnt_t n)

  Initializes a semaphore with the specified counter value.
- void chSemReset (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.
- void chSemResetI (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.
- msg_t chSemWait (semaphore_t *sp)

  Performs a wait operation on a semaphore.
- msg_t chSemWaitS (semaphore_t *sp)

  Performs a wait operation on a semaphore.
- msg_t chSemWaitTimeout (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.
- msg_t chSemWaitTimeoutS (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.
- void chSemSignal (semaphore_t *sp)

  Performs a signal operation on a semaphore.
- void chSemSignalI (semaphore_t *sp)

  Performs a signal operation on a semaphore.
- void chSemAddCounterI (semaphore_t *sp, cnt_t n)

  Adds the specified value to the semaphore counter.
- msg_t chSemSignalWait (semaphore_t *sps, semaphore_t *spw)

  Performs atomic signal and wait operations on two semaphores.
- static void chSemFastWaitI (semaphore_t *sp)

  Decreases the semaphore counter.
- static void chSemFastSignalI (semaphore_t *sp)

  Increases the semaphore counter.
- static cnt_t chSemGetCounterI (const semaphore_t *sp)

  Returns the semaphore counter current value.

8.11.2 Macro Definition Documentation

8.11.2.1 #define _SEMAPHORE_DATA(*name, n*) {_THREADS_QUEUE_DATA(name.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	*name*	the name of the semaphore variable
in	*n*	the counter initial value, this value must be non-negative

8.11.2.2 #define SEMAPHORE_DECL(*name, n*) semaphore_t name = _SEMAPHORE_DATA(name, n)

Static semaphore initializer.

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

Parameters

in	*name*	the name of the semaphore variable
in	*n*	the counter initial value, this value must be non-negative

8.11.3 Typedef Documentation

8.11.3.1 typedef struct ch_semaphore semaphore_t

Semaphore structure.

8.11.4 Function Documentation

8.11.4.1 void chSemObjectInit (semaphore_t * *sp*, cnt_t *n*)

Initializes a semaphore with the specified counter value.

Parameters

out	*sp*	pointer to a `semaphore_t` structure
in	*n*	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.

Here is the call graph for this function:

8.11.4.2 void chSemReset (semaphore_t * *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 waked up by a reset rather than a signal because the `chSemWait()` will return `MSG_RESET` instead of `MSG_OK`.

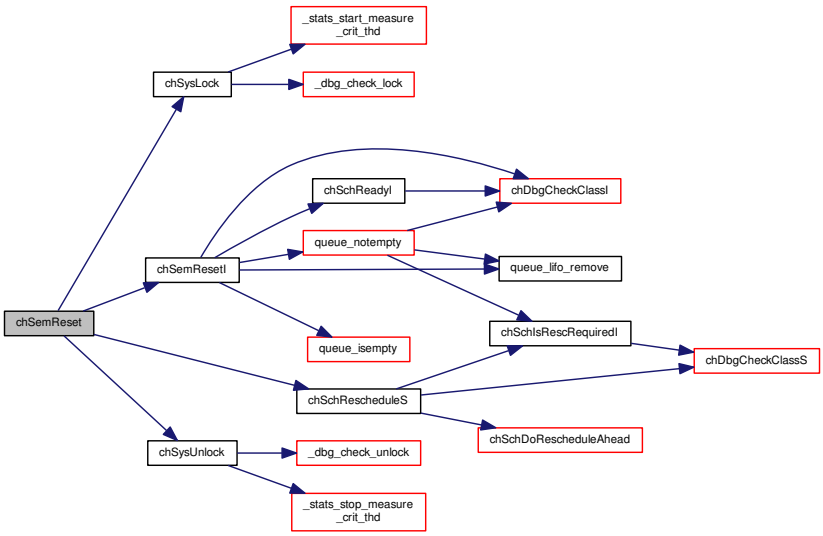
Parameters

in	*sp*	pointer to a `semaphore_t` structure
in	*n*	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:



8.11.4.3 void chSemResetI (semaphore_t * *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 waked up by a reset rather than a signal because the `chSemWait()` will return `MSG_RESET` instead of `MSG_OK`.

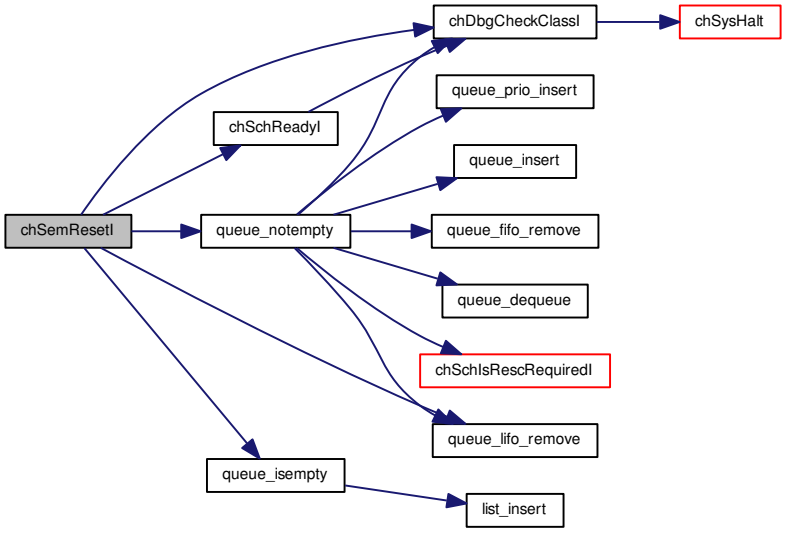
Parameters

in	*sp*	pointer to a `semaphore_t` structure
in	*n*	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:



8.11.4.4 msg_t chSemWait (semaphore_t * *sp*)

Performs a wait operation on a semaphore.

Parameters

in	*sp*	pointer to a semaphore_t structure

Returns

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

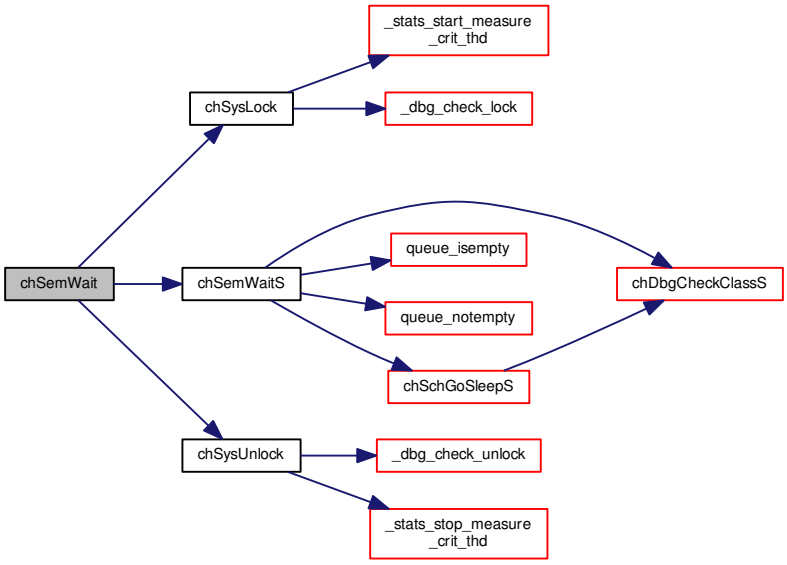
Return values

MSG_OK	if the thread has not stopped on the semaphore or the semaphore has been signaled.
MSG_RESET	if the semaphore has been reset using chSemReset().

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:



8.11.4.5 msg_t chSemWaitS (semaphore_t * *sp*)

Performs a wait operation on a semaphore.

Parameters

in	*sp*	pointer to a `semaphore_t` structure

Returns

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

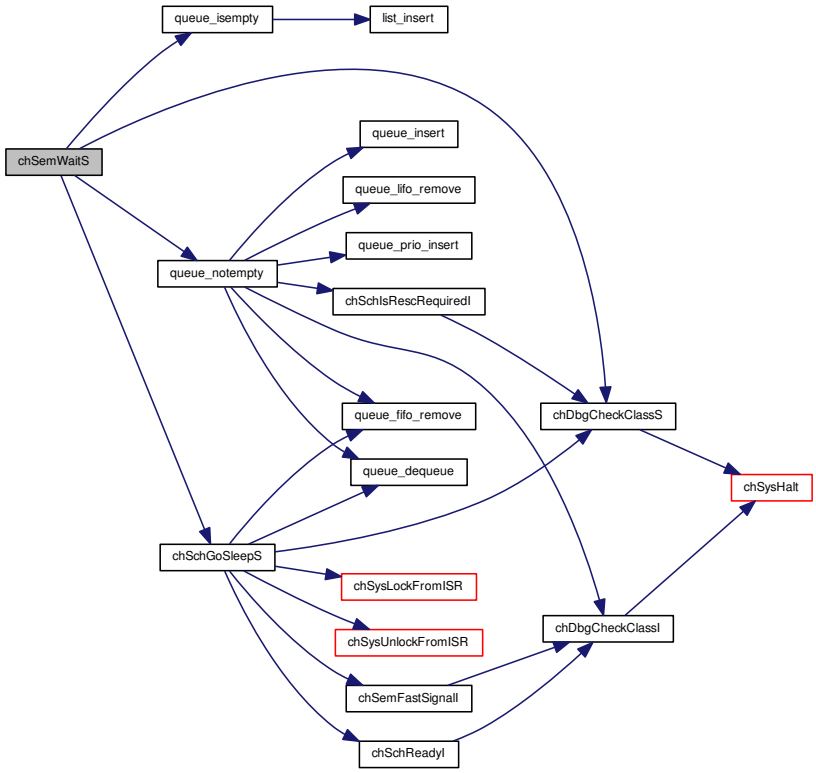
Return values

MSG_OK	if the thread has not stopped on the semaphore or the semaphore has been signaled.
MSG_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:



8.11.4.6 msg_t chSemWaitTimeout (semaphore_t * *sp*, sysinterval_t *timeout*)

Performs a wait operation on a semaphore with timeout specification.

Parameters

in	*sp*	pointer to a `semaphore_t` structure
in	*timeout*	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.

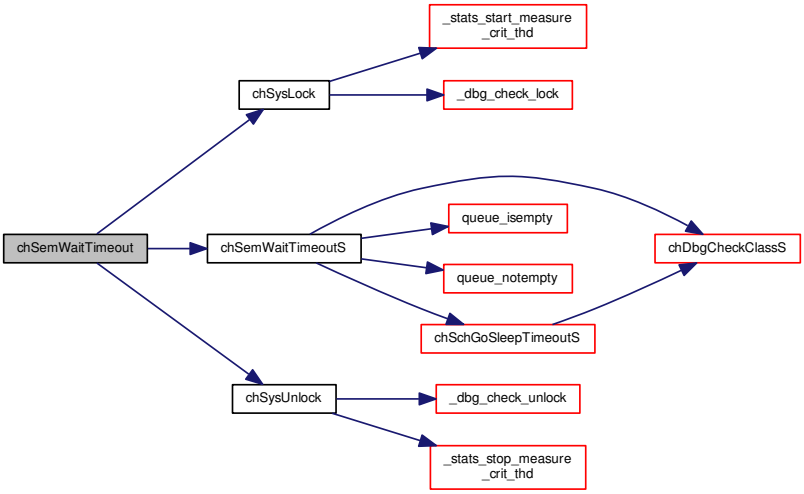
Return values

MSG_OK	if the thread has not stopped on the semaphore or the semaphore has been signaled.
MSG_RESET	if the semaphore has been reset using `chSemReset()`.
MSG_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:



8.11.4.7 msg_t chSemWaitTimeoutS (semaphore_t * *sp*, sysinterval_t *timeout*)

Performs a wait operation on a semaphore with timeout specification.

Parameters

in	*sp*	pointer to a `semaphore_t` structure
in	*timeout*	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.

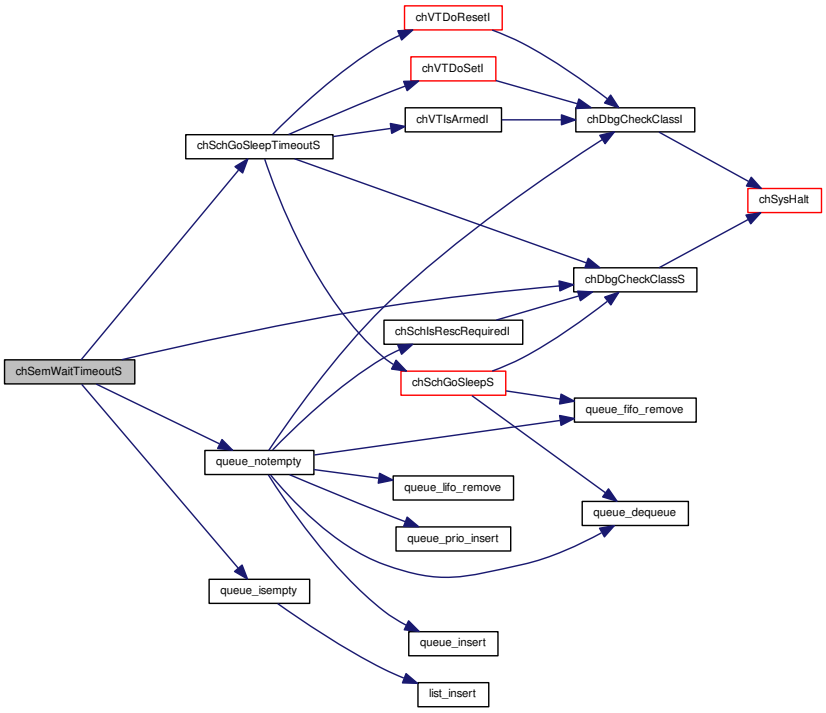
Return values

MSG_OK	if the thread has not stopped on the semaphore or the semaphore has been signaled.
MSG_RESET	if the semaphore has been reset using `chSemReset()`.
MSG_TIMEOUT	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:



8.11.4.8 void chSemSignal (semaphore_t * *sp*)

Performs a signal operation on a semaphore.

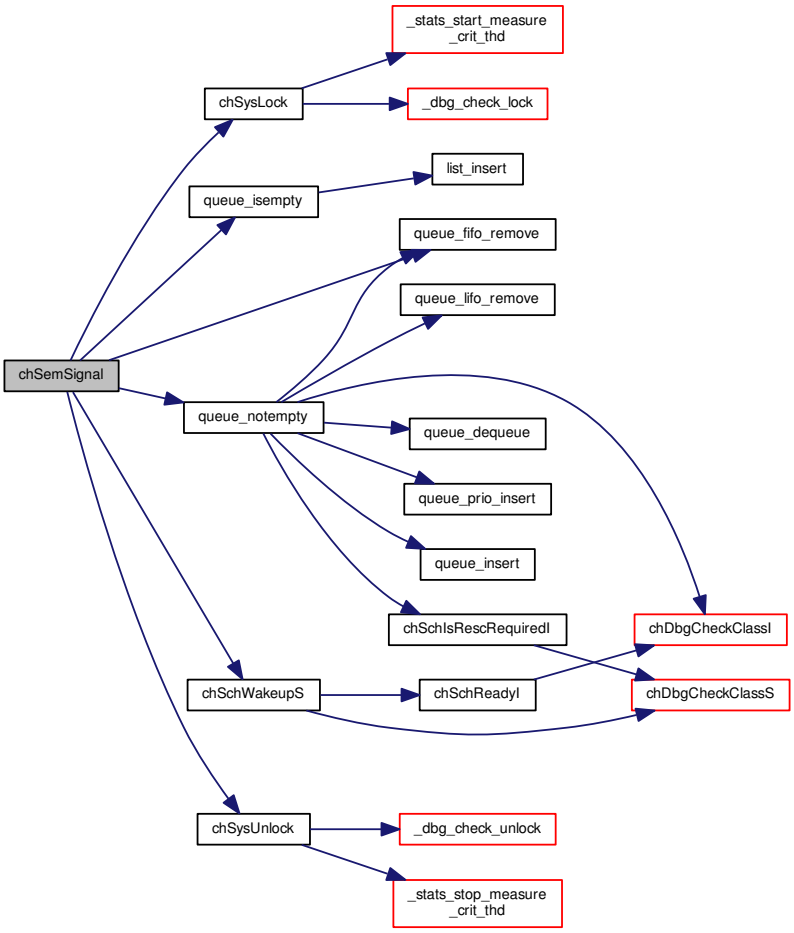
Parameters

in	*sp*	pointer to a `semaphore_t` 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:



8.11.4.9 void chSemSignalI (semaphore_t * *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.

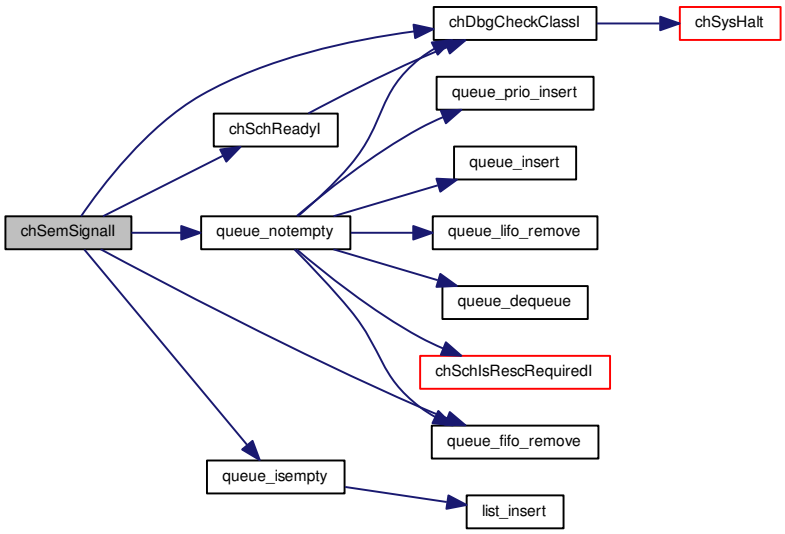
Parameters

in	*sp*	pointer to a `semaphore_t` 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:



8.11.4.10 void chSemAddCounterI (semaphore_t * *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.

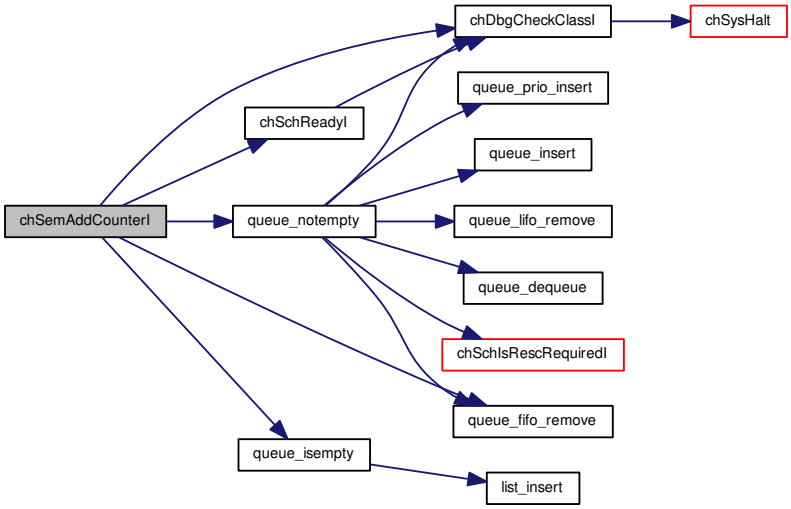
Parameters

`in`	*sp*	pointer to a `semaphore_t` structure
`in`	*n*	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:



8.11.4.11 msg_t chSemSignalWait (semaphore_t * *sps*, semaphore_t * *spw*)

Performs atomic signal and wait operations on two semaphores.

Parameters

in	*sps*	pointer to a `semaphore_t` structure to be signaled
in	*spw*	pointer to a `semaphore_t` structure to wait on

Returns

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

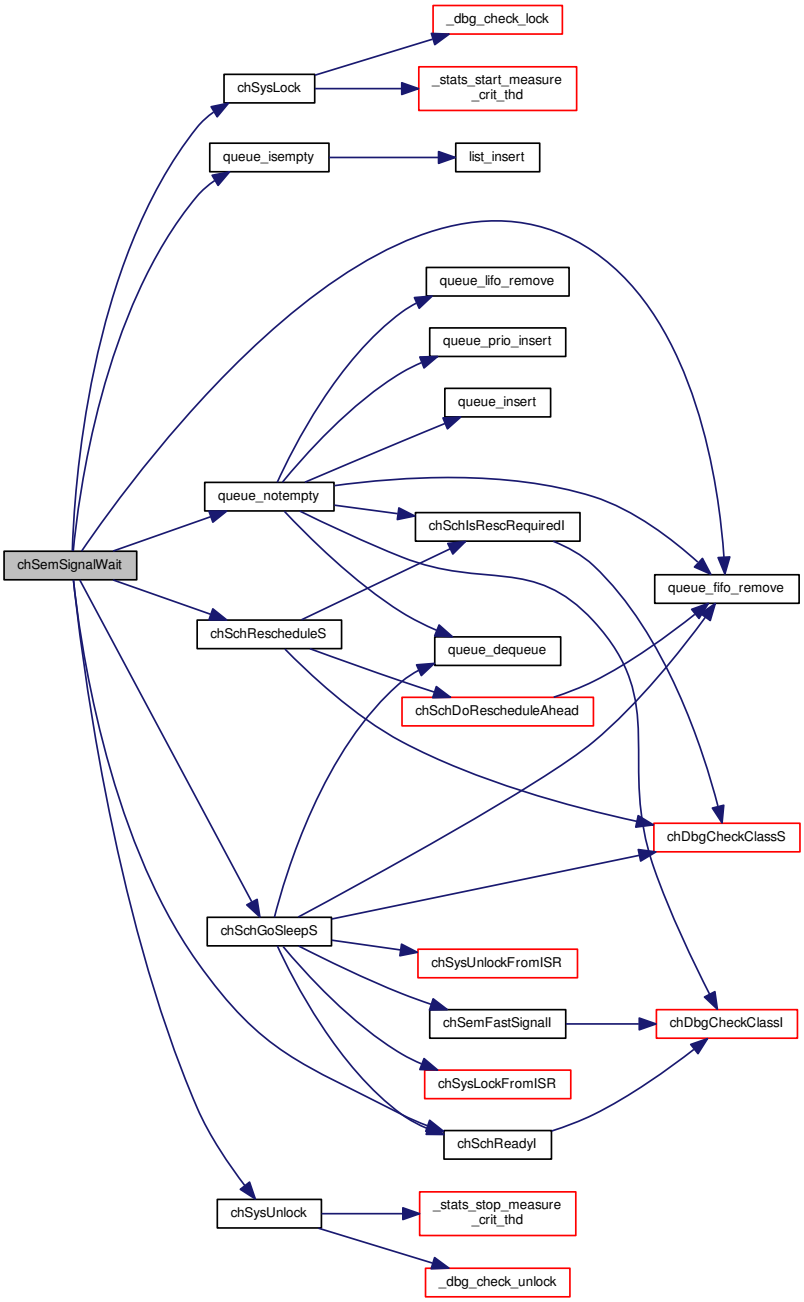
Return values

MSG_OK	if the thread has not stopped on the semaphore or the semaphore has been signaled.
MSG_RESET	if the semaphore has been reset using `chSemReset()`.

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:



8.11.4.12 static void chSemFastWaitI (semaphore_t * *sp*) [inline], [static]

Decreases the semaphore counter.

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

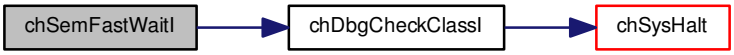
Parameters

in	*sp*	pointer to a semaphore_t 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:



8.11.4.13 static void chSemFastSignalI (semaphore_t * *sp*) [inline], [static]

Increases the semaphore counter.

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

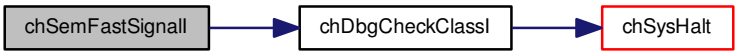
Parameters

in	*sp*	pointer to a semaphore_t 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:



8.11.4.14 static cnt_t chSemGetCounterI (const semaphore_t * *sp*) [inline], [static]

Returns the semaphore counter current value.

Parameters

in	*sp*	pointer to a semaphore_t structure

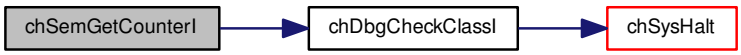
Returns

The semaphore counter 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.

Here is the call graph for this function:



8.12 Binary Semaphores

8.12.1 Detailed Description

Binary semaphores related APIs and services.

Operation mode

Binary semaphores are implemented as a set of inline functions 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 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_CFG_USE_SEMAPHORES` option must be enabled in `chconf.h`.

Macros

- #define _BSEMAPHORE_DATA(name, taken) {_SEMAPHORE_DATA(name.sem, ((taken) ? 0 : 1))}

  Data part of a static semaphore initializer.
- #define BSEMAPHORE_DECL(name, taken) binary_semaphore_t name = _BSEMAPHORE_DATA(name, taken)

  Static semaphore initializer.

Typedefs

- typedef struct ch_binary_semaphore binary_semaphore_t

  Binary semaphore type.

Data Structures

- struct ch_binary_semaphore

  Binary semaphore type.

Functions

- static void chBSemObjectInit (binary_semaphore_t *bsp, bool taken)

  Initializes a binary semaphore.
- static msg_t chBSemWait (binary_semaphore_t *bsp)

  Wait operation on the binary semaphore.
- static msg_t chBSemWaitS (binary_semaphore_t *bsp)

  Wait operation on the binary semaphore.
- static msg_t chBSemWaitTimeoutS (binary_semaphore_t *bsp, sysinterval_t timeout)

Wait operation on the binary semaphore.

- static msg_t chBSemWaitTimeout (binary_semaphore_t *bsp, sysinterval_t timeout)

  Wait operation on the binary semaphore.
- static void chBSemResetI (binary_semaphore_t *bsp, bool taken)

  Reset operation on the binary semaphore.
- static void chBSemReset (binary_semaphore_t *bsp, bool taken)

  Reset operation on the binary semaphore.
- static void chBSemSignalI (binary_semaphore_t *bsp)

  Performs a signal operation on a binary semaphore.
- static void chBSemSignal (binary_semaphore_t *bsp)

  Performs a signal operation on a binary semaphore.
- static bool chBSemGetStateI (const binary_semaphore_t *bsp)

  Returns the binary semaphore current state.

8.12.2 Macro Definition Documentation

8.12.2.1 #define _BSEMAPHORE_DATA(*name, taken*) {_SEMAPHORE_DATA(name.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	*name*	the name of the semaphore variable
in	*taken*	the semaphore initial state

8.12.2.2 #define BSEMAPHORE_DECL(*name, taken*) binary_semaphore_t name = _BSEMAPHORE_DATA(name, taken)

Static semaphore initializer.

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

Parameters

in	*name*	the name of the semaphore variable
in	*taken*	the semaphore initial state

8.12.3 Typedef Documentation

8.12.3.1 typedef struct ch_binary_semaphore binary_semaphore_t

Binary semaphore type.

8.12.4 Function Documentation

8.12.4.1 static void chBSemObjectInit (binary_semaphore_t * *bsp,* bool *taken*) [inline], [static]

Initializes a binary semaphore.

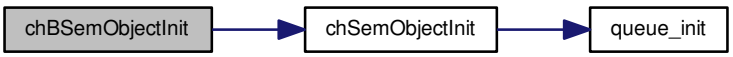
Parameters

out	*bsp*	pointer to a `binary_semaphore_t` structure
in	*taken*	initial state of the binary semaphore: • *false*, the initial state is not taken. • *true*, the initial state is taken.

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:



8.12.4.2 static msg_t chBSemWait (binary_semaphore_t * *bsp*) [inline], [static]

Wait operation on the binary semaphore.

Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure

Returns

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

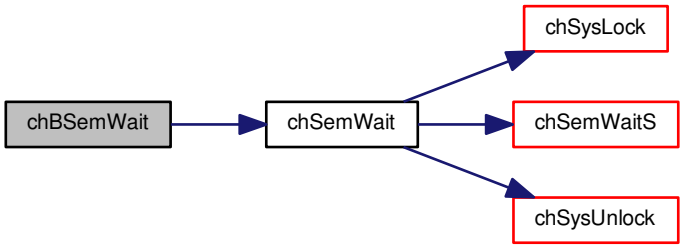
Return values

MSG_OK	if the binary semaphore has been successfully taken.
MSG_RESET	if the binary semaphore has been reset using `bsemReset()`.

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:



8.12.4.3 static msg_t chBSemWaitS (binary_semaphore_t * *bsp*) [inline], [static]

Wait operation on the binary semaphore.

Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure

Returns

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

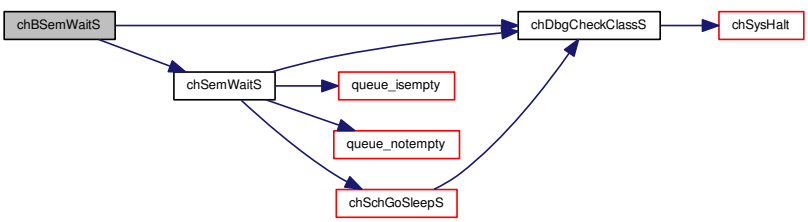
Return values

MSG_OK	if the binary semaphore has been successfully taken.
MSG_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.

Here is the call graph for this function:



8.12.4.4 static msg_t chBSemWaitTimeoutS (binary_semaphore_t * *bsp*, sysinterval_t *timeout*) [inline], [static]

Wait operation on the binary semaphore.

Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure
in	*timeout*	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.

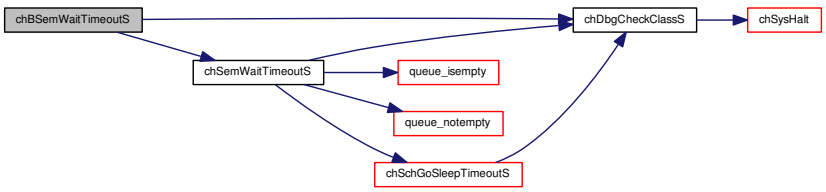
Return values

MSG_OK	if the binary semaphore has been successfully taken.
MSG_RESET	if the binary semaphore has been reset using `bsemReset()`.
MSG_TIMEOUT	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.

Here is the call graph for this function:



8.12.4.5 static msg_t chBSemWaitTimeout (binary_semaphore_t * *bsp*, sysinterval_t *timeout*) [inline], [static]

Wait operation on the binary semaphore.

Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure
in	*timeout*	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.

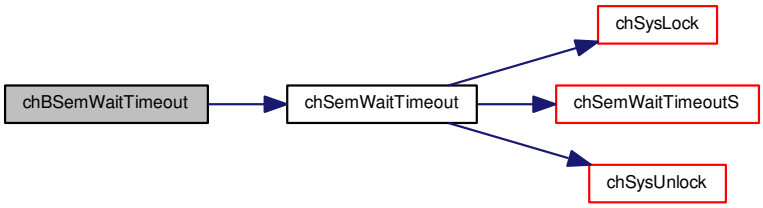
Return values

MSG_OK	if the binary semaphore has been successfully taken.
MSG_RESET	if the binary semaphore has been reset using `bsemReset()`.
MSG_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.

Here is the call graph for this function:



8.12.4.6 static void chBSemResetI (binary_semaphore_t * *bsp*, bool *taken*) [inline], [static]

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 `MSG_RESET` instead of `MSG_OK`.
This function does not reschedule.

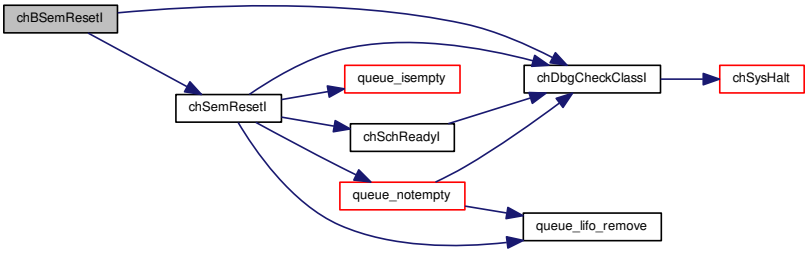
Parameters

`in`	*bsp*	pointer to a `binary_semaphore_t` structure
`in`	*taken*	new state of the binary semaphore • *false*, the new state is not taken. • *true*, 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.

Here is the call graph for this function:



8.12.4.7 static void chBSemReset (binary_semaphore_t * *bsp*, bool *taken*) [inline], [static]

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 `bsem↩Wait()` will return `MSG_RESET` instead of `MSG_OK`.

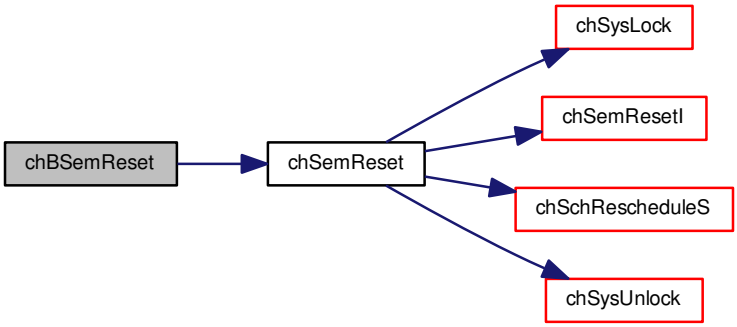
Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure
in	*taken*	new state of the binary semaphore • *false*, the new state is not taken. • *true*, 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.

Here is the call graph for this function:



8.12.4.8 static void chBSemSignalI (binary_semaphore_t * *bsp*) [inline], [static]

Performs a signal operation on a binary semaphore.

Note

This function does not reschedule.

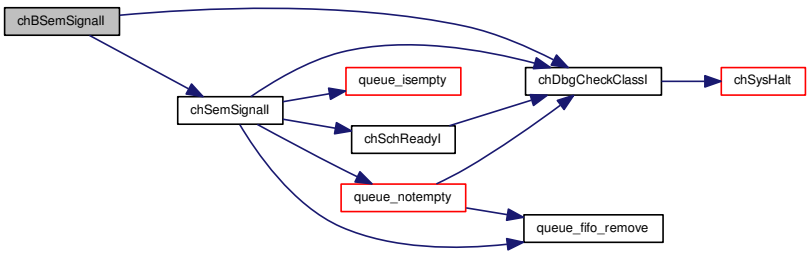
Parameters

in	*bsp*	pointer to a `binary_semaphore_t` 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:



8.12.4.9 static void chBSemSignal (binary_semaphore_t * *bsp*) [inline], [static]

Performs a signal operation on a binary semaphore.

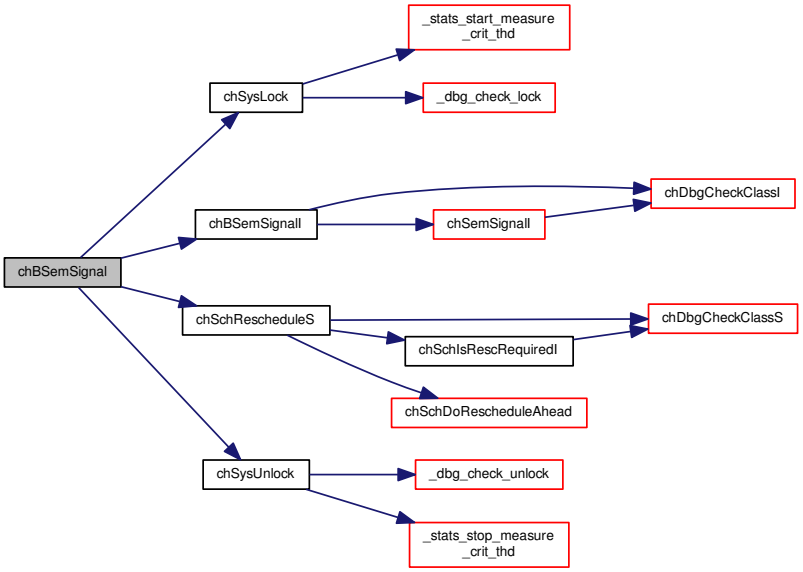
Parameters

in	*bsp*	pointer to a `binary_semaphore_t` 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:



8.12.4.10 static bool chBSemGetStateI (const binary_semaphore_t * *bsp*) [inline], [static]

Returns the binary semaphore current state.

Parameters

in	*bsp*	pointer to a `binary_semaphore_t` structure

Returns

The binary semaphore current state.

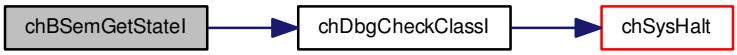
Return values

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.

Here is the call graph for this function:



8.13 Mutexes

8.13.1 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 must always be performed in lock-reverse order. This restriction both improves the performance and is required for an efficient implementation of the priority inheritance mechanism.
Operating under this restriction also ensures that deadlocks are no possible.

Recursive mode

By default mutexes are not recursive, this mean that it is not possible to take a mutex already owned by the same thread. It is possible to enable the recursive behavior by enabling the option `CH_CFG_USE_MUTEXES_RECURSIVE`.

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_CFG_USE_MUTEXES` option must be enabled in `chconf.h`.

Postcondition

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

Macros

- #define _MUTEX_DATA(name) {_THREADS_QUEUE_DATA(name.queue), NULL, NULL, 0}
  Data part of a static mutex initializer.
- #define MUTEX_DECL(name) mutex_t name = _MUTEX_DATA(name)
  Static mutex initializer.

Typedefs

- typedef struct ch_mutex mutex_t

  Type of a mutex structure.

Data Structures

- struct ch_mutex

  Mutex structure.

Functions

- void chMtxObjectInit (mutex_t *mp)

  Initializes s `mutex_t` structure.
- void chMtxLock (mutex_t *mp)

  Locks the specified mutex.
- void chMtxLockS (mutex_t *mp)

  Locks the specified mutex.
- bool chMtxTryLock (mutex_t *mp)

  Tries to lock a mutex.
- bool chMtxTryLockS (mutex_t *mp)

  Tries to lock a mutex.
- void chMtxUnlock (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockS (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockAllS (void)

  Unlocks all mutexes owned by the invoking thread.
- void chMtxUnlockAll (void)

  Unlocks all mutexes owned by the invoking thread.
- static bool chMtxQueueNotEmptyS (mutex_t *mp)

  Returns `true` if the mutex queue contains at least a waiting thread.
- static mutex_t * chMtxGetNextMutexS (void)

  Returns the next mutex in the mutexes stack of the current thread.

8.13.2 Macro Definition Documentation

8.13.2.1 #define _MUTEX_DATA(*name*) {_THREADS_QUEUE_DATA(name.queue), NULL, NULL, 0}

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

8.13.2.2 #define MUTEX_DECL(*name*) mutex_t 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

8.13.3 Typedef Documentation

8.13.3.1 typedef struct ch_mutex mutex_t

Type of a mutex structure.

8.13.4 Function Documentation

8.13.4.1 void chMtxObjectInit (mutex_t * *mp*)

Initializes s `mutex_t` structure.

Parameters

`out`	*mp*	pointer to a `mutex_t` structure

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:

8.13.4.2 void chMtxLock (mutex_t * *mp*)

Locks the specified mutex.

Postcondition

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

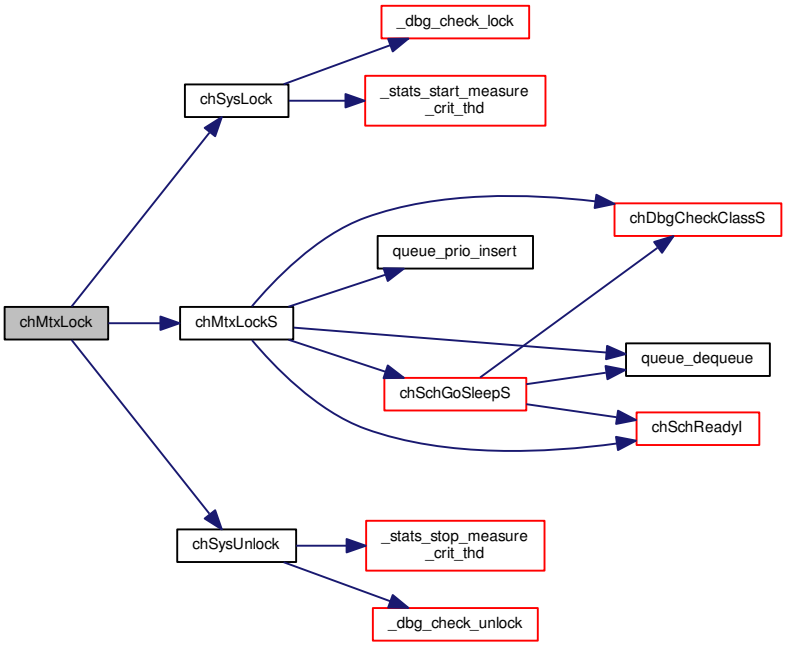
Parameters

`in`	*mp*	pointer to the `mutex_t` 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:



8.13.4.3 void chMtxLockS (mutex_t * *mp*)

Locks the specified mutex.

Postcondition

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

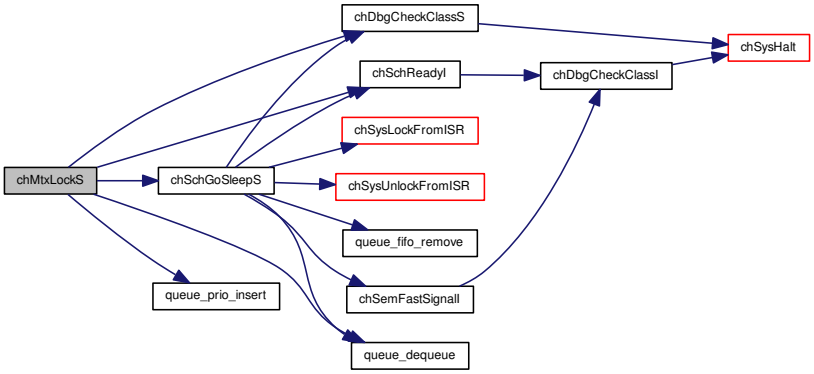
Parameters

`in`	*mp*	pointer to the `mutex_t` 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:



8.13.4.4 bool chMtxTryLock (mutex_t * *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_t` structure

Returns

The operation status.

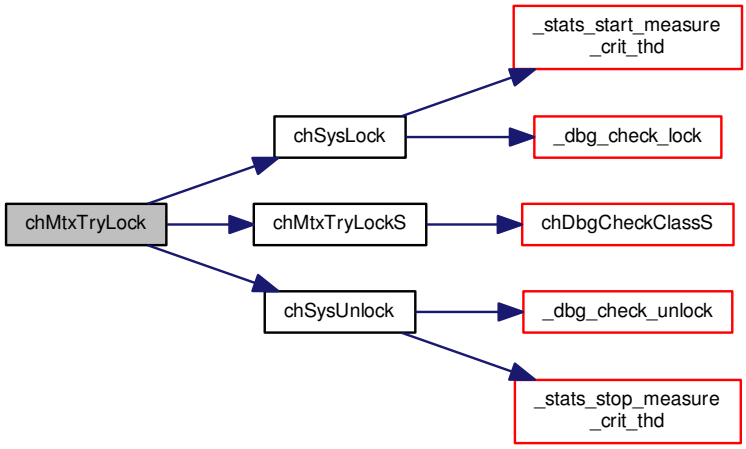
Return values

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:



8.13.4.5 bool chMtxTryLockS (mutex_t * *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_t` structure

Returns

The operation status.

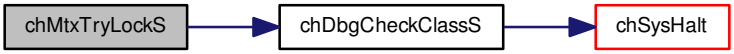
Return values

true	if the mutex has been successfully acquired
false	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:



8.13.4.6 void chMtxUnlock (mutex_t * *mp*)

Unlocks the specified mutex.

Note

Mutexes must be unlocked in reverse lock order. Violating this rules will result in a panic if assertions are enabled.

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.

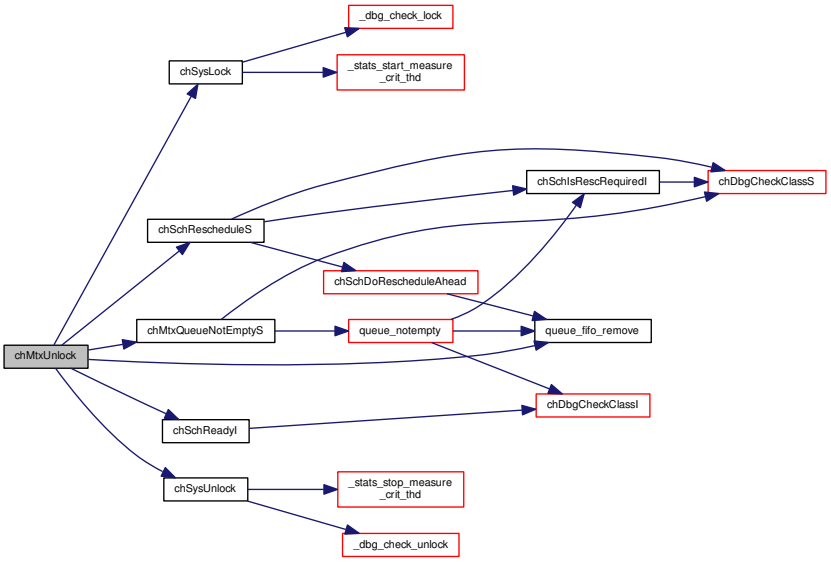
Parameters

`in`	*mp*	pointer to the `mutex_t` 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:



8.13.4.7 void chMtxUnlockS (mutex_t * *mp*)

Unlocks the specified mutex.

Note

Mutexes must be unlocked in reverse lock order. Violating this rules will result in a panic if assertions are enabled.

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.

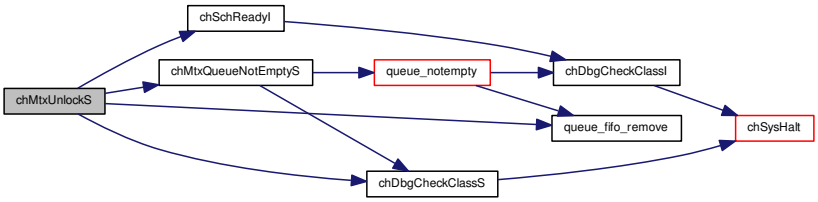
Parameters

`in`	*mp*	pointer to the `mutex_t` 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:



8.13.4.8 void chMtxUnlockAllS (void)

Unlocks all mutexes owned by the invoking thread.

Postcondition

The stack of owned mutexes is emptied and all the found mutexes are unlocked.
This function does not reschedule so a call to a rescheduling function must be performed before unlocking the kernel.

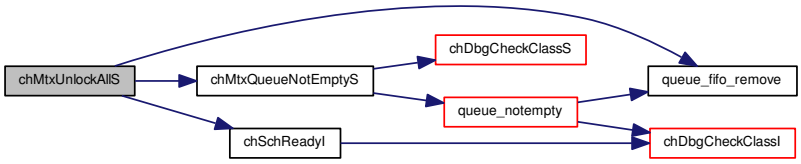
Note

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:

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:



8.13.4.9 void chMtxUnlockAll (void)

Unlocks all mutexes owned by the invoking thread.

Postcondition

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

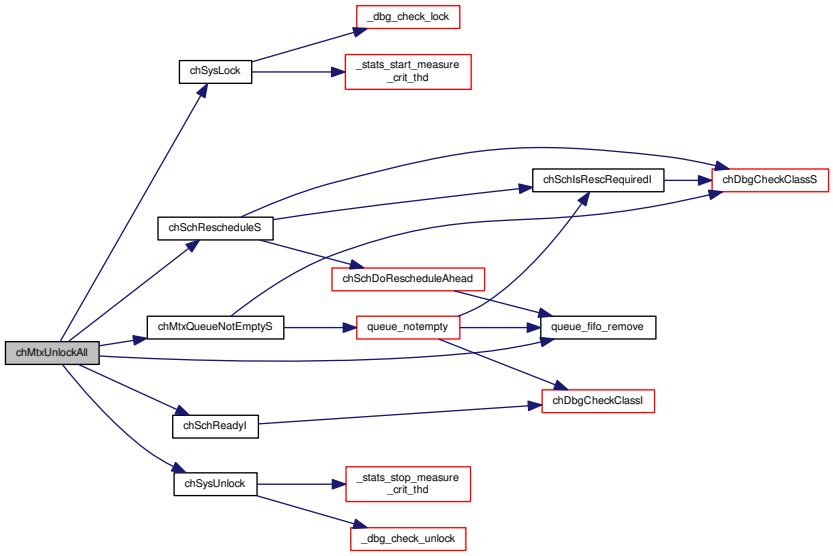
Note

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:



8.13.4.10 static bool chMtxQueueNotEmptyS (mutex_t * *mp*) [inline], [static]

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

Parameters

`out`	*mp*	pointer to a `mutex_t` structure

Returns

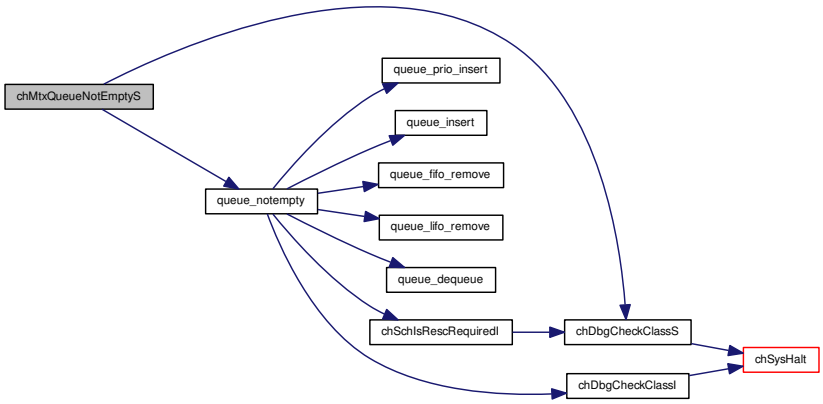
The mutex queue status.

Deprecated

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:



8.13.4.11 static mutex_t* chMtxGetNextMutexS (void) [inline], [static]

Returns the next mutex in the mutexes stack of the current thread.

Returns

A pointer to the next mutex in the stack.

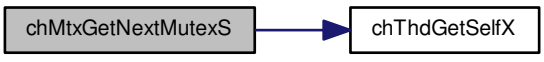
Return values

NULL	if the stack is empty.

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:



8.14 Condition Variables

8.14.1 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 condition variables together can implement a Monitor construct.

Precondition

In order to use the condition variable APIs the `CH_CFG_USE_CONDVARS` option must be enabled in `chconf.h`.

Macros

- #define _CONDVAR_DATA(name) {_THREADS_QUEUE_DATA(name.queue)}

  Data part of a static condition variable initializer.
- #define CONDVAR_DECL(name) condition_variable_t name = _CONDVAR_DATA(name)

  Static condition variable initializer.

Typedefs

- typedef struct condition_variable condition_variable_t

  condition_variable_t structure.

Data Structures

- struct condition_variable

  condition_variable_t structure.

Functions

- void chCondObjectInit (condition_variable_t *cp)

  Initializes s `condition_variable_t` structure.
- void chCondSignal (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondSignalI (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondBroadcast (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- void chCondBroadcastI (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- msg_t chCondWait (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitS (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitTimeout (condition_variable_t *cp, sysinterval_t timeout)

Waits on the condition variable releasing the mutex lock.

- msg_t chCondWaitTimeoutS (condition_variable_t *cp, sysinterval_t timeout)

  Waits on the condition variable releasing the mutex lock.

8.14.2 Macro Definition Documentation

8.14.2.1 #define _CONDVAR_DATA(*name*) {_THREADS_QUEUE_DATA(name.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

8.14.2.2 #define CONDVAR_DECL(*name*) condition_variable_t 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

8.14.3 Typedef Documentation

8.14.3.1 typedef struct condition_variable condition_variable_t

condition_variable_t structure.

8.14.4 Function Documentation

8.14.4.1 void chCondObjectInit (condition_variable_t * *cp*)

Initializes s `condition_variable_t` structure.

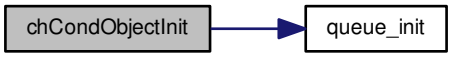
Parameters

out	*cp*	pointer to a `condition_variable_t` structure

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:



8.14.4.2 void chCondSignal (condition_variable_t * *cp*)

Signals one thread that is waiting on the condition variable.

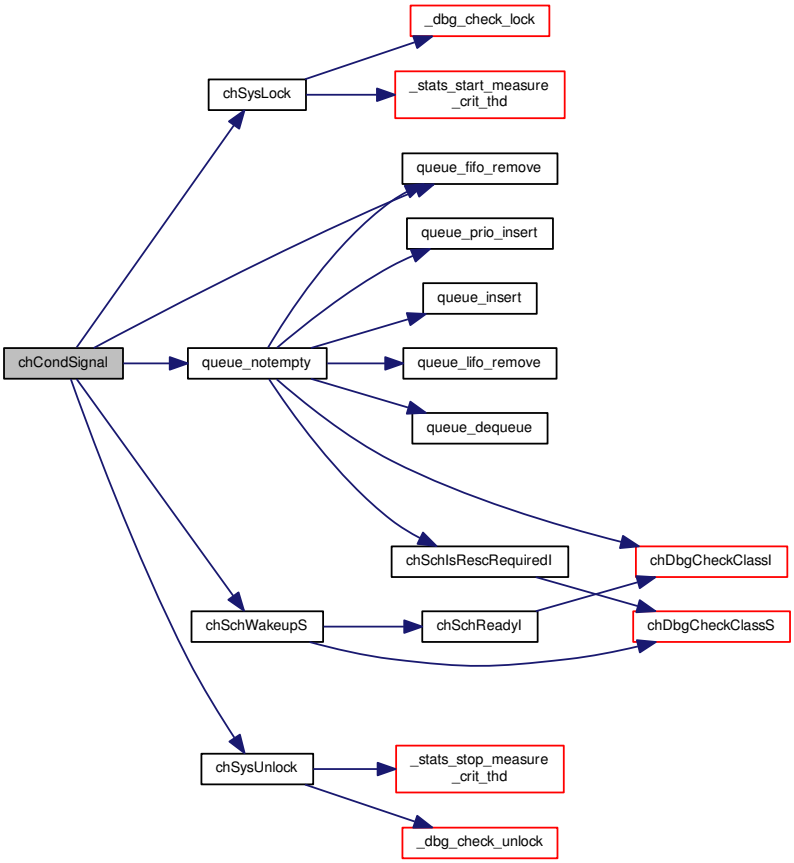
Parameters

in	*cp*	pointer to the `condition_variable_t` 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:



8.14.4.3 void chCondSignalI (condition_variable_t * *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.

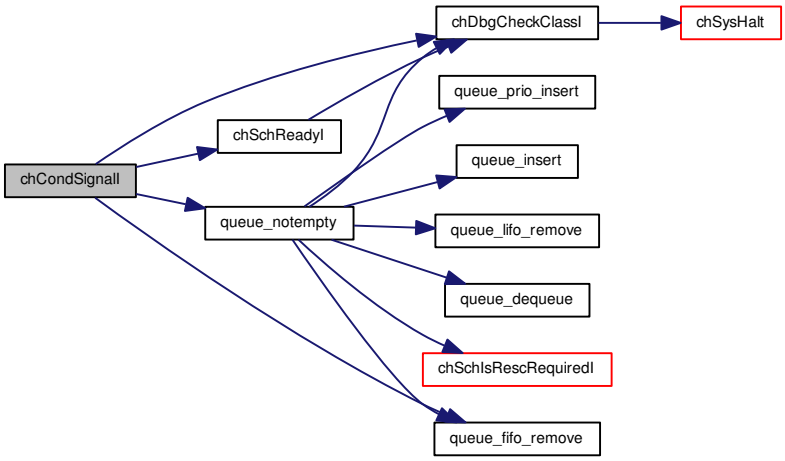
Parameters

in	*cp*	pointer to the `condition_variable_t` 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:



8.14.4.4 void chCondBroadcast (condition_variable_t * *cp*)

Signals all threads that are waiting on the condition variable.

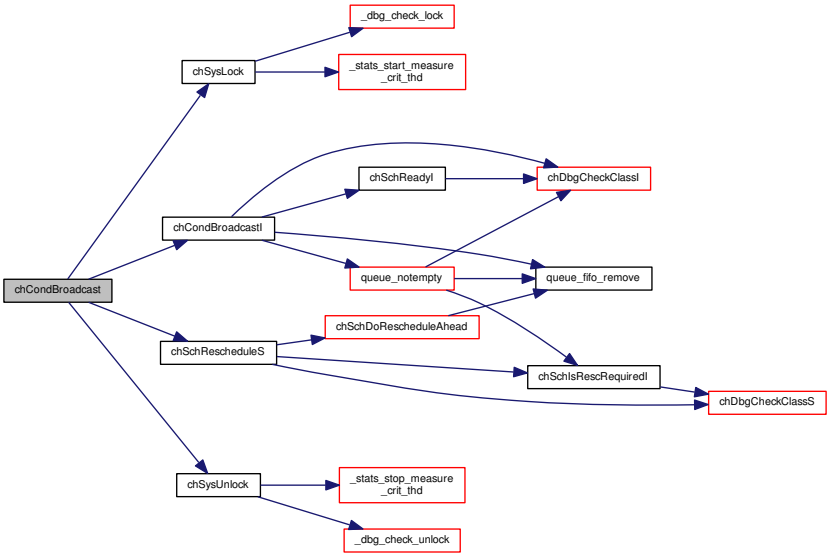
Parameters

in	*cp*	pointer to the condition_variable_t 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:



8.14.4.5 void chCondBroadcastI (condition_variable_t * *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.

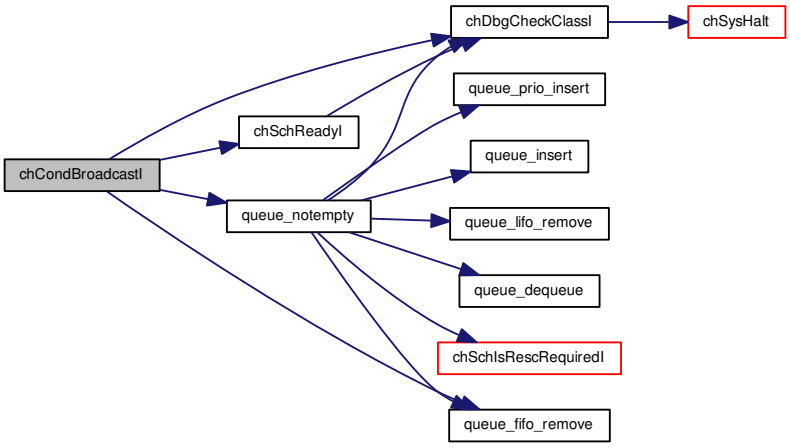
Parameters

in	*cp*	pointer to the `condition_variable_t` 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:



8.14.4.6 msg_t chCondWait (condition_variable_t * *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`	*cp*	pointer to the `condition_variable_t` structure

Returns

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

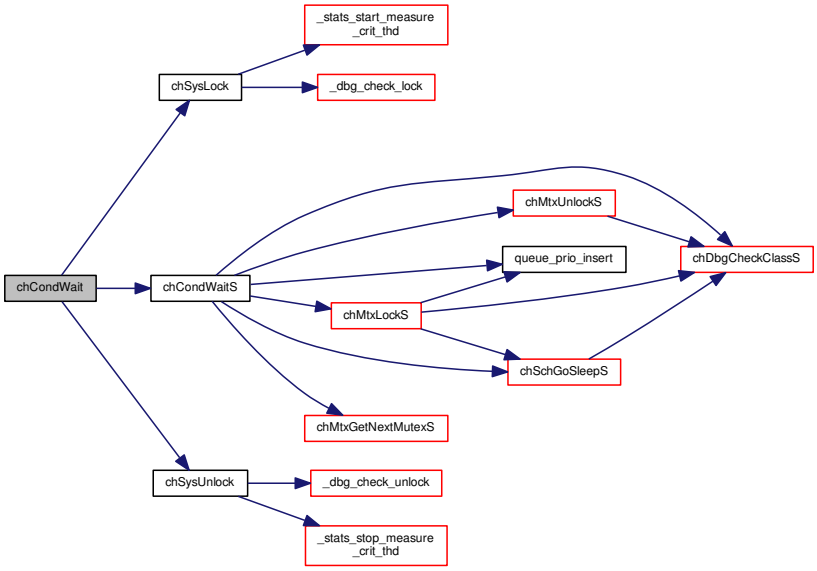
Return values

MSG_OK	if the condition variable has been signaled using `chCondSignal()`.
MSG_RESET	if the condition variable has been signaled using `chCondBroadcast()`.

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:



8.14.4.7 msg_t chCondWaitS (condition_variable_t * *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	*cp*	pointer to the `condition_variable_t` structure

Returns

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

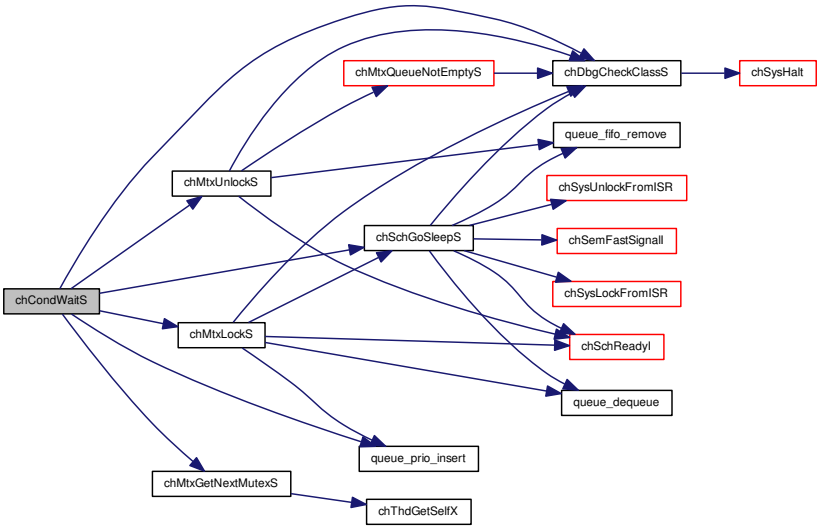
Return values

MSG_OK	if the condition variable has been signaled using `chCondSignal()`.
MSG_RESET	if the condition variable 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:



8.14.4.8 msg_t chCondWaitTimeout (condition_variable_t * *cp*, sysinterval_t *timeout*)

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_CFG_USE_CONDVARS_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`	*cp*	pointer to the `condition_variable_t` structure
`in`	*timeout*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* no timeout. • *TIME_IMMEDIATE* this value is not allowed.

Returns

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

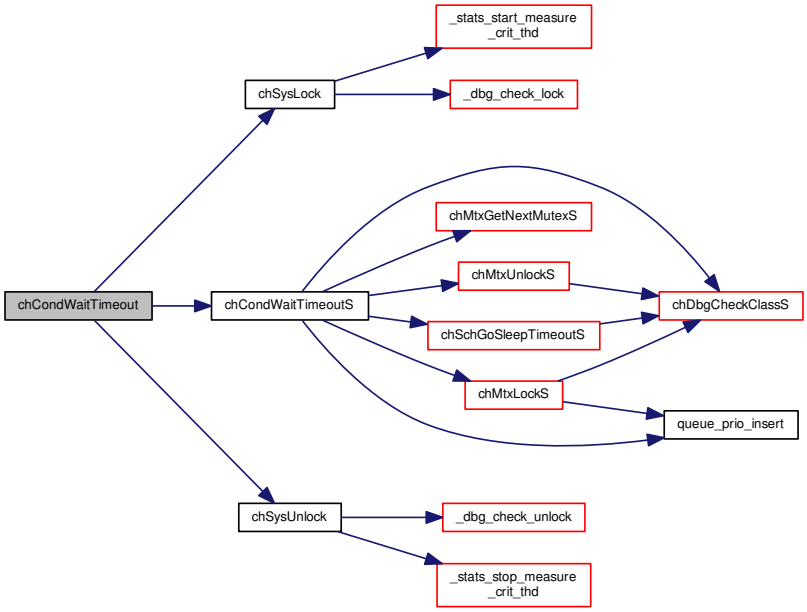
Return values

MSG_OK	if the condition variable has been signaled using `chCondSignal()`.
MSG_RESET	if the condition variable has been signaled using `chCondBroadcast()`.
MSG_TIMEOUT	if the condition variable 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:



8.14.4.9 msg_t chCondWaitTimeoutS (condition_variable_t * *cp*, sysinterval_t *timeout*)

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_CFG_USE_CONDVARS_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	*cp*	pointer to the condition_variable_t structure
in	*timeout*	the number of ticks before the operation timeouts, the special values are handled as follow: • *TIME_INFINITE* no timeout. • *TIME_IMMEDIATE* this value is not allowed.

Returns

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

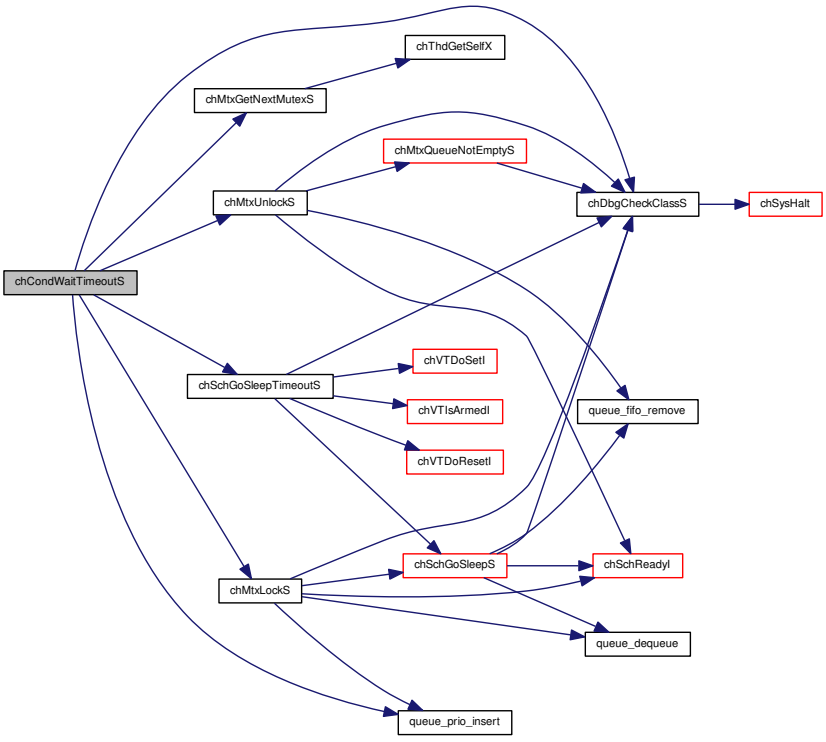
Return values

MSG_OK	if the condition variable has been signaled using chCondSignal().
MSG_RESET	if the condition variable has been signaled using chCondBroadcast().
MSG_TIMEOUT	if the condition variable 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:



8.15 Event Flags

8.15.1 Detailed Description

Event Flags, Event Sources and Event Listeners.

Operation mode

Each thread has a mask of pending events inside its `thread_t` structure. Operations defined for events:

- **Wait**, the invoking thread goes to sleep until a certain AND/OR combination of events become pending.
- **Clear**, a mask of events is cleared from the pending events, the cleared events mask is returned (only the events that were actually pending and then cleared).
- **Signal**, an events mask is directly ORed to the mask of the signaled thread.
- **Broadcast**, each thread registered on an Event Source is signaled with the events 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_CFG_USE_EVENTS` option must be enabled in `chconf.h`.

Postcondition

Enabling events requires 1-4 (depending on the architecture) extra bytes in the `thread_t` structure.

Macros

- #define ALL_EVENTS ((eventmask_t)-1)

  All events allowed mask.
- #define EVENT_MASK(eid) ((eventmask_t)1 << (eventmask_t)(eid))

  Returns an event mask from an event identifier.
- #define _EVENTSOURCE_DATA(name) {(event_listener_t *)(&name)}

  Data part of a static event source initializer.
- #define EVENTSOURCE_DECL(name) event_source_t name = _EVENTSOURCE_DATA(name)

  Static event source initializer.

Typedefs

- typedef struct event_source event_source_t

  Event Source structure.
- typedef void(* evhandler_t) (eventid_t id)

  Event Handler callback function.

Data Structures

- struct event_listener

  Event Listener structure.
- struct event_source

  Event Source structure.

Functions

- void chEvtRegisterMaskWithFlags (event_source_t *esp, event_listener_t *elp, eventmask_t events, eventflags_t wflags)

  Registers an Event Listener on an Event Source.
- void chEvtUnregister (event_source_t *esp, event_listener_t *elp)

  Unregisters an Event Listener from its Event Source.
- eventmask_t chEvtGetAndClearEventsI (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtGetAndClearEvents (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtAddEvents (eventmask_t events)

  *Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvt↩Signal()`.*
- void chEvtBroadcastFlagsI (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- eventflags_t chEvtGetAndClearFlags (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- void chEvtSignal (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtSignalI (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtBroadcastFlags (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- eventflags_t chEvtGetAndClearFlagsI (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- void chEvtDispatch (const evhandler_t *handlers, eventmask_t events)

  Invokes the event handlers associated to an event flags mask.
- eventmask_t chEvtWaitOne (eventmask_t events)

  Waits for exactly one of the specified events.
- eventmask_t chEvtWaitAny (eventmask_t events)

  Waits for any of the specified events.
- eventmask_t chEvtWaitAll (eventmask_t events)

  Waits for all the specified events.
- eventmask_t chEvtWaitOneTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for exactly one of the specified events.
- eventmask_t chEvtWaitAnyTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for any of the specified events.
- eventmask_t chEvtWaitAllTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for all the specified events.
- static void chEvtObjectInit (event_source_t *esp)

  Initializes an Event Source.
- static void chEvtRegisterMask (event_source_t *esp, event_listener_t *elp, eventmask_t events)

  Registers an Event Listener on an Event Source.

- static void chEvtRegister (event_source_t *esp, event_listener_t *elp, eventid_t event)

  Registers an Event Listener on an Event Source.
- static bool chEvtIsListeningI (event_source_t *esp)

  Verifies if there is at least one `event_listener_t` registered.
- static void chEvtBroadcast (event_source_t *esp)

  Signals all the Event Listeners registered on the specified Event Source.
- static void chEvtBroadcastI (event_source_t *esp)

  Signals all the Event Listeners registered on the specified Event Source.
- static eventmask_t chEvtAddEventsI (eventmask_t events)

  *Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvtSignal()`.*
- static eventmask_t chEvtGetEventsX (void)

  Returns the events mask.

8.15.2 Macro Definition Documentation

8.15.2.1 #define ALL_EVENTS ((eventmask_t)-1)

All events allowed mask.

8.15.2.2 #define EVENT_MASK(*eid*) ((eventmask_t)1 << (eventmask_t)(eid))

Returns an event mask from an event identifier.

8.15.2.3 #define _EVENTSOURCE_DATA(*name*) {(event_listener_t *)(&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

8.15.2.4 #define EVENTSOURCE_DECL(*name*) event_source_t 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

8.15.3 Typedef Documentation

8.15.3.1 typedef struct event_source event_source_t

Event Source structure.

8.15.3.2 typedef void(∗ evhandler_t) (eventid_t id)

Event Handler callback function.

8.15.4 Function Documentation

8.15.4.1 void chEvtRegisterMaskWithFlags (event_source_t ∗ *esp*, event_listener_t ∗ *elp*, eventmask_t *events*, eventflags_t *wflags*)

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.

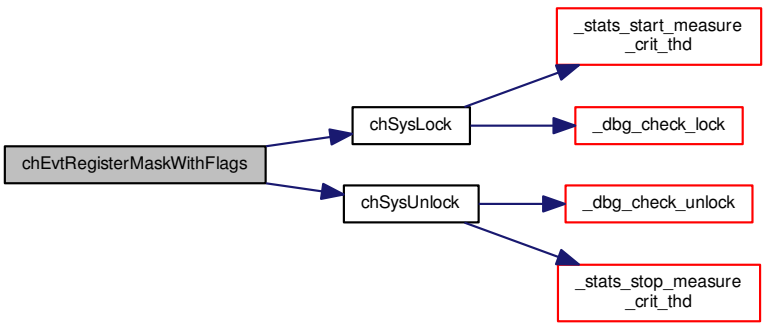
Parameters

in	*esp*	pointer to the `event_source_t` structure
in	*elp*	pointer to the `event_listener_t` structure
in	*events*	events to be ORed to the thread when the event source is broadcasted
in	*wflags*	mask of flags the listening thread is interested in

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:



8.15.4.2 void chEvtUnregister (event_source_t ∗ *esp*, event_listener_t ∗ *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).

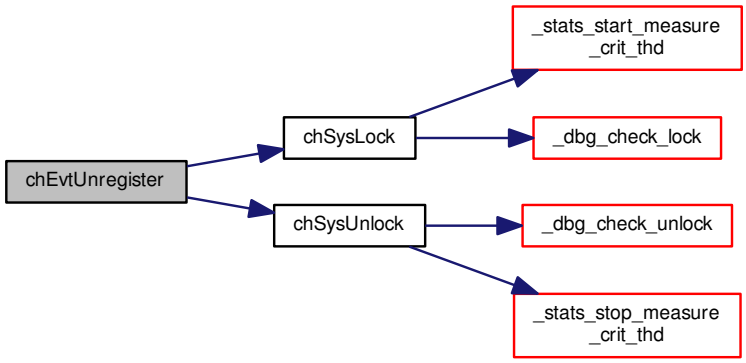
Parameters

in	*esp*	pointer to the `event_source_t` structure
in	*elp*	pointer to the `event_listener_t` 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:



8.15.4.3 eventmask_t chEvtGetAndClearEventsI (eventmask_t *events*)

Clears the pending events specified in the events mask.

Parameters

in	*events*	the events to be cleared

Returns

The mask of pending events that were cleared.

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.

8.15.4.4 eventmask_t chEvtGetAndClearEvents (eventmask_t *events*)

Clears the pending events specified in the events mask.

Parameters

in	*events*	the events to be cleared

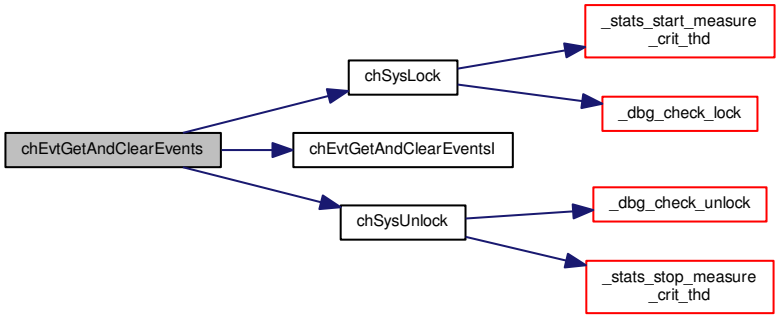
Returns

The mask of 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.

Here is the call graph for this function:



8.15.4.5 eventmask_t chEvtAddEvents (eventmask_t *events*)

Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvtSignal()`.

Parameters

in	*events*	the events to be added

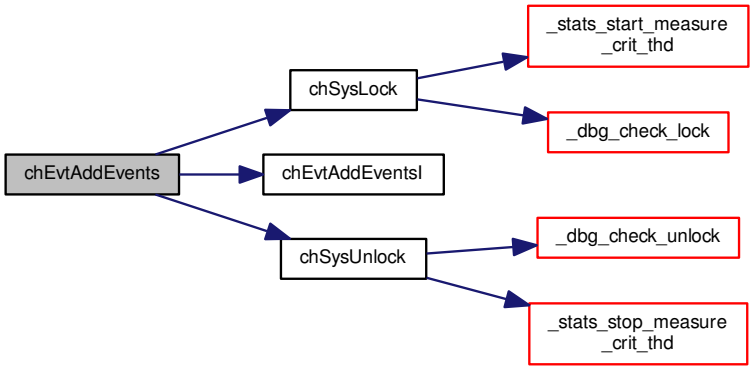
Returns

The mask of currently pending 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:



8.15.4.6 void chEvtBroadcastFlagsI (event_source_t * *esp*, eventflags_t *flags*)

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 `event_source_t` in addition to the event flags specified by the threads themselves in the `event_listener_t` 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.

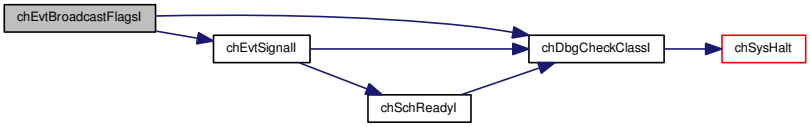
Parameters

`in`	*esp*	pointer to the `event_source_t` structure
`in`	*flags*	the flags set to be added to the listener flags 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.

Here is the call graph for this function:



8.15.4.7 eventflags_t chEvtGetAndClearFlags (event_listener_t * *elp*)

Returns the flags associated to an `event_listener_t`.

The flags are returned and the `event_listener_t` flags mask is cleared.

Parameters

`in`	*elp*	pointer to the `event_listener_t` structure

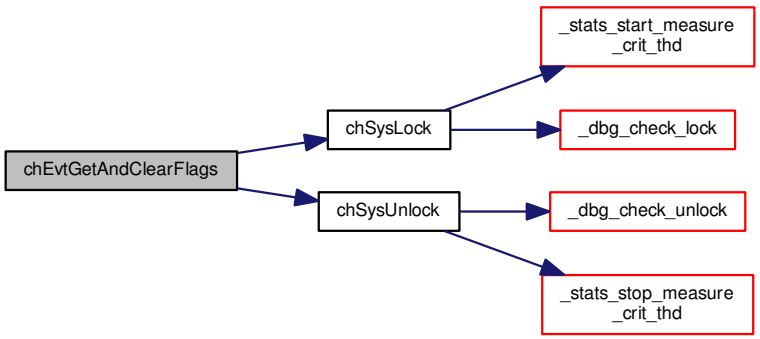
Returns

The flags added to the listener by the associated event source.

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:



8.15.4.8 void chEvtSignal (thread_t * *tp*, eventmask_t *events*)

Adds a set of event flags directly to the specified `thread_t`.

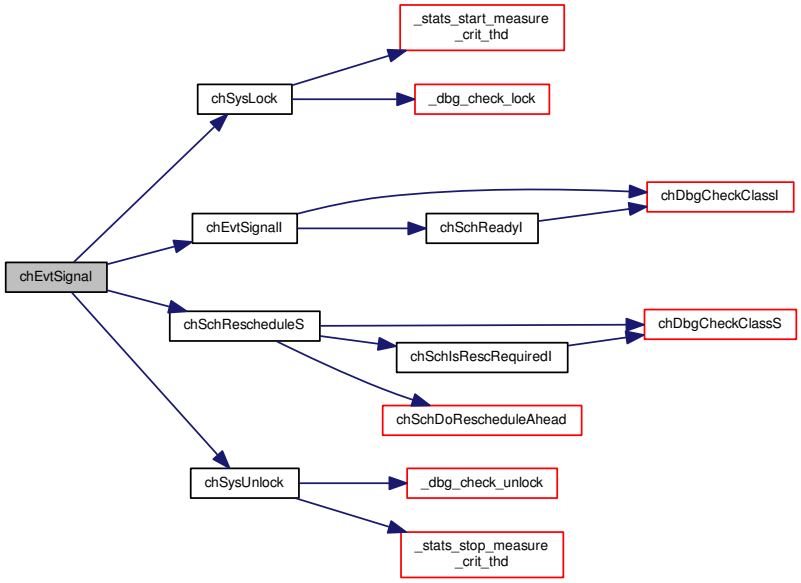
Parameters

in	*tp*	the thread to be signaled
in	*events*	the events 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:



8.15.4.9 void chEvtSignalI (thread_t * *tp*, eventmask_t *events*)

Adds a set of event flags directly to the specified `thread_t`.

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.

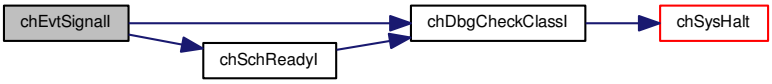
Parameters

in	*tp*	the thread to be signaled
in	*events*	the events 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:



8.15.4.10 void chEvtBroadcastFlags (event_source_t * *esp*, eventflags_t *flags*)

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 `event_source_t` in addition to the event flags specified by the threads themselves in the `event_listener_t` objects.

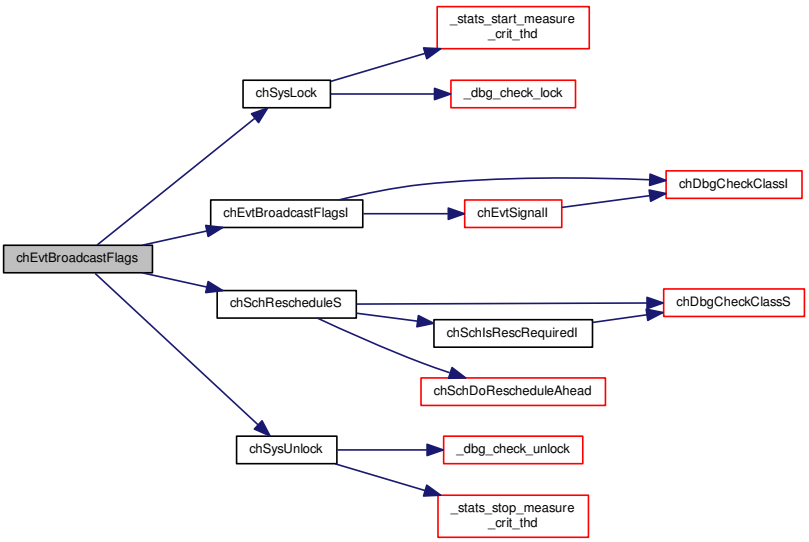
Parameters

`in`	*esp*	pointer to the `event_source_t` structure
`in`	*flags*	the flags set to be added to the listener flags mask

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:



8.15.4.11 eventflags_t chEvtGetAndClearFlagsI (event_listener_t * *elp*)

Returns the flags associated to an `event_listener_t`.

The flags are returned and the event_listener_t flags mask is cleared.

Parameters

in	*elp*	pointer to the event_listener_t structure

Returns

The flags added to the listener by the associated event source.

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.

8.15.4.12 void chEvtDispatch (const evhandler_t * *handlers*, eventmask_t *events*)

Invokes the event handlers associated to an event flags mask.

Parameters

in	*events*	mask of events to be dispatched
in	*handlers*	an array of evhandler_t. The array must have size equal to the number of bits in eventmask_t.

Function Class:

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

8.15.4.13 eventmask_t chEvtWaitOne (eventmask_t *events*)

Waits for exactly one of the specified events.

The function waits for one event among those specified in events 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	*events*	events that the function should wait for, ALL_EVENTS enables all the events

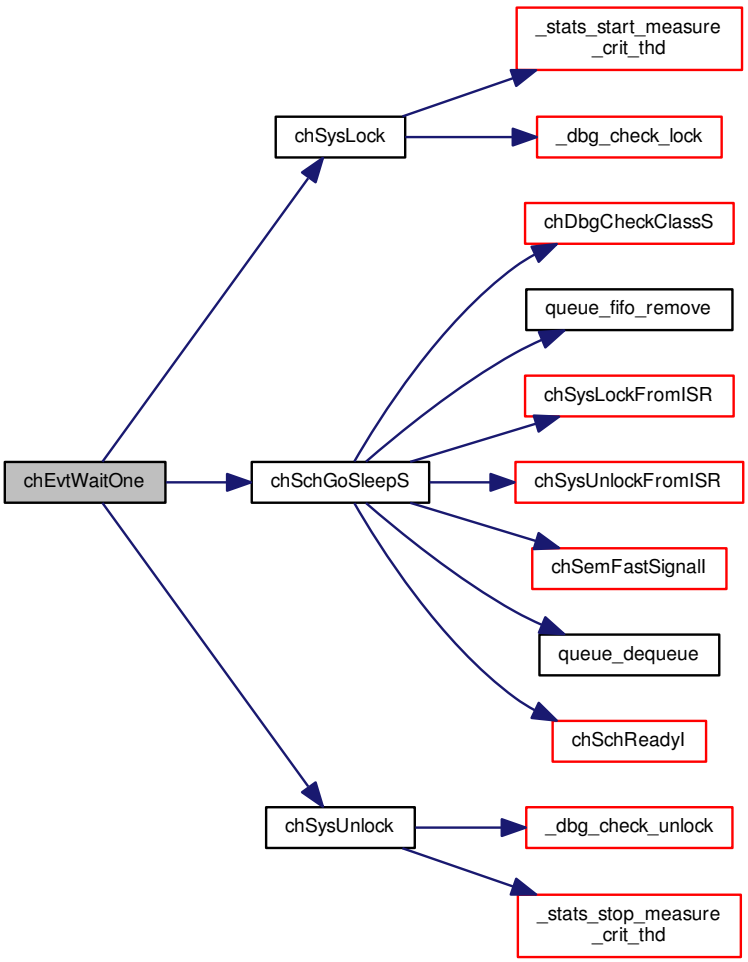
Returns

The mask of the lowest event id served and cleared.

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:



8.15.4.14 eventmask_t chEvtWaitAny (eventmask_t *events*)

Waits for any of the specified events.

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

Parameters

`in`	*events*	events that the function should wait for, `ALL_EVENTS` enables all the events

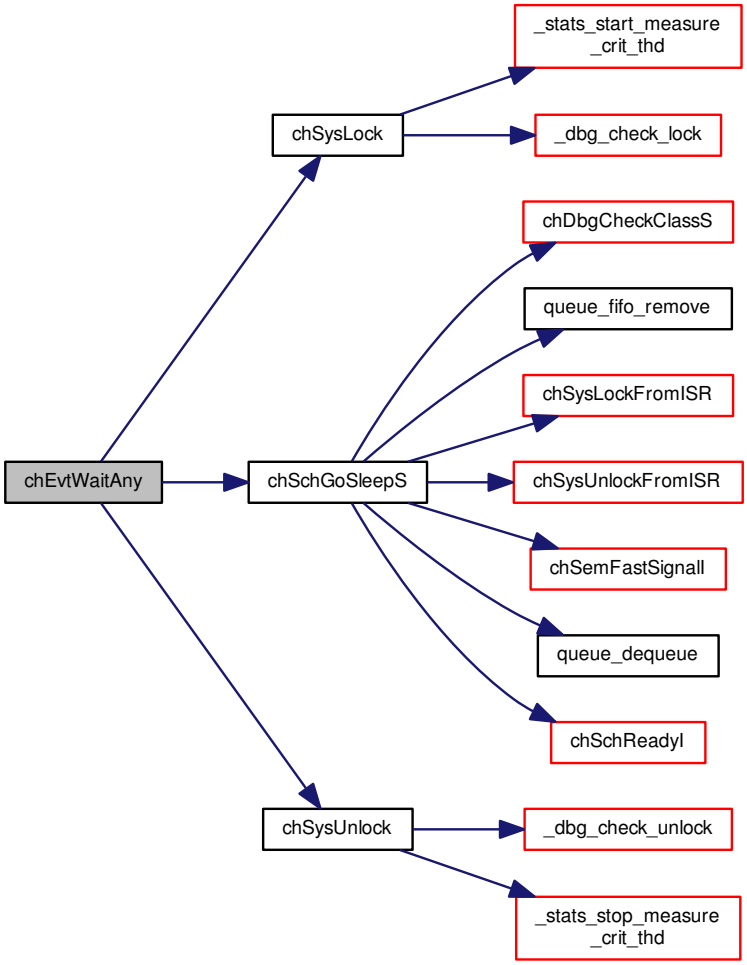
Returns

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:



8.15.4.15 eventmask_t chEvtWaitAll (eventmask_t *events*)

Waits for all the specified events.

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

Parameters

`in`	*events*	events that the function should wait for, `ALL_EVENTS` requires all the events

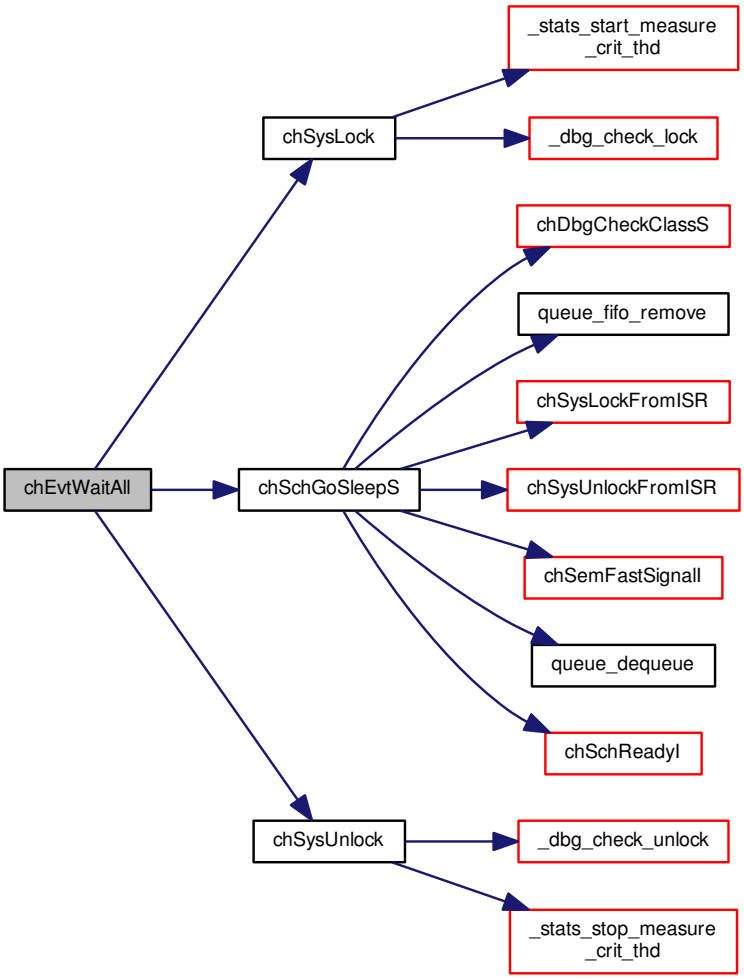
Returns

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:



8.15.4.16 eventmask_t chEvtWaitOneTimeout (eventmask_t *events*, sysinterval_t *timeout*)

Waits for exactly one of the specified events.

The function waits for one event among those specified in `events` 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`	*events*	events that the function should wait for, `ALL_EVENTS` enables all the events

Parameters

in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The mask of the lowest event id served and cleared.

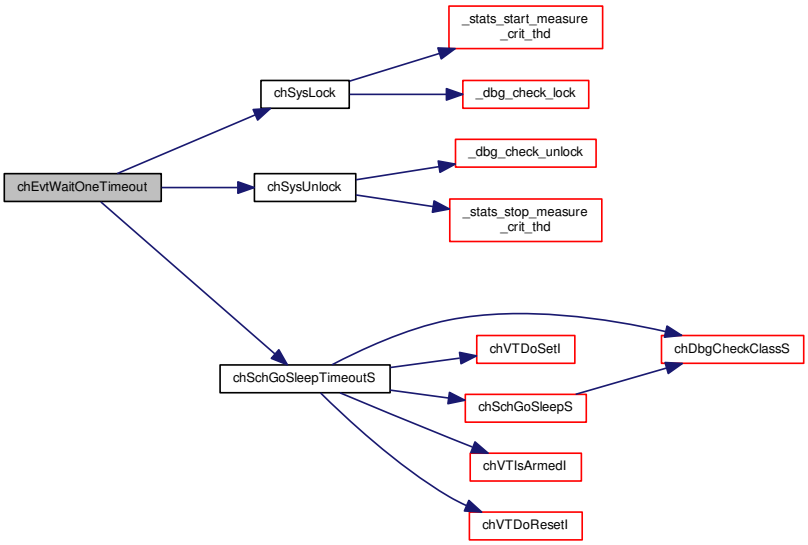
Return values

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:



8.15.4.17 eventmask_t chEvtWaitAnyTimeout (eventmask_t *events*, sysinterval_t *timeout*)

Waits for any of the specified events.

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

Parameters

in	*events*	events that the function should wait for, `ALL_EVENTS` enables all the events

Parameters

in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The mask of the served and cleared events.

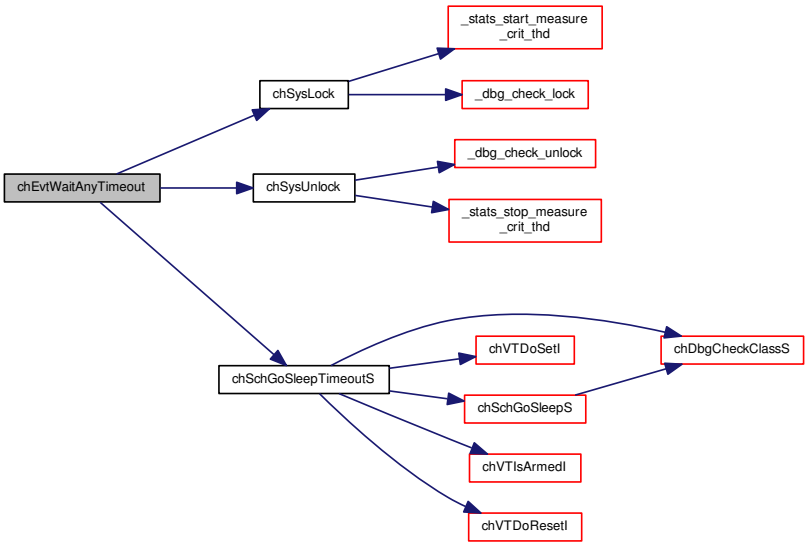
Return values

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:



8.15.4.18 eventmask_t chEvtWaitAllTimeout (eventmask_t *events*, sysinterval_t *timeout*)

Waits for all the specified events.

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

Parameters

in	*events*	events that the function should wait for, `ALL_EVENTS` requires all the events

Parameters

in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The mask of the served and cleared events.

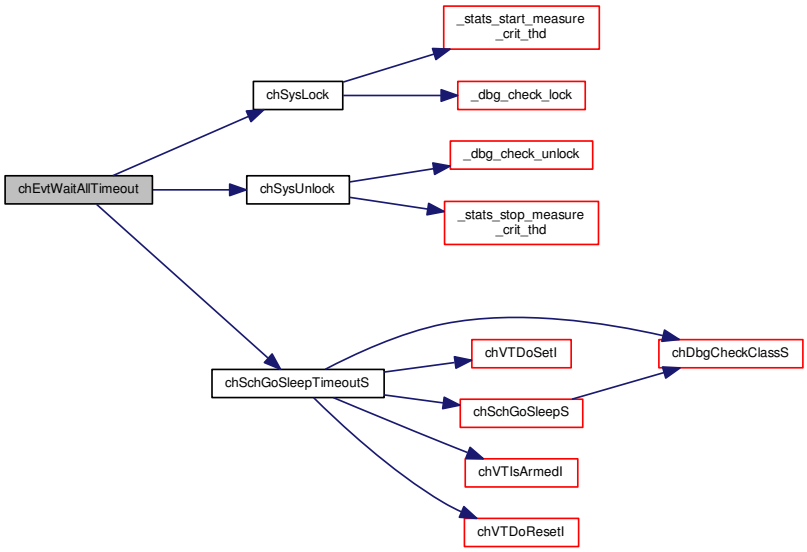
Return values

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:



8.15.4.19 static void chEvtObjectInit (event_source_t * *esp*) [inline], [static]

Initializes an Event Source.

Note

This function can be invoked before the kernel is initialized because it just prepares a `event_source_t` structure.

Parameters

in	*esp*	pointer to the event_source_t structure

Function Class:

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

8.15.4.20 static void chEvtRegisterMask (event_source_t * *esp*, event_listener_t * *elp*, eventmask_t *events*) [inline], [static]

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	*esp*	pointer to the event_source_t structure
out	*elp*	pointer to the event_listener_t structure
in	*events*	the mask of events 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.

Here is the call graph for this function:

8.15.4.21 static void chEvtRegister (event_source_t * *esp*, event_listener_t * *elp*, eventid_t *event*) [inline], [static]

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.

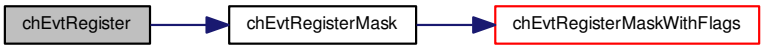
Parameters

in	*esp*	pointer to the event_source_t structure
out	*elp*	pointer to the event_listener_t structure
in	*event*	numeric identifier assigned to the Event Listener. The value must range between zero and the size, in bit, of the eventmask_t type minus one.

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:



8.15.4.22 static bool chEvtIsListeningI (event_source_t * *esp*) [inline], [static]

Verifies if there is at least one event_listener_t registered.

Parameters

in	*esp*	pointer to the event_source_t structure

Returns

The event source status.

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.

8.15.4.23 static void chEvtBroadcast (event_source_t * *esp*) [inline], [static]

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

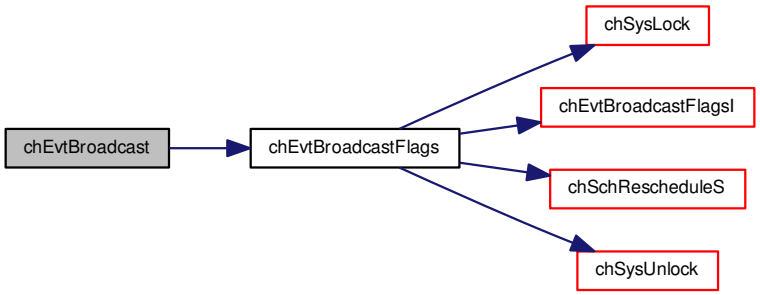
Parameters

in	*esp*	pointer to the event_source_t 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:



8.15.4.24 static void chEvtBroadcastI (event_source_t * *esp*) [inline], [static]

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.

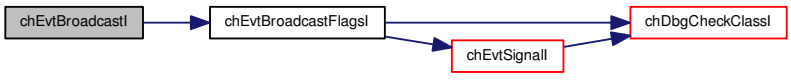
Parameters

in	*esp*	pointer to the event_source_t 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:



8.15.4.25 static eventmask_t chEvtAddEventsI (eventmask_t *events*) [inline], [static]

Adds (OR) a set of events to the current thread, this is **much** faster than using chEvtBroadcast() or chEvt↩Signal().

Parameters

in	*events*	the events to be added

Returns

The mask of currently pending events.

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.

8.15.4.26 static eventmask_t chEvtGetEventsX (void) [inline], [static]

Returns the events mask.

The pending events mask is returned but not altered in any way.

Returns

The pending events mask.

Function Class:

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

8.16 Synchronous Messages

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

Precondition

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

Postcondition

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

Functions

- msg_t chMsgSend (thread_t *tp, msg_t msg)

  Sends a message to the specified thread.
- thread_t * chMsgWait (void)

  Suspends the thread and waits for an incoming message.
- void chMsgRelease (thread_t *tp, msg_t msg)

  Releases a sender thread specifying a response message.
- static bool chMsgIsPendingI (thread_t *tp)

  Evaluates to `true` if the thread has pending messages.
- static msg_t chMsgGet (thread_t *tp)

  Returns the message carried by the specified thread.
- static void chMsgReleaseS (thread_t *tp, msg_t msg)

  Releases the thread waiting on top of the messages queue.

8.16.2 Function Documentation

8.16.2.1 msg_t chMsgSend (thread_t * *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`	*tp*	the pointer to the thread
`in`	*msg*	the message

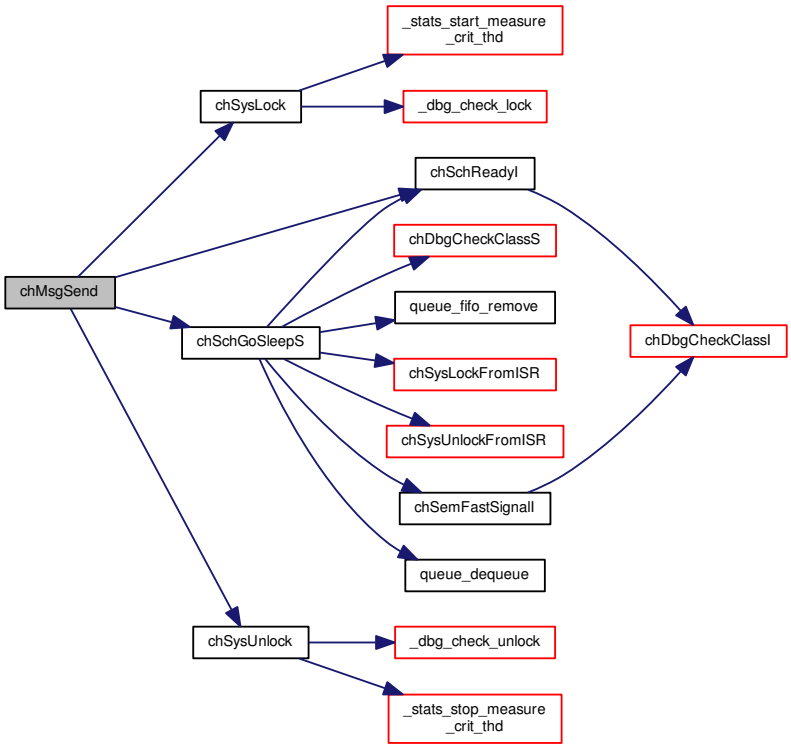
Returns

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:



8.16.2.2 thread_t * 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.

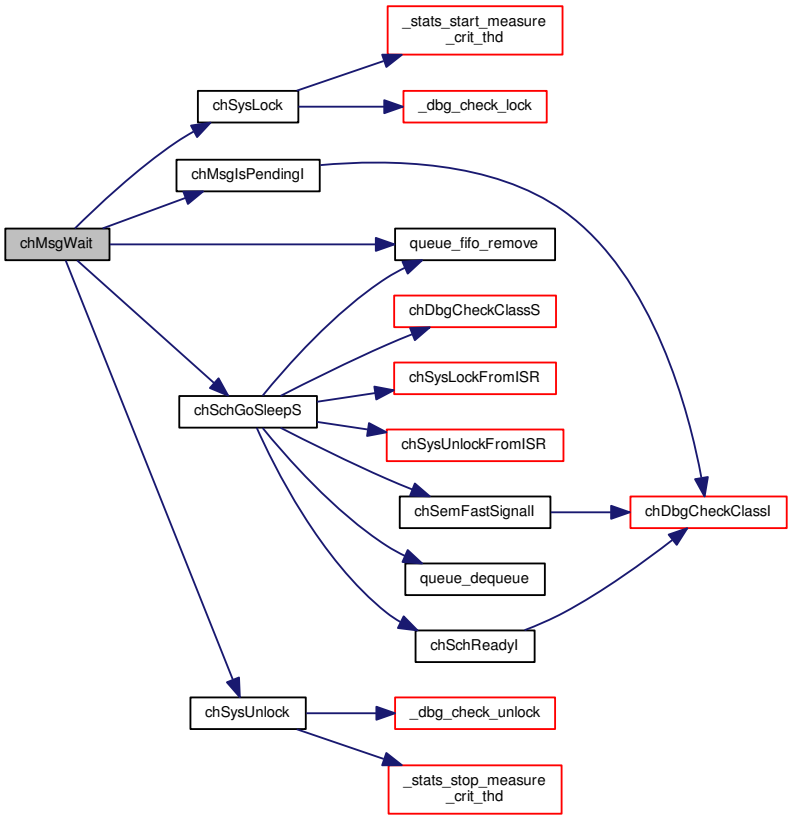
Returns

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:



8.16.2.3 void chMsgRelease (thread_t * *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()`.

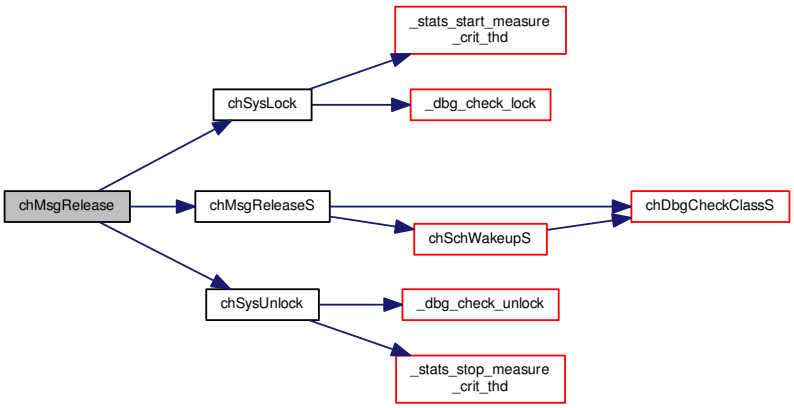
Parameters

in	*tp*	pointer to the thread
in	*msg*	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.

Here is the call graph for this function:



8.16.2.4 static bool chMsgIsPendingI (thread_t * *tp*) [inline], [static]

Evaluates to `true` if the thread has pending messages.

Parameters

`in`	*tp*	pointer to the thread

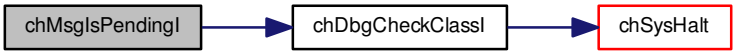
Returns

The pending messages status.

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:



8.16.2.5 static msg_t chMsgGet (thread_t * *tp*) [inline], [static]

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.

8.16.2.6 static void chMsgReleaseS (thread_t * *tp*, msg_t *msg*) [inline], [static]

Releases the thread waiting on top of the messages queue.

Precondition

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

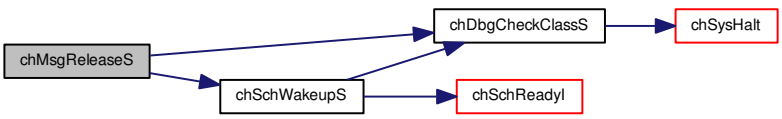
Parameters

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.

Here is the call graph for this function:



8.17 Mailboxes

8.17.1 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 for 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_CFG_USE_MAILBOXES` option must be enabled in `chconf.h`.

Note

Compatible with RT and NIL.

Macros

- #define _MAILBOX_DATA(name, buffer, size)

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

  Static mailbox initializer.

Data Structures

- struct mailbox_t

  Structure representing a mailbox object.

Functions

- void chMBObjectInit (mailbox_t *mbp, msg_t *buf, size_t n)

  Initializes a `mailbox_t` object.
- void chMBReset (mailbox_t *mbp)

  Resets a `mailbox_t` object.
- void chMBResetI (mailbox_t *mbp)

  Resets a `mailbox_t` object.
- msg_t chMBPostTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

Posts a message into a mailbox.

- msg_t chMBPostTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts a message into a mailbox.
- msg_t chMBPostI (mailbox_t *mbp, msg_t msg)

  Posts a message into a mailbox.
- msg_t chMBPostAheadTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadI (mailbox_t *mbp, msg_t msg)

  Posts an high priority message into a mailbox.
- msg_t chMBFetchTimeout (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)

  Retrieves a message from a mailbox.
- msg_t chMBFetchTimeoutS (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)

  Retrieves a message from a mailbox.
- msg_t chMBFetchI (mailbox_t *mbp, msg_t *msgp)

  Retrieves a message from a mailbox.
- static size_t chMBGetSizeI (const mailbox_t *mbp)

  Returns the mailbox buffer size as number of messages.
- static size_t chMBGetUsedCountI (const mailbox_t *mbp)

  Returns the number of used message slots into a mailbox.
- static size_t chMBGetFreeCountI (const mailbox_t *mbp)

  Returns the number of free message slots into a mailbox.
- static msg_t chMBPeekI (const mailbox_t *mbp)

  Returns the next message in the queue without removing it.
- static void chMBResumeX (mailbox_t *mbp)

  Terminates the reset state.

8.17.2 Macro Definition Documentation

8.17.2.1 #define _MAILBOX_DATA(*name, buffer, size*)

Value:

```
{                                   \
  (msg_t *)(buffer),                                                \
  (msg_t *)(buffer) + size,                                         \
  (msg_t *)(buffer),                                                \
  (msg_t *)(buffer),                                                \
  (size_t)0,                                                        \
  false,                                                            \
  _THREADS_QUEUE_DATA(name.qw),
      \
  _THREADS_QUEUE_DATA(name.qr),                                     \
}
```

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	*name*	the name of the mailbox variable
in	*buffer*	pointer to the mailbox buffer array of msg_t
in	*size*	number of msg_t elements in the buffer array

8.17.2.2 #define MAILBOX_DECL(*name, buffer, size*) mailbox_t name = _MAILBOX_DATA(name, buffer, size)

Static mailbox initializer.

Statically initialized mailboxes require no explicit initialization using `chMBObjectInit()`.

Parameters

in	*name*	the name of the mailbox variable
in	*buffer*	pointer to the mailbox buffer array of `msg_t`
in	*size*	number of `msg_t` elements in the buffer array

8.17.3 Function Documentation

8.17.3.1 void chMBObjectInit (mailbox_t * *mbp*, msg_t * *buf*, size_t *n*)

Initializes a `mailbox_t` object.

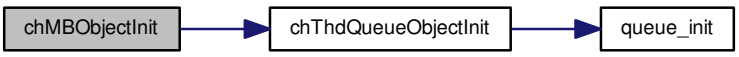
Parameters

out	*mbp*	the pointer to the `mailbox_t` structure to be initialized
in	*buf*	pointer to the messages buffer as an array of `msg_t`
in	*n*	number of elements in the buffer array

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:



8.17.3.2 void chMBReset (mailbox_t * *mbp*)

Resets a `mailbox_t` object.

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

Postcondition

The mailbox is in reset state, all operations will fail and return `MSG_RESET` until the mailbox is enabled again using `chMBResumeX()`.

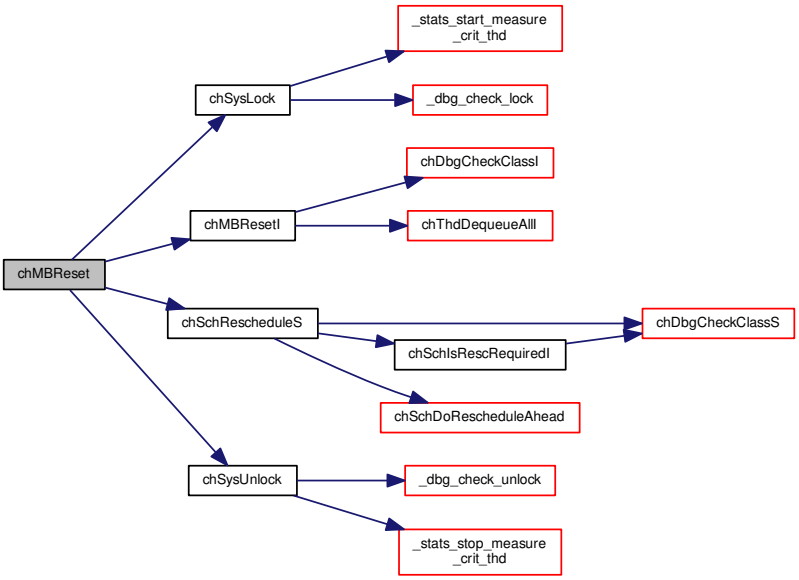
Parameters

in	*mbp*	the pointer to an initialized `mailbox_t` 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:



8.17.3.3 void chMBResetI (mailbox_t * *mbp*)

Resets a `mailbox_t` object.

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

Postcondition

The mailbox is in reset state, all operations will fail and return `MSG_RESET` until the mailbox is enabled again using `chMBResumeX()`.

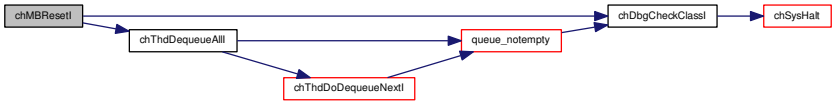
Parameters

`in`	*mbp*	the pointer to an initialized `mailbox_t` 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:



8.17.3.4 msg_t chMBPostTimeout (mailbox_t * *mbp*, msg_t *msg*, sysinterval_t *timeout*)

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_t` object
in	*msg*	the message to be posted on the mailbox
in	*timeout*	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.

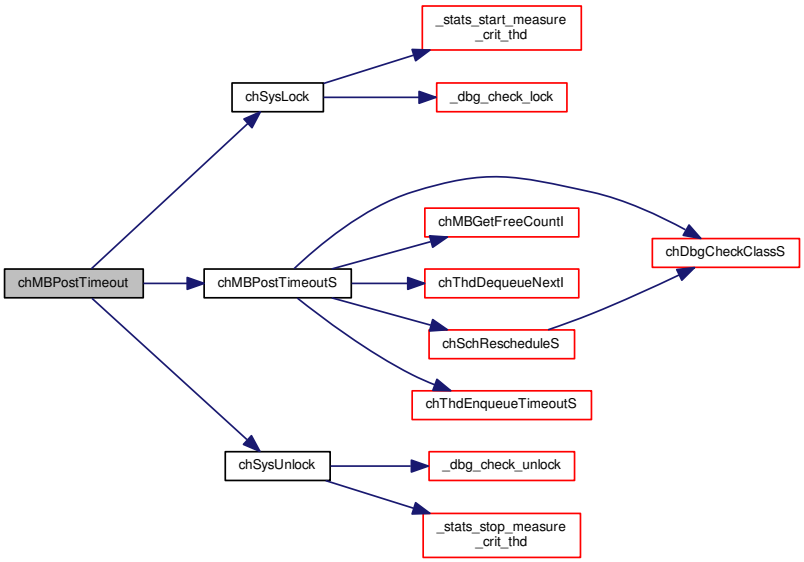
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.5 msg_t chMBPostTimeoutS (mailbox_t * *mbp*, msg_t *msg*, sysinterval_t *timeout*)

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_t` object
in	*msg*	the message to be posted on the mailbox
in	*timeout*	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.

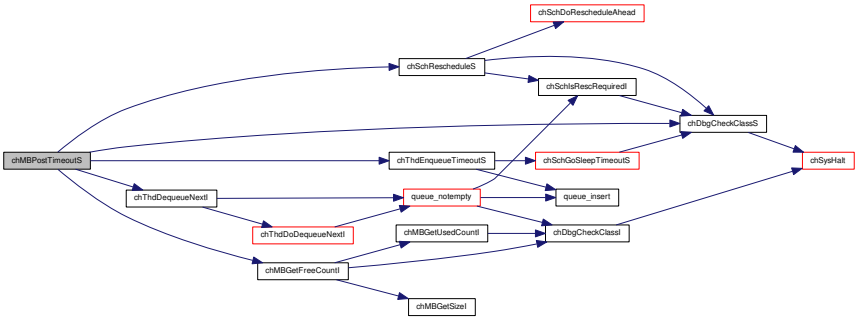
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.6 msg_t chMBPostI (mailbox_t * *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

in	*mbp*	the pointer to an initialized `mailbox_t` object
in	*msg*	the message to be posted on the mailbox

Returns

The operation status.

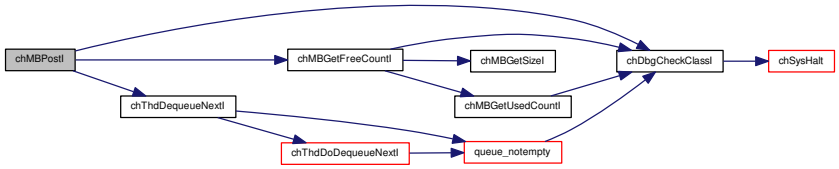
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.7 msg_t chMBPostAheadTimeout (mailbox_t * *mbp*, msg_t *msg*, sysinterval_t *timeout*)

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	*mbp*	the pointer to an initialized mailbox_t object
in	*msg*	the message to be posted on the mailbox
in	*timeout*	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.

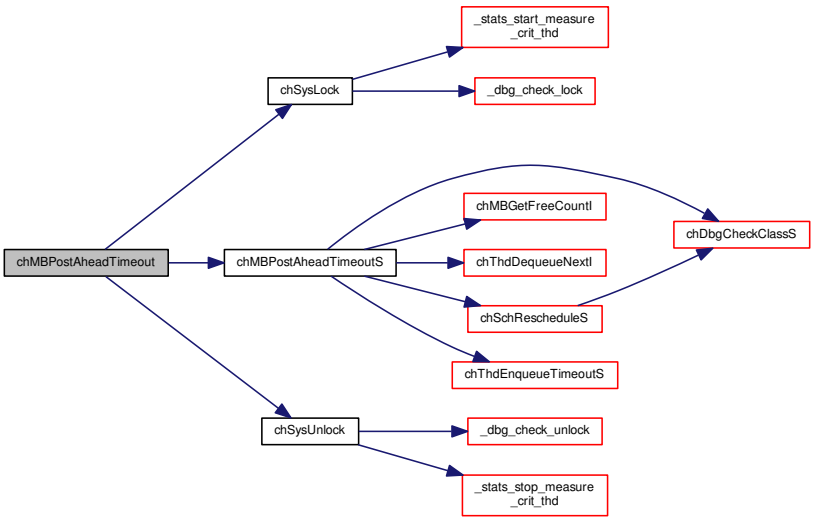
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.8 msg_t chMBPostAheadTimeoutS (mailbox_t * *mbp*, msg_t *msg*, sysinterval_t *timeout*)

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	*mbp*	the pointer to an initialized mailbox_t object
in	*msg*	the message to be posted on the mailbox
in	*timeout*	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.

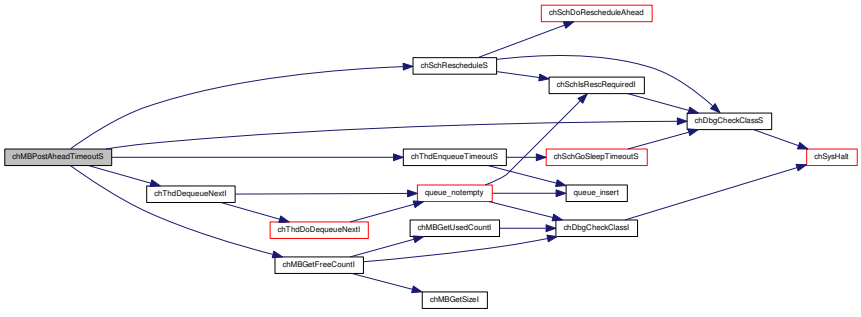
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.9 msg_t chMBPostAheadI (mailbox_t * *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	*mbp*	the pointer to an initialized mailbox_t object
in	*msg*	the message to be posted on the mailbox

Returns

The operation status.

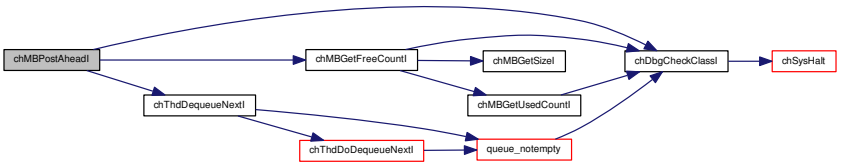
Return values

MSG_OK	if a message has been correctly posted.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.10 msg_t chMBFetchTimeout (mailbox_t * *mbp*, msg_t * *msgp*, sysinterval_t *timeout*)

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`	*mbp*	the pointer to an initialized `mailbox_t` object
`out`	*msgp*	pointer to a message variable for the received message
`in`	*timeout*	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.

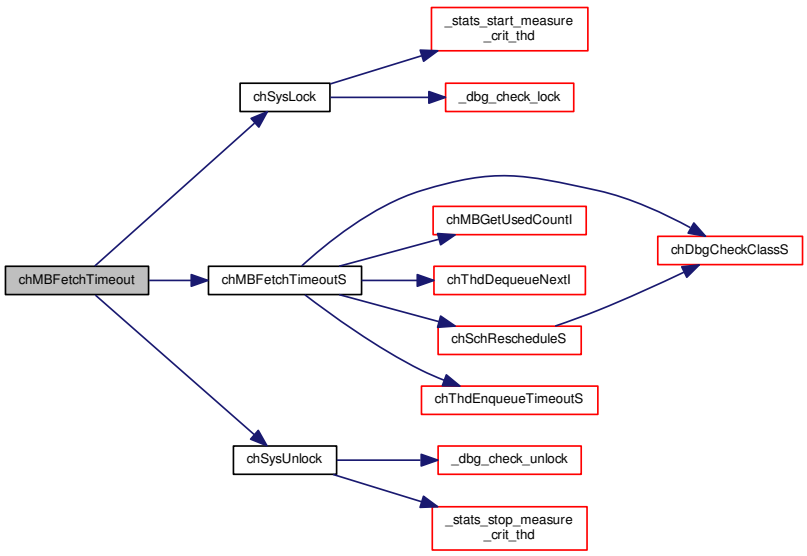
Return values

MSG_OK	if a message has been correctly fetched.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.11 msg_t chMBFetchTimeoutS (mailbox_t * *mbp*, msg_t * *msgp*, sysinterval_t *timeout*)

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	*mbp*	the pointer to an initialized mailbox_t object
out	*msgp*	pointer to a message variable for the received message
in	*timeout*	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.

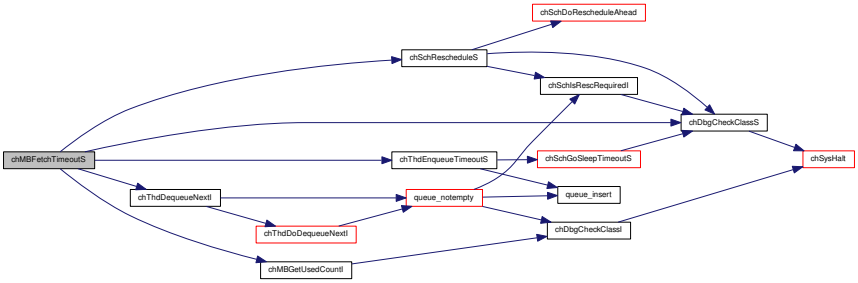
Return values

MSG_OK	if a message has been correctly fetched.
MSG_RESET	if the mailbox has been reset.
MSG_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:



8.17.3.12 msg_t chMBFetchI (mailbox_t * *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 empty.

Parameters

in	*mbp*	the pointer to an initialized mailbox_t object
out	*msgp*	pointer to a message variable for the received message

Returns

The operation status.

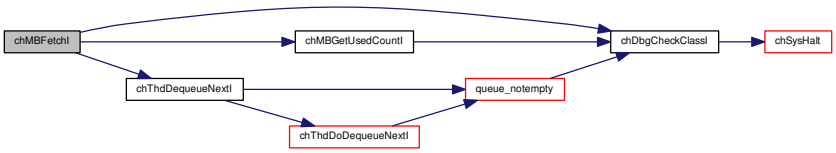
Return values

MSG_OK	if a message has been correctly fetched.
MSG_RESET	if the mailbox has been reset.
MSG_TIMEOUT	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:



8.17.3.13 static size_t chMBGetSizeI (const mailbox_t * *mbp*) [inline], [static]

Returns the mailbox buffer size as number of messages.

Parameters

in	*mbp*	the pointer to an initialized mailbox_t object

Returns

The size of the mailbox.

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.

8.17.3.14 static size_t chMBGetUsedCountI (const mailbox_t * *mbp*) [inline], [static]

Returns the number of used message slots into a mailbox.

Parameters

in	*mbp*	the pointer to an initialized mailbox_t object

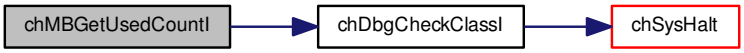
Returns

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.

Here is the call graph for this function:



8.17.3.15 static size_t chMBGetFreeCountI (const mailbox_t * *mbp*) [inline], [static]

Returns the number of free message slots into a mailbox.

Parameters

in	*mbp*	the pointer to an initialized mailbox_t object

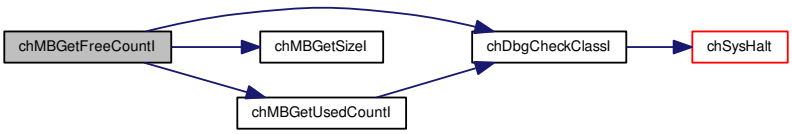
Returns

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.

Here is the call graph for this function:



8.17.3.16 static msg_t chMBPeekI (const mailbox_t * *mbp*) [inline], [static]

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 `chMBGetUsedCountI()` and then use this macro, all within a lock state.

Parameters

in	*mbp*	the pointer to an initialized `mailbox_t` object

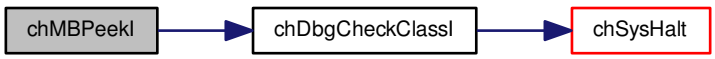
Returns

The next message in queue.

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:



8.17.3.17 static void chMBResumeX (mailbox_t * *mbp*) [inline], [static]

Terminates the reset state.

Parameters

in	*mbp*	the pointer to an initialized `mailbox_t` object

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.18 Memory Alignment

8.18.1 Detailed Description

Memory Alignment services.

Memory alignment support macros

- #define MEM_ALIGN_MASK(a) ((size_t)(a) - 1U)

  Alignment mask constant.
- #define MEM_ALIGN_PREV(p, a)

  Aligns to the previous aligned memory address.
- #define MEM_ALIGN_NEXT(p, a)

  Aligns to the next aligned memory address.
- #define MEM_IS_ALIGNED(p, a) (((size_t)(p) & MEM_ALIGN_MASK(a)) == 0U)

  Returns whatever a pointer or memory size is aligned.
- #define MEM_IS_VALID_ALIGNMENT(a) (((size_t)(a) != 0U) && (((size_t)(a) & ((size_t)(a) - 1U)) == 0U))

  Returns whatever a constant is a valid alignment.

8.18.2 Macro Definition Documentation

8.18.2.1 #define MEM_ALIGN_MASK(*a*) ((size_t)(a) - 1U)

Alignment mask constant.

Parameters

in	*a*	alignment, must be a power of two

8.18.2.2 #define MEM_ALIGN_PREV(*p*, *a*)

Value:

```
/*lint -save -e9033 [10.8] The cast is safe.*/                          \
  ((size_t)(p) & ~MEM_ALIGN_MASK(a))                                      \
  /*lint -restore*/
```

Aligns to the previous aligned memory address.

Parameters

in	*p*	variable to be aligned
in	*a*	alignment, must be a power of two

8.18.2.3 #define MEM_ALIGN_NEXT(*p*, *a*)

Value:

```
/*lint -save -e9033 [10.8] The cast is safe.*/                          \
  MEM_ALIGN_PREV((size_t)(p) + MEM_ALIGN_MASK(a), (a))                    \
  /*lint -restore*/
```

Aligns to the next aligned memory address.

Parameters

in	*p*	variable to be aligned
in	*a*	alignment, must be a power of two

8.18.2.4 #define MEM_IS_ALIGNED(*p*, *a*) (((size_t)(p) & MEM_ALIGN_MASK(a)) == 0U)

Returns whatever a pointer or memory size is aligned.

Parameters

in	*p*	variable to be aligned
in	*a*	alignment, must be a power of two

8.18.2.5 #define MEM_IS_VALID_ALIGNMENT(*a*) (((size_t)(a) != 0U) && (((size_t)(a) & ((size_t)(a) - 1U)) == 0U))

Returns whatever a constant is a valid alignment.

Valid alignments are powers of two.

Parameters

in	*a*	alignment to be checked, must be a constant

8.19 Memory Management

8.19.1 Detailed Description

Memory Management services.

Modules

- Core Memory Manager
- Heaps
- Memory Pools
- Dynamic Threads

8.20 Core Memory Manager

8.20.1 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_CFG_USE_MEMCORE` option must be enabled in `chconf.h`.

Note

Compatible with RT and NIL.

Macros

- #define CH_CFG_MEMCORE_SIZE 0
  Managed RAM size.

Typedefs

- typedef void *(* memgetfunc_t) (size_t size, unsigned align)
  Memory get function.
- typedef void *(* memgetfunc2_t) (size_t size, unsigned align, size_t offset)
  Enhanced memory get function.

Data Structures

- struct memcore_t
  Type of memory core object.

Functions

- void _core_init (void)
  Low level memory manager initialization.
- void * chCoreAllocAlignedWithOffsetI (size_t size, unsigned align, size_t offset)
  Allocates a memory block.

- void * chCoreAllocAlignedWithOffset (size_t size, unsigned align, size_t offset)

  Allocates a memory block.
- size_t chCoreGetStatusX (void)

  Core memory status.
- static void * chCoreAllocAlignedI (size_t size, unsigned align)

  Allocates a memory block.
- static void * chCoreAllocAligned (size_t size, unsigned align)

  Allocates a memory block.
- static void * chCoreAllocI (size_t size)

  Allocates a memory block.
- static void * chCoreAlloc (size_t size)

  Allocates a memory block.

Variables

- memcore_t ch_memcore

  Memory core descriptor.

8.20.2 Macro Definition Documentation

8.20.2.1 #define CH_CFG_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_CFG_USE_MEMCORE.

8.20.3 Typedef Documentation

8.20.3.1 typedef void*(* memgetfunc_t) (size_t size, unsigned align)

Memory get function.

8.20.3.2 typedef void*(* memgetfunc2_t) (size_t size, unsigned align, size_t offset)

Enhanced memory get function.

8.20.4 Function Documentation

8.20.4.1 void _core_init (void)

Low level memory manager initialization.

Function Class:

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

8.20.4.2 void * chCoreAllocAlignedWithOffsetI (size_t *size*, unsigned *align*, size_t *offset*)

Allocates a memory block.

This function allocates a block of `offset + size` bytes. The returned pointer has `offset` bytes before its address and `size` bytes after.

Parameters

`in`	*size*	the size of the block to be allocated.
`in`	*align*	desired memory alignment
`in`	*offset*	aligned pointer offset

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:

8.20.4.3 void * chCoreAllocAlignedWithOffset (size_t *size*, unsigned *align*, size_t *offset*)

Allocates a memory block.

This function allocates a block of `offset + size` bytes. The returned pointer has `offset` bytes before its address and `size` bytes after.

Parameters

`in`	*size*	the size of the block to be allocated.
`in`	*align*	desired memory alignment
`in`	*offset*	aligned pointer offset

Returns

A pointer to the allocated memory block.

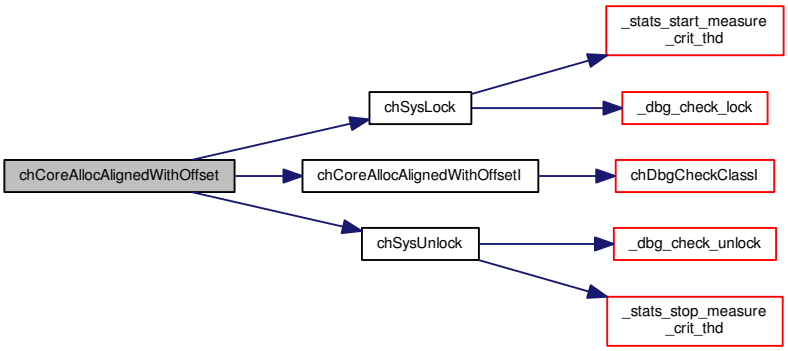
Return values

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:



8.20.4.4 size_t chCoreGetStatusX (void)

Core memory status.

Returns

The size, in bytes, of the free core memory.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.20.4.5 static void* chCoreAllocAlignedI (size_t *size*, unsigned *align*) [inline], [static]

Allocates a memory block.

The allocated block is guaranteed to be properly aligned to the specified alignment.

Parameters

in	*size*	the size of the block to be allocated.
in	*align*	desired memory alignment

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:

8.20.4.6 static void* chCoreAllocAligned (size_t *size*, unsigned *align*) [inline], [static]

Allocates a memory block.

The allocated block is guaranteed to be properly aligned to the specified alignment.

Parameters

in	*size*	the size of the block to be allocated
in	*align*	desired memory alignment

Returns

A pointer to the allocated memory block.

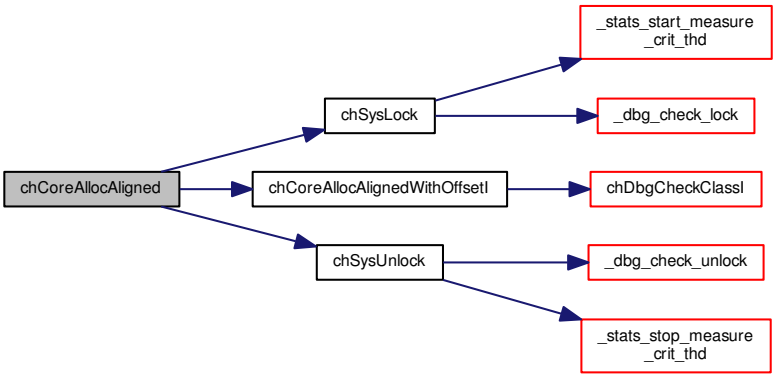
Return values

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:



8.20.4.7 static void* chCoreAllocI (size_t *size*) [inline], [static]

Allocates a memory block.

The allocated block is guaranteed to be properly aligned for a pointer data type.

Parameters

in	*size*	the size of the block to be allocated.

Returns

A pointer to the allocated memory block.

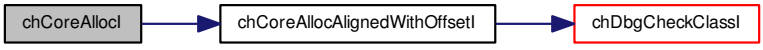
Return values

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:



8.20.4.8 static void* chCoreAlloc (size_t *size*) [inline], [static]

Allocates a memory block.

The allocated block is guaranteed to be properly aligned for a pointer data type.

Parameters

in	*size*	the size of the block to be allocated.

Returns

A pointer to the allocated memory block.

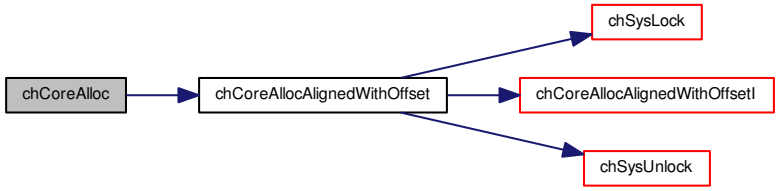
Return values

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:



8.20.5 Variable Documentation

8.20.5.1 memcore_t ch_memcore

Memory core descriptor.

8.21 Heaps

8.21.1 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 and there is the ability to return memory blocks aligned to arbitrary powers of two.

Precondition

In order to use the heap APIs the `CH_CFG_USE_HEAP` option must be enabled in `chconf.h`.

Note

Compatible with RT and NIL.

Macros

- #define CH_HEAP_ALIGNMENT 8U

  Minimum alignment used for heap.
- #define CH_HEAP_AREA(name, size)

  Allocation of an aligned static heap buffer.

Typedefs

- typedef struct memory_heap memory_heap_t

  Type of a memory heap.
- typedef union heap_header heap_header_t

  Type of a memory heap header.

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 chHeapObjectInit (memory_heap_t *heapp, void *buf, size_t size)

  Initializes a memory heap from a static memory area.
- void * chHeapAllocAligned (memory_heap_t *heapp, size_t size, unsigned align)

  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 (memory_heap_t *heapp, size_t *totalp, size_t *largestp)

  Reports the heap status.

- static void * chHeapAlloc (memory_heap_t *heapp, size_t size)

  Allocates a block of memory from the heap by using the first-fit algorithm.

- static size_t chHeapGetSize (const void *p)

  Returns the size of an allocated block.

Variables

- static memory_heap_t default_heap

  Default heap descriptor.

8.21.2 Macro Definition Documentation

8.21.2.1 #define CH_HEAP_ALIGNMENT 8U

Minimum alignment used for heap.

Note

Cannot use the sizeof operator in this macro.

8.21.2.2 #define CH_HEAP_AREA(*name, size*)

Value:

```
ALIGNED_VAR(CH_HEAP_ALIGNMENT)                                          \
  uint8_t name[MEM_ALIGN_NEXT((size), CH_HEAP_ALIGNMENT)]
```

Allocation of an aligned static heap buffer.

8.21.3 Typedef Documentation

8.21.3.1 typedef struct memory_heap memory_heap_t

Type of a memory heap.

8.21.3.2 typedef union heap_header heap_header_t

Type of a memory heap header.

8.21.4 Function Documentation

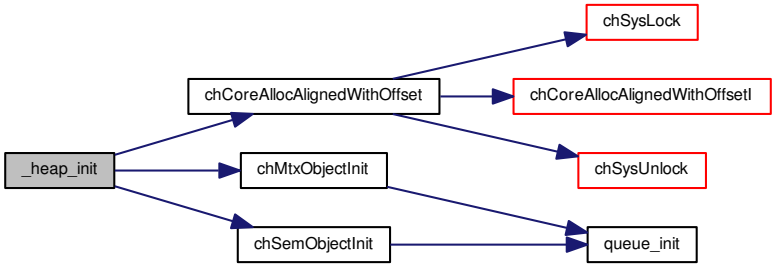
8.21.4.1 void _heap_init (void)

Initializes the default heap.

Function Class:

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

Here is the call graph for this function:



8.21.4.2 void chHeapObjectInit (memory_heap_t * *heapp*, void * *buf*, size_t *size*)

Initializes a memory heap from a static memory area.

Note

The heap buffer base and size are adjusted if the passed buffer is not aligned to `CH_HEAP_ALIGNMENT`. This mean that the effective heap size can be less than `size`.

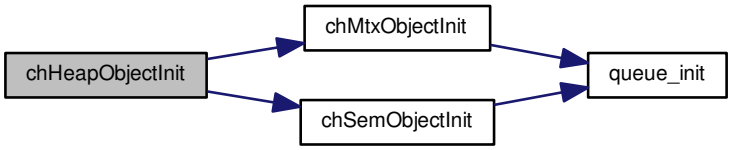
Parameters

`out`	*heapp*	pointer to the memory heap descriptor to be initialized
`in`	*buf*	heap buffer base
`in`	*size*	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:



8.21.4.3 void * chHeapAllocAligned (memory_heap_t * *heapp*, size_t *size*, unsigned *align*)

Allocates a block of memory from the heap by using the first-fit algorithm.

The allocated block is guaranteed to be properly aligned to the specified alignment.

Parameters

in	*heapp*	pointer to a heap descriptor or NULL in order to access the default heap.
in	*size*	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.
in	*align*	desired memory alignment

Returns

A pointer to the aligned 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.

8.21.4.4 void chHeapFree (void * *p*)

Frees a previously allocated memory block.

Parameters

in	*p*	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.

8.21.4.5 size_t chHeapStatus (memory_heap_t * *heapp*, size_t * *totalp*, size_t * *largestp*)

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.

Parameters

in	*heapp*	pointer to a heap descriptor or NULL in order to access the default heap.
in	*totalp*	pointer to a variable that will receive the total fragmented free space or NULL
in	*largestp*	pointer to a variable that will receive the largest free free block found space or NULL

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.

8.21.4.6 static void* chHeapAlloc (memory_heap_t * *heapp*, size_t *size*) [inline], [static]

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.

Parameters

in	*heapp*	pointer to a heap descriptor or NULL in order to access the default heap.
in	*size*	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.

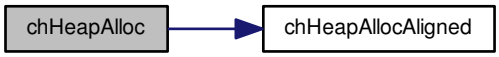
Return values

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.

Here is the call graph for this function:



8.21.4.7 static size_t chHeapGetSize (const void * *p*) [inline], [static]

Returns the size of an allocated block.

Note

The returned value is the requested size, the real size is the same value aligned to the next CH_HEAP_AL↩IGNMENT multiple.

Parameters

in	*p*	pointer to the memory block

Returns

Size of the block.

Function Class:

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

8.21.5 Variable Documentation

8.21.5.1 memory_heap_t default_heap [static]

Default heap descriptor.

8.22 Memory Pools

8.22.1 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.
Memory Pools do not enforce any alignment constraint on the contained object however the objects must be properly aligned to contain a pointer to void.

Precondition

In order to use the memory pools APIs the `CH_CFG_USE_MEMPOOLS` option must be enabled in `chconf.h`.

Note

Compatible with RT and NIL.

Macros

- #define _MEMORYPOOL_DATA(name, size, align, provider) {NULL, size, align, provider}

  Data part of a static memory pool initializer.
- #define MEMORYPOOL_DECL(name, size, align, provider) memory_pool_t name = _MEMORYPOOL_DATA(name, size, align, provider)

  Static memory pool initializer.
- #define _GUARDEDMEMORYPOOL_DATA(name, size, align)

  Data part of a static guarded memory pool initializer.
- #define GUARDEDMEMORYPOOL_DECL(name, size, align) guarded_memory_pool_t name = _GUARDEDMEMORYPOOL_DATA(name, size, align)

  Static guarded memory pool initializer.

Data Structures

- struct pool_header

  Memory pool free object header.
- struct memory_pool_t

  Memory pool descriptor.
- struct guarded_memory_pool_t

  Guarded memory pool descriptor.

Functions

- void chPoolObjectInitAligned (memory_pool_t *mp, size_t size, unsigned align, memgetfunc_t provider)

  Initializes an empty memory pool.
- void chPoolLoadArray (memory_pool_t *mp, void *p, size_t n)

  Loads a memory pool with an array of static objects.
- void * chPoolAllocI (memory_pool_t *mp)

  Allocates an object from a memory pool.

- void * chPoolAlloc (memory_pool_t *mp)

  Allocates an object from a memory pool.
- void chPoolFreeI (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chPoolFree (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chGuardedPoolObjectInitAligned (guarded_memory_pool_t *gmp, size_t size, unsigned align)

  Initializes an empty guarded memory pool.
- void chGuardedPoolLoadArray (guarded_memory_pool_t *gmp, void *p, size_t n)

  Loads a guarded memory pool with an array of static objects.
- void * chGuardedPoolAllocTimeoutS (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void * chGuardedPoolAllocTimeout (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void chGuardedPoolFreeI (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.
- void chGuardedPoolFree (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.
- static void chPoolObjectInit (memory_pool_t *mp, size_t size, memgetfunc_t provider)

  Initializes an empty memory pool.
- static void chPoolAdd (memory_pool_t *mp, void *objp)

  Adds an object to a memory pool.
- static void chPoolAddI (memory_pool_t *mp, void *objp)

  Adds an object to a memory pool.
- static void chGuardedPoolObjectInit (guarded_memory_pool_t *gmp, size_t size)

  Initializes an empty guarded memory pool.
- static void chGuardedPoolAdd (guarded_memory_pool_t *gmp, void *objp)

  Adds an object to a guarded memory pool.
- static void chGuardedPoolAddI (guarded_memory_pool_t *gmp, void *objp)

  Adds an object to a guarded memory pool.
- static void * chGuardedPoolAllocI (guarded_memory_pool_t *gmp)

  Allocates an object from a guarded memory pool.

8.22.2 Macro Definition Documentation

8.22.2.1 #define _MEMORYPOOL_DATA(*name, size, align, provider*) {NULL, size, align, 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	*name*	the name of the memory pool variable
in	*size*	size of the memory pool contained objects
in	*align*	required memory alignment
in	*provider*	memory provider function for the memory pool

8.22.2.2 #define MEMORYPOOL_DECL(*name, size, align, provider*) memory_pool_t name = _MEMORYPOOL_DATA(name, size, align, provider)

Static memory pool initializer.

Statically initialized memory pools require no explicit initialization using `chPoolInit()`.

Parameters

in	*name*	the name of the memory pool variable
in	*size*	size of the memory pool contained objects
in	*align*	required memory alignment
in	*provider*	memory provider function for the memory pool or `NULL` if the pool is not allowed to grow automatically

8.22.2.3 #define _GUARDEDMEMORYPOOL_DATA(*name, size, align*)

Value:

```
{                           \
  _SEMAPHORE_DATA(name.sem, (cnt_t)0),
      \
  _MEMORYPOOL_DATA(NULL, size, align, NULL)                                 \
}
```

Data part of a static guarded memory pool initializer.

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

Parameters

in	*name*	the name of the memory pool variable
in	*size*	size of the memory pool contained objects
in	*align*	required memory alignment

8.22.2.4 #define GUARDEDMEMORYPOOL_DECL(*name, size, align*) guarded_memory_pool_t name = _GUARDEDMEMORYPOOL_DATA(name, size, align)

Static guarded memory pool initializer.

Statically initialized guarded memory pools require no explicit initialization using `chGuardedPoolInit()`.

Parameters

in	*name*	the name of the guarded memory pool variable
in	*size*	size of the memory pool contained objects
in	*align*	required memory alignment

8.22.3 Function Documentation

8.22.3.1 void chPoolObjectInitAligned (memory_pool_t * *mp,* size_t *size,* unsigned *align,* memgetfunc_t *provider*)

Initializes an empty memory pool.

Parameters

out	*mp*	pointer to a memory_pool_t structure
in	*size*	the size of the objects contained in this memory pool, the minimum accepted size is the size of a pointer to void.
in	*align*	required memory alignment
in	*provider*	memory provider function for the memory pool or NULL 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.

8.22.3.2 void chPoolLoadArray (memory_pool_t * *mp*, void * *p*, size_t *n*)

Loads a memory pool with an array of static objects.

Precondition

The memory pool must already be initialized.
The array elements must be of the right size for the specified memory pool.
The array elements size must be a multiple of the alignment requirement for the pool.

Postcondition

The memory pool contains the elements of the input array.

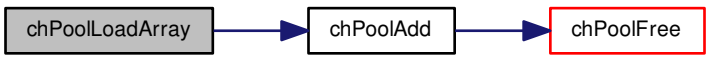
Parameters

in	*mp*	pointer to a memory_pool_t structure
in	*p*	pointer to the array first element
in	*n*	number of elements in the array

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:



8.22.3.3 void * chPoolAlloc (memory_pool_t * *mp*)

Allocates an object from a memory pool.

Precondition

The memory pool must already be initialized.

Parameters

in	*mp*	pointer to a memory_pool_t structure

Returns

The pointer to the allocated object.

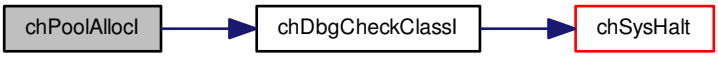
Return values

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:



8.22.3.4 void * chPoolAlloc (memory_pool_t * *mp*)

Allocates an object from a memory pool.

Precondition

The memory pool must already be initialized.

Parameters

in	*mp*	pointer to a memory_pool_t structure

Returns

The pointer to the allocated object.

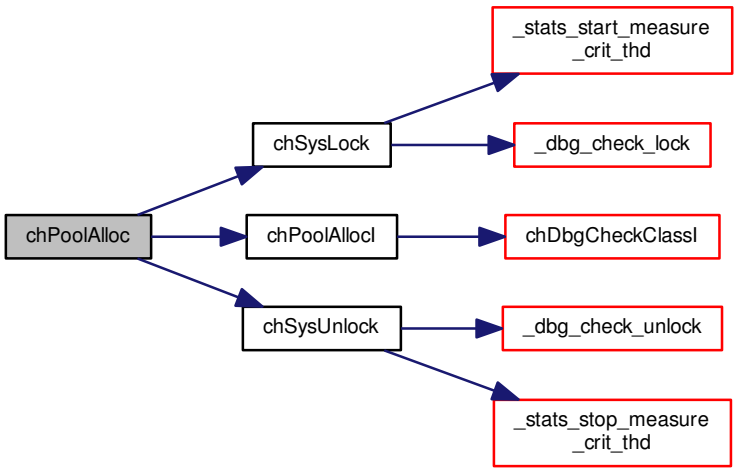
Return values

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:



8.22.3.5 void chPoolFreeI (memory_pool_t * *mp*, void * *objp*)

Releases an object into a memory pool.

Precondition

The memory pool must already be initialized.
The freed object must be of the right size for the specified memory pool.
The added object must be properly aligned.

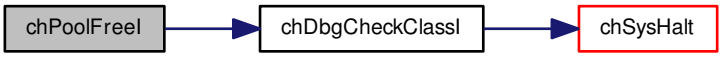
Parameters

in	*mp*	pointer to a memory_pool_t structure
in	*objp*	the pointer to the object to be released

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:



8.22.3.6 void chPoolFree (memory_pool_t * *mp*, void * *objp*)

Releases an object into a memory pool.

Precondition

The memory pool must already be initialized.
The freed object must be of the right size for the specified memory pool.
The added object must be properly aligned.

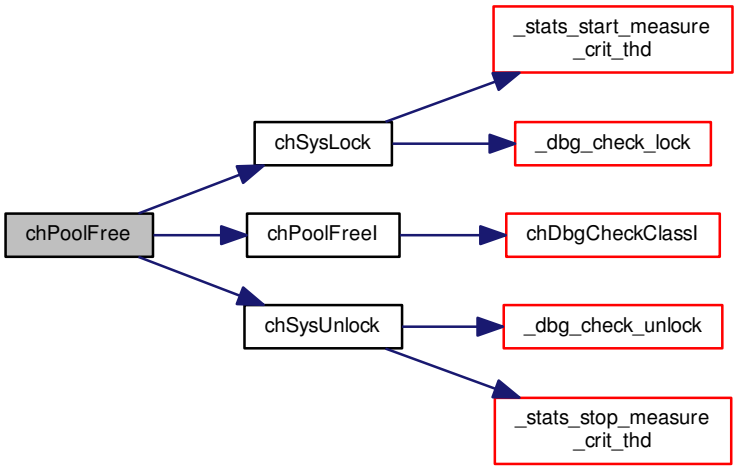
Parameters

in	*mp*	pointer to a memory_pool_t structure
in	*objp*	the pointer to the object to be released

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:



8.22.3.7 void chGuardedPoolObjectInitAligned (guarded_memory_pool_t * *gmp*, size_t *size*, unsigned *align*)

Initializes an empty guarded memory pool.

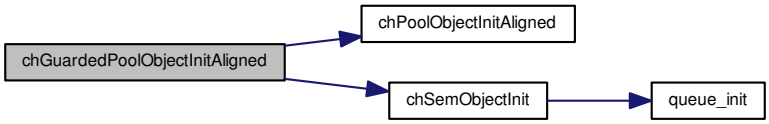
Parameters

out	*gmp*	pointer to a guarded_memory_pool_t structure
in	*size*	the size of the objects contained in this guarded memory pool, the minimum accepted size is the size of a pointer to void.
in	*align*	required memory alignment

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:



8.22.3.8 void chGuardedPoolLoadArray (guarded_memory_pool_t * *gmp*, void * *p*, size_t *n*)

Loads a guarded memory pool with an array of static objects.

Precondition

The guarded memory pool must already be initialized.
The array elements must be of the right size for the specified guarded memory pool.

Postcondition

The guarded memory pool contains the elements of the input array.

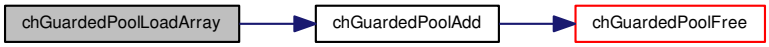
Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure
in	*p*	pointer to the array first element
in	*n*	number of elements in the array

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:



8.22.3.9 void * chGuardedPoolAllocTimeoutS (guarded_memory_pool_t * *gmp*, sysinterval_t *timeout*)

Allocates an object from a guarded memory pool.

Precondition

The guarded memory pool must already be initialized.

Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure
in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The pointer to the allocated object.

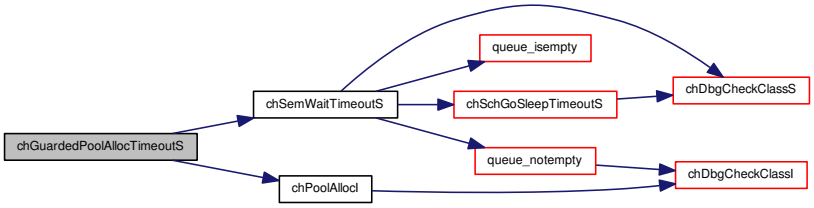
Return values

NULL	if the operation 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:



8.22.3.10 void * chGuardedPoolAllocTimeout (guarded_memory_pool_t * *gmp*, sysinterval_t *timeout*)

Allocates an object from a guarded memory pool.

Precondition

The guarded memory pool must already be initialized.

Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure
in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The pointer to the allocated object.

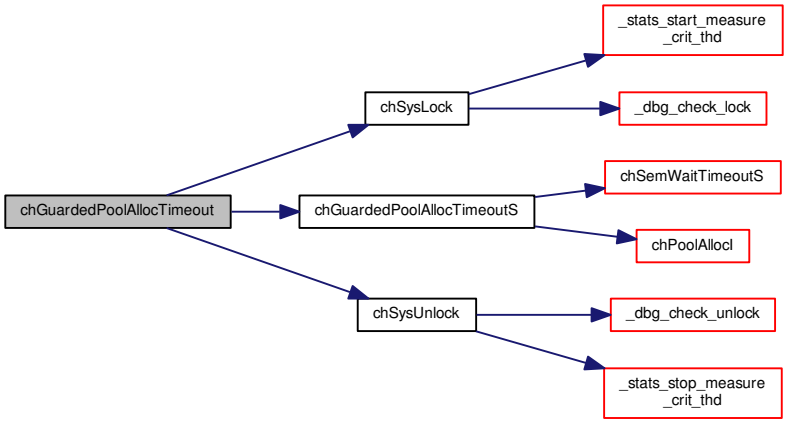
Return values

NULL	if the operation 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:



8.22.3.11 void chGuardedPoolFreeI (guarded_memory_pool_t * *gmp*, void * *objp*)

Releases an object into a guarded memory pool.

Precondition

The guarded memory pool must already be initialized.
The freed object must be of the right size for the specified guarded memory pool.
The added object must be properly aligned.

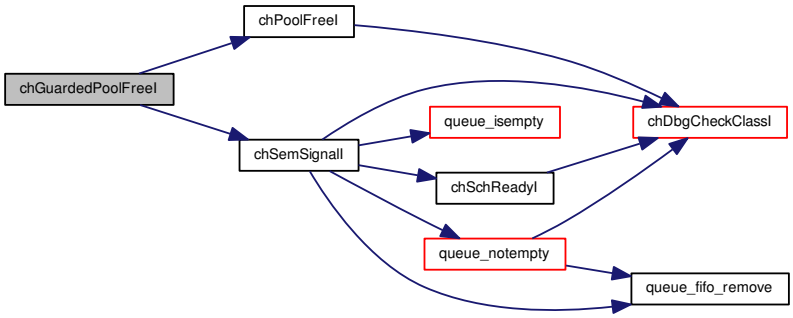
Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure
in	*objp*	the pointer to the object to be released

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:



8.22.3.12 void chGuardedPoolFree (guarded_memory_pool_t * *gmp*, void * *objp*)

Releases an object into a guarded memory pool.

Precondition

The guarded memory pool must already be initialized.
The freed object must be of the right size for the specified guarded memory pool.
The added object must be properly aligned.

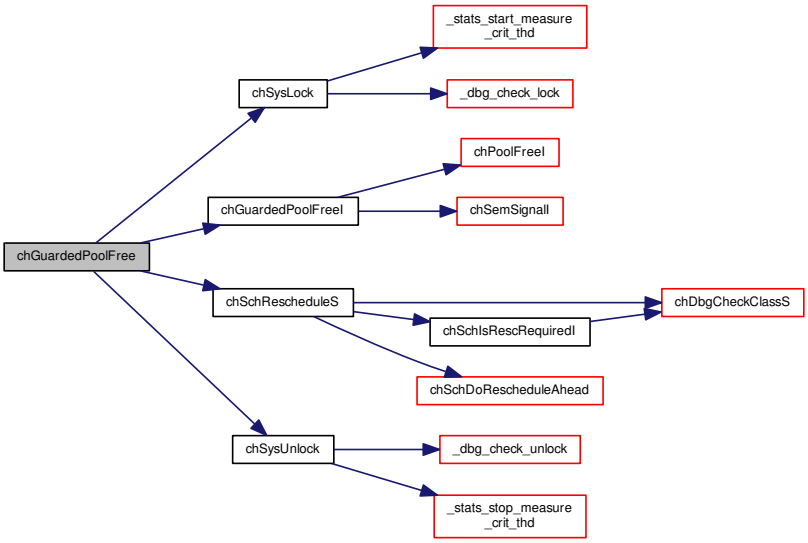
Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure
in	*objp*	the pointer to the object to be released

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:



8.22.3.13 static void chPoolObjectInit (memory_pool_t * *mp*, size_t *size*, memgetfunc_t *provider*) [inline], [static]

Initializes an empty memory pool.

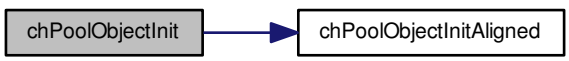
Parameters

out	*mp*	pointer to a memory_pool_t structure
in	*size*	the size of the objects contained in this memory pool, the minimum accepted size is the size of a pointer to void.
in	*provider*	memory provider function for the memory pool or NULL 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.

Here is the call graph for this function:



8.22.3.14 static void chPoolAdd (memory_pool_t * *mp*, void * *objp*) [inline], [static]

Adds an object to a memory pool.

Precondition

The memory pool must be already been initialized.
The added object must be of the right size for the specified memory pool.
The added object must be memory aligned to the size of `stkalign_t` type.

Note

This function is just an alias for `chPoolFree()` and has been added for clarity.

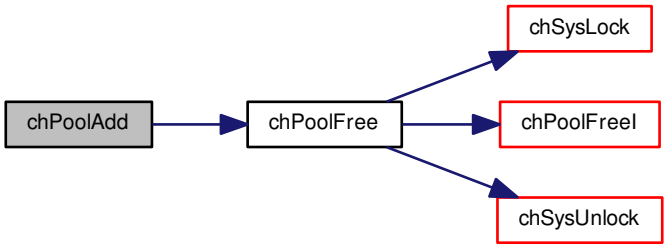
Parameters

`in`	*mp*	pointer to a `memory_pool_t` structure
`in`	*objp*	the pointer to the object to be 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:



8.22.3.15 static void chPoolAddI (memory_pool_t * *mp*, void * *objp*) [inline], [static]

Adds an object to a memory pool.

Precondition

The memory pool must be already been initialized.
The added object must be of the right size for the specified memory pool.
The added object must be memory aligned to the size of `stkalign_t` type.

Note

This function is just an alias for `chPoolFreeI()` and has been added for clarity.

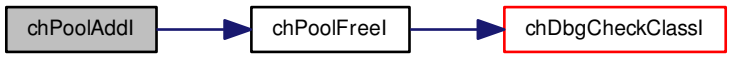
Parameters

in	*mp*	pointer to a memory_pool_t structure
in	*objp*	the pointer to the object to be 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:



8.22.3.16 static void chGuardedPoolObjectInit (guarded_memory_pool_t * *gmp*, size_t *size*) [inline], [static]

Initializes an empty guarded memory pool.

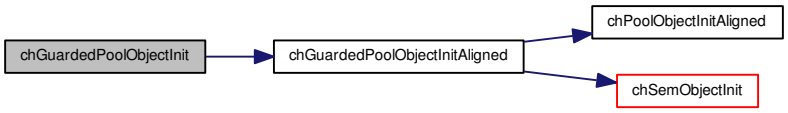
Parameters

out	*gmp*	pointer to a guarded_memory_pool_t structure
in	*size*	the size of the objects contained in this guarded memory pool, the minimum accepted size is the size of a pointer to void.

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:



8.22.3.17 static void chGuardedPoolAdd (guarded_memory_pool_t * *gmp*, void * *objp*) [inline], [static]

Adds an object to a guarded memory pool.

Precondition

The guarded memory pool must be already been initialized.
The added object must be of the right size for the specified guarded memory pool.
The added object must be properly aligned.

Note

This function is just an alias for `chGuardedPoolFree()` and has been added for clarity.

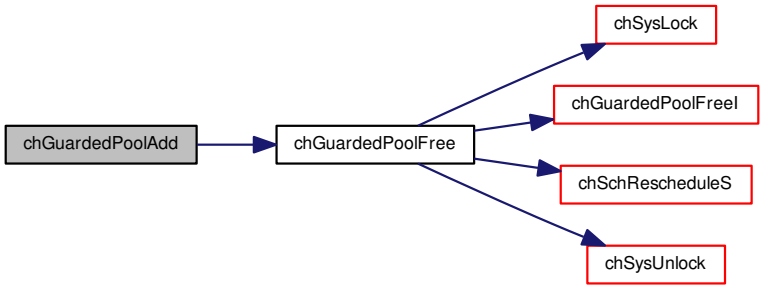
Parameters

in	*gmp*	pointer to a `guarded_memory_pool_t` structure
in	*objp*	the pointer to the object to be 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:



8.22.3.18 static void chGuardedPoolAddI (guarded_memory_pool_t * *gmp*, void * *objp*) [inline], [static]

Adds an object to a guarded memory pool.

Precondition

The guarded memory pool must be already been initialized.
The added object must be of the right size for the specified guarded memory pool.
The added object must be properly aligned.

Note

This function is just an alias for `chGuardedPoolFreeI()` and has been added for clarity.

Parameters

in	*gmp*	pointer to a `guarded_memory_pool_t` structure
in	*objp*	the pointer to the object to be 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:

8.22.3.19 static void* chGuardedPoolAllocI (guarded_memory_pool_t * *gmp*) [inline], [static]

Allocates an object from a guarded memory pool.

Precondition

The guarded memory pool must be already been initialized.

Parameters

in	*gmp*	pointer to a guarded_memory_pool_t structure

Returns

The pointer to the allocated object.

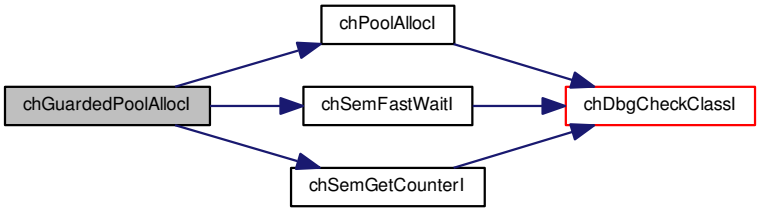
Return values

NULL	if the 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:



8.23 Dynamic Threads

8.23.1 Detailed Description

Dynamic threads related APIs and services.

Functions

- thread_t * chThdCreateFromHeap (memory_heap_t *heapp, size_t size, const char *name, tprio_t prio, tfunc_t pf, void *arg)

  Creates a new thread allocating the memory from the heap.
- thread_t * chThdCreateFromMemoryPool (memory_pool_t *mp, const char *name, tprio_t prio, tfunc_t pf, void *arg)

  Creates a new thread allocating the memory from the specified memory pool.

8.23.2 Function Documentation

8.23.2.1 thread_t * chThdCreateFromHeap (memory_heap_t * *heapp*, size_t *size*, const char * *name*, tprio_t *prio*, tfunc_t *pf*, void * *arg*)

Creates a new thread allocating the memory from the heap.

Precondition

The configuration options `CH_CFG_USE_DYNAMIC` and `CH_CFG_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 automatically, it is responsibility of the creator thread to call `chThdWait()` and then release the allocated memory.

Parameters

`in`	*heapp*	heap from which allocate the memory or `NULL` for the default heap
`in`	*size*	size of the working area to be allocated
`in`	*name*	thread name
`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_t` structure allocated for the thread into the working space area.

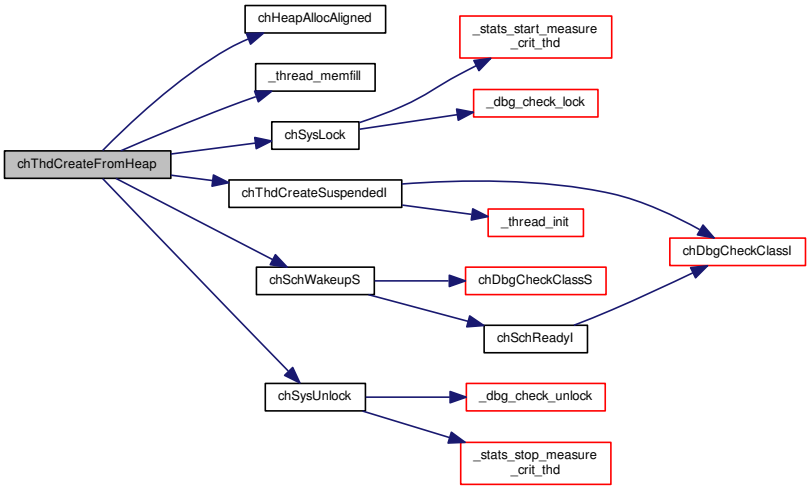
Return values

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:



8.23.2.2 thread_t * chThdCreateFromMemoryPool (memory_pool_t * *mp*, const char * *name*, 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_CFG_USE_DYNAMIC` and `CH_CFG_USE_MEMPOOLS` must be enabled in order to use this function.
The pool must be initialized to contain only objects with alignment `PORT_WORKING_AREA_ALIGN`.

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 automatically, it is responsibility of the creator thread to call `chThdWait()` and then release the allocated memory.

Parameters

in	*mp*	pointer to the memory pool object
in	*name*	thread name
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_t` structure allocated for the thread into the working space area.

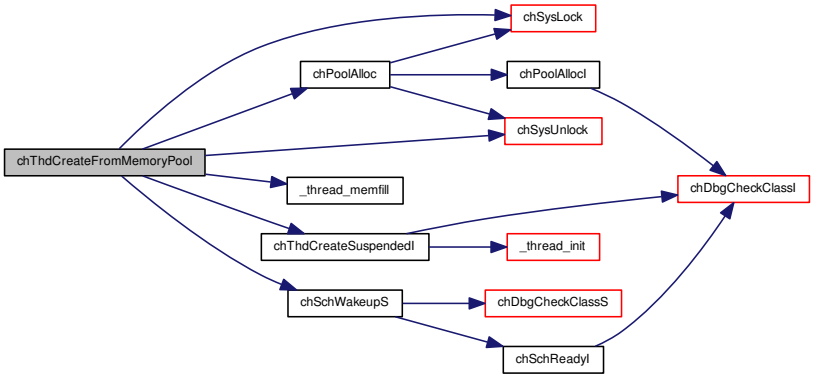
Return values

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:



8.24 Registry

8.24.1 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_CFG_USE_REGISTRY` option must be enabled in `chconf.h`.

Macros

- #define REG_REMOVE(tp)

  Removes a thread from the registry list.
- #define REG_INSERT(tp)

  Adds a thread to the registry list.

Data Structures

- struct chdebug_t

  ChibiOS/RT memory signature record.

Functions

- thread_t * chRegFirstThread (void)

  Returns the first thread in the system.
- thread_t * chRegNextThread (thread_t *tp)

  Returns the thread next to the specified one.
- thread_t * chRegFindThreadByName (const char *name)

  Retrieves a thread pointer by name.
- thread_t * chRegFindThreadByPointer (thread_t *tp)

  Confirms that a pointer is a valid thread pointer.
- thread_t * chRegFindThreadByWorkingArea (stkalign_t *wa)

  Confirms that a working area is being used by some active thread.
- static void chRegSetThreadName (const char *name)

  Sets the current thread name.
- static const char * chRegGetThreadNameX (thread_t *tp)

  Returns the name of the specified thread.
- static void chRegSetThreadNameX (thread_t *tp, const char *name)

  Changes the name of the specified thread.

8.24.2 Macro Definition Documentation

8.24.2.1 #define REG_REMOVE(*tp*)

Value:

```
{                                                   \
  (tp)->older->newer = (tp)->newer;                 \
  (tp)->newer->older = (tp)->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

8.24.2.2 #define REG_INSERT(*tp*)

Value:

```
{                                                   \
  (tp)->newer = (thread_t *)&ch.rlist;              \
  (tp)->older = ch.rlist.older;                     \
  (tp)->older->newer = (tp);                        \
  ch.rlist.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

8.24.3 Function Documentation

8.24.3.1 thread_t * 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.

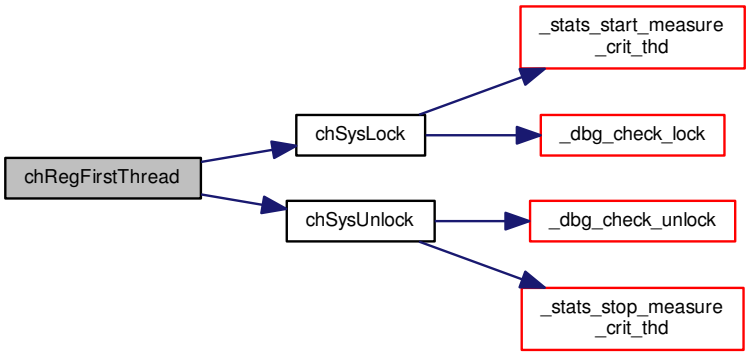
Returns

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.

Here is the call graph for this function:



8.24.3.2 thread_t * chRegNextThread (thread_t * *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.

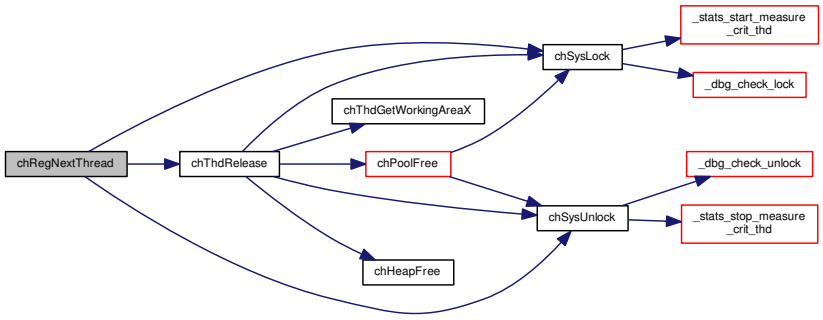
Return values

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:



8.24.3.3 thread_t * chRegFindThreadByName (const char * *name*)

Retrieves a thread pointer by name.

Note

The reference counter of the found thread is increased by one so it cannot be disposed incidentally after the pointer has been returned.

Parameters

in	*name*	the thread name

Returns

A pointer to the found thread.

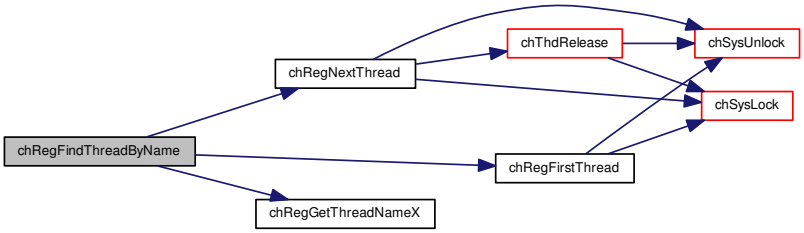
Return values

NULL	if a matching thread has not been found.

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:



8.24.3.4 thread_t * chRegFindThreadByPointer (thread_t * *tp*)

Confirms that a pointer is a valid thread pointer.

Note

The reference counter of the found thread is increased by one so it cannot be disposed incidentally after the pointer has been returned.

Parameters

in	*tp*	pointer to the thread

Returns

A pointer to the found thread.

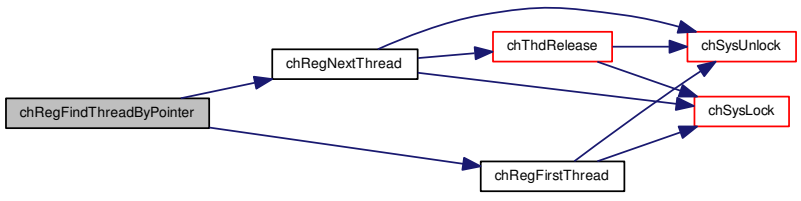
Return values

NULL	if a matching thread has not been found.

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:



8.24.3.5 thread_t * chRegFindThreadByWorkingArea (stkalign_t * *wa*)

Confirms that a working area is being used by some active thread.

Note

The reference counter of the found thread is increased by one so it cannot be disposed incidentally after the pointer has been returned.

Parameters

in	*wa*	pointer to a static working area

Returns

A pointer to the found thread.

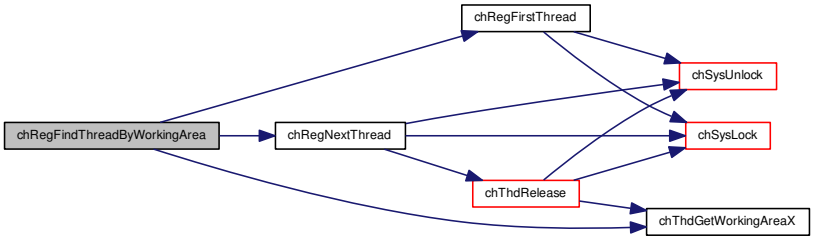
Return values

NULL	if a matching thread has not been found.

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:



8.24.3.6 static void chRegSetThreadName (const char * *name*) [inline], [static]

Sets the current thread name.

Precondition

This function only stores the pointer to the name if the option CH_CFG_USE_REGISTRY is enabled else no action is performed.

Parameters

in	*name*	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.

8.24.3.7 static const char* chRegGetThreadNameX (thread_t * tp) [inline], [static]

Returns the name of the specified thread.

Precondition

This function only returns the pointer to the name if the option `CH_CFG_USE_REGISTRY` is enabled else `NULL` is returned.

Parameters

`in`	*tp*	pointer to the thread

Returns

Thread name as a zero terminated string.

Return values

NULL	if the thread name has not been set.

8.24.3.8 static void chRegSetThreadNameX (thread_t * tp, const char * name) [inline], [static]

Changes the name of the specified thread.

Precondition

This function only stores the pointer to the name if the option `CH_CFG_USE_REGISTRY` is enabled else no action is performed.

Parameters

`in`	*tp*	pointer to the thread
`in`	*name*	thread name as a zero terminated string

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.25 Debug

8.25.1 Detailed Description

Debug APIs and services:

- Runtime system state and call protocol check. The following panic messages can be generated:
  - SV#1, misplaced `chSysDisable()`.
    - Called from an ISR.
    - Called from a critical zone.
  - SV#2, misplaced `chSysSuspend()`
    - Called from an ISR.
    - Called from a critical zone.
  - SV#3, misplaced `chSysEnable()`.
    - Called from an ISR.
    - Called from a critical zone.
  - SV#4, misplaced `chSysLock()`.
    - Called from an ISR.
    - Called from a critical zone.
  - SV#5, misplaced `chSysUnlock()`.
    - Called from an ISR.
    - Not called from a critical zone.
  - SV#6, misplaced `chSysLockFromISR()`.
    - Not called from an ISR.
    - Called from a critical zone.
  - SV#7, misplaced `chSysUnlockFromISR()`.
    - Not called from an ISR.
    - Not called from a critical zone.
  - SV#8, misplaced `CH_IRQ_PROLOGUE()`.
    - Not called at ISR begin.
    - Called from a critical zone.
  - SV#9, misplaced `CH_IRQ_EPILOGUE()`.
    - `CH_IRQ_PROLOGUE()` missing.
    - Not called at ISR end.
    - Called from a critical zone.
  - SV#10, misplaced I-class function.
    - I-class function not called from within a critical zone.
  - SV#11, misplaced S-class function.
    - S-class function not called from within a critical zone.
    - Called from an ISR.
- 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.

Debug related settings

- #define CH_DBG_STACK_FILL_VALUE 0x55

  Fill value for thread stack area in debug mode.

Macro Functions

- #define chDbgCheck(c)

  Function parameters check.
- #define chDbgAssert(c, r)

  Condition assertion.

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.

8.25.2 Macro Definition Documentation

8.25.2.1 #define CH_DBG_STACK_FILL_VALUE 0x55

Fill value for thread stack area in debug mode.

8.25.2.2 #define chDbgCheck(*c*)

Value:

```
do {                                                                \
  /*lint -save -e506 -e774 [2.1, 14.3] Can be a constant by design.*/       \
  if (CH_DBG_ENABLE_CHECKS != FALSE) {                                      \
    if (!(c)) {                                                             \
  /*lint -restore*/                                                         \
      chSysHalt(__func__);                                                  \
    }                                                                       \
  }                                                                         \
} while (false)
```

Function parameters 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	*c*	the condition to be verified to be true

Function Class:

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

8.25.2.3 #define chDbgAssert(*c*, *r*)

Value:

```
do {                                                                \
  /*lint -save -e506 -e774 [2.1, 14.3] Can be a constant by design.*/       \
  if (CH_DBG_ENABLE_ASSERTS != FALSE) {                                     \
    if (!(c)) {                                                             \
  /*lint -restore*/                                                         \
      chSysHalt(__func__);                                                  \
    }                                                                       \
  }                                                                         \
} while (false)
```

Condition assertion.

If the condition check fails then the kernel panics with a 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 remark string is not currently used except for putting a comment in the code about the assertion.

Parameters

in	*c*	the condition to be verified to be true
in	*r*	a remark string

Function Class:

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

8.25.3 Function Documentation

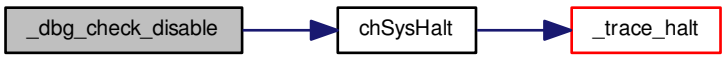
8.25.3.1 void _dbg_check_disable (void)

Guard code for `chSysDisable()`.

Function Class:

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

Here is the call graph for this function:



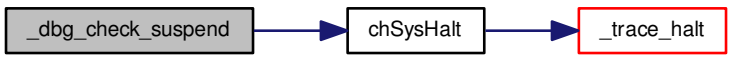
8.25.3.2 void _dbg_check_suspend (void)

Guard code for `chSysSuspend()`.

Function Class:

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

Here is the call graph for this function:



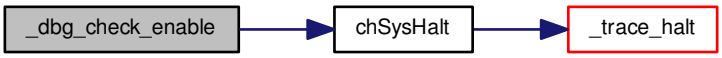
8.25.3.3 void _dbg_check_enable (void)

Guard code for `chSysEnable()`.

Function Class:

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

Here is the call graph for this function:



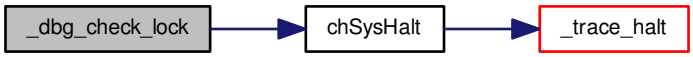
8.25.3.4 void _dbg_check_lock (void)

Guard code for `chSysLock()`.

Function Class:

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

Here is the call graph for this function:



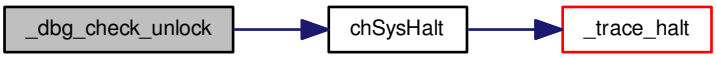
8.25.3.5 void _dbg_check_unlock (void)

Guard code for `chSysUnlock()`.

Function Class:

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

Here is the call graph for this function:



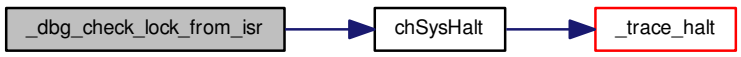
8.25.3.6 void _dbg_check_lock_from_isr (void)

Guard code for `chSysLockFromIsr()`.

Function Class:

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

Here is the call graph for this function:



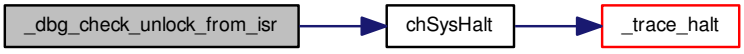
8.25.3.7 void _dbg_check_unlock_from_isr (void)

Guard code for `chSysUnlockFromIsr()`.

Function Class:

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

Here is the call graph for this function:



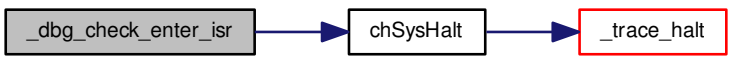
8.25.3.8 void _dbg_check_enter_isr (void)

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:



8.25.3.9 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:

8.25.3.10 void chDbgCheckClassI (void)

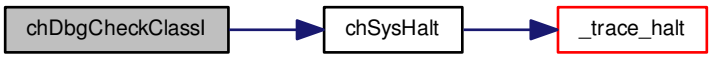
I-class functions context check.

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:



8.25.3.11 void chDbgCheckClassS (void)

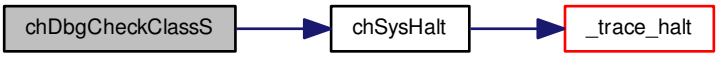
S-class functions context check.

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:



8.26 Trace

8.26.1 Detailed Description

System events tracing service.

Trace record types

- #define **CH_TRACE_TYPE_UNUSED** 0U
- #define **CH_TRACE_TYPE_SWITCH** 1U
- #define **CH_TRACE_TYPE_ISR_ENTER** 2U
- #define **CH_TRACE_TYPE_ISR_LEAVE** 3U
- #define **CH_TRACE_TYPE_HALT** 4U
- #define **CH_TRACE_TYPE_USER** 5U

Events to trace

- #define **CH_DBG_TRACE_MASK_DISABLED** 255U
- #define **CH_DBG_TRACE_MASK_NONE** 0U
- #define **CH_DBG_TRACE_MASK_SWITCH** 1U
- #define **CH_DBG_TRACE_MASK_ISR** 2U
- #define **CH_DBG_TRACE_MASK_HALT** 4U
- #define **CH_DBG_TRACE_MASK_USER** 8U
- #define **CH_DBG_TRACE_MASK_SLOW**
- #define **CH_DBG_TRACE_MASK_ALL**

Debug related settings

- #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_DISABLED

  Trace buffer entries.
- #define CH_DBG_TRACE_BUFFER_SIZE 128

  Trace buffer entries.

Data Structures

- struct ch_trace_event_t

  Trace buffer record.
- struct ch_trace_buffer_t

  Trace buffer header.

Functions

- static NOINLINE void trace_next (void)

  Writes a time stamp and increases the trace buffer pointer.
- void _trace_init (void)

  Trace circular buffer subsystem initialization.
- void _trace_switch (thread_t *ntp, thread_t *otp)

  Inserts in the circular debug trace buffer a context switch record.
- void _trace_isr_enter (const char *isr)

  Inserts in the circular debug trace buffer an ISR-enter record.

- void _trace_isr_leave (const char *isr)

  Inserts in the circular debug trace buffer an ISR-leave record.
- void _trace_halt (const char *reason)

  Inserts in the circular debug trace buffer an halt record.
- void chDbgWriteTraceI (void *up1, void *up2)

  Adds an user trace record to the trace buffer.
- void chDbgWriteTrace (void *up1, void *up2)

  Adds an user trace record to the trace buffer.
- void chDbgSuspendTraceI (uint16_t mask)

  Suspends one or more trace events.
- void chDbgSuspendTrace (uint16_t mask)

  Suspends one or more trace events.
- void chDbgResumeTraceI (uint16_t mask)

  Resumes one or more trace events.
- void chDbgResumeTrace (uint16_t mask)

  Resumes one or more trace events.

8.26.2 Macro Definition Documentation

8.26.2.1 #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_DISABLED

Trace buffer entries.

8.26.2.2 #define CH_DBG_TRACE_BUFFER_SIZE 128

Trace buffer entries.

Note

The trace buffer is only allocated if `CH_DBG_TRACE_MASK` is different from `CH_DBG_TRACE_MASK_D↩ISABLED`.

8.26.3 Function Documentation

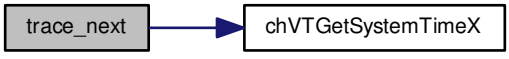
8.26.3.1 static NOINLINE void trace_next (void) [static]

Writes a time stamp and increases the trace buffer pointer.

Function Class:

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

Here is the call graph for this function:



8.26.3.2 void _trace_init (void)

Trace circular buffer subsystem initialization.

Note

Internal use only.

8.26.3.3 void _trace_switch (thread_t * *ntp*, thread_t * *otp*)

Inserts in the circular debug trace buffer a context switch record.

Parameters

in	*ntp*	the thread being switched in
in	*otp*	the thread being switched out

Function Class:

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

Here is the call graph for this function:

8.26.3.4 void _trace_isr_enter (const char * *isr*)

Inserts in the circular debug trace buffer an ISR-enter record.

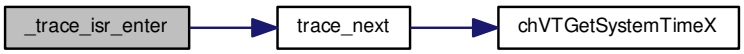
Parameters

in	*isr*	name of the isr

Function Class:

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

Here is the call graph for this function:



8.26.3.5 void _trace_isr_leave (const char * *isr*)

Inserts in the circular debug trace buffer an ISR-leave record.

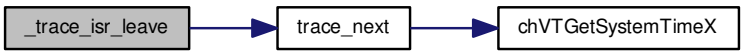
Parameters

in	*isr*	name of the isr

Function Class:

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

Here is the call graph for this function:



8.26.3.6 void _trace_halt (const char * *reason*)

Inserts in the circular debug trace buffer an halt record.

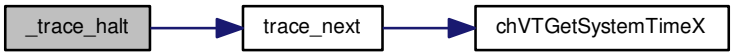
Parameters

in	*reason*	the halt error string

Function Class:

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

Here is the call graph for this function:



8.26.3.7 void chDbgWriteTraceI (void * *up1*, void * *up2*)

Adds an user trace record to the trace buffer.

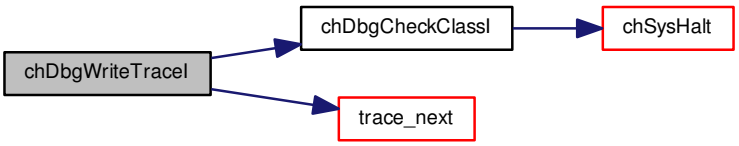
Parameters

in	*up1*	user parameter 1
in	*up2*	user parameter 2

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:



8.26.3.8 void chDbgWriteTrace (void * *up1*, void * *up2*)

Adds an user trace record to the trace buffer.

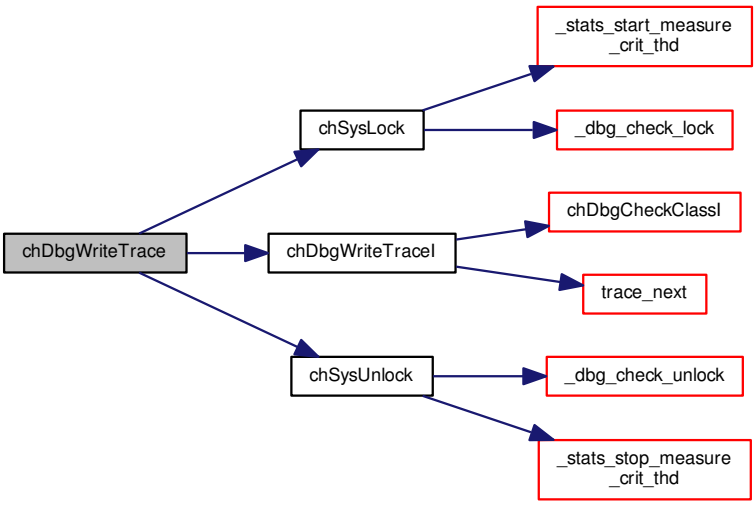
Parameters

in	*up1*	user parameter 1
in	*up2*	user parameter 2

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:



8.26.3.9 void chDbgSuspendTraceI (uint16_t *mask*)

Suspends one or more trace events.

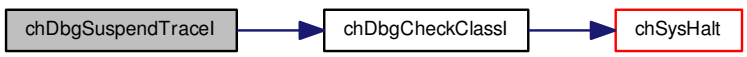
Parameters

in	*mask*	mask of the trace events to be suspended

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:



8.26.3.10 void chDbgSuspendTrace (uint16_t *mask*)

Suspends one or more trace events.

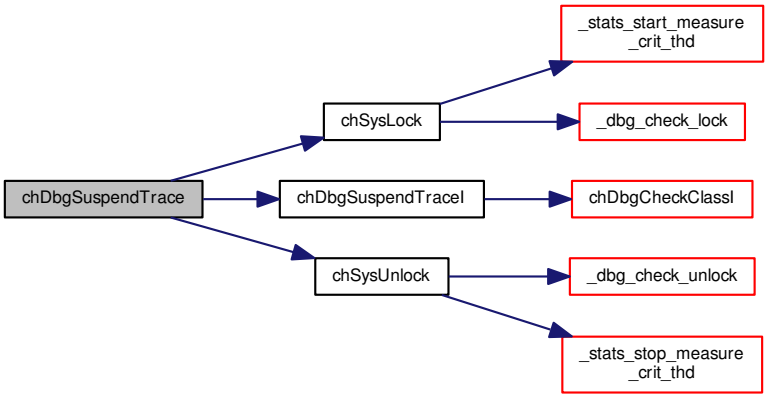
Parameters

in	*mask*	mask of the trace events to be suspended

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:



8.26.3.11 void chDbgResumeTraceI (uint16_t *mask*)

Resumes one or more trace events.

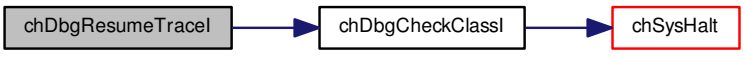
Parameters

in	*mask*	mask of the trace events to be resumed

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:



8.26.3.12 void chDbgResumeTrace (uint16_t *mask*)

Resumes one or more trace events.

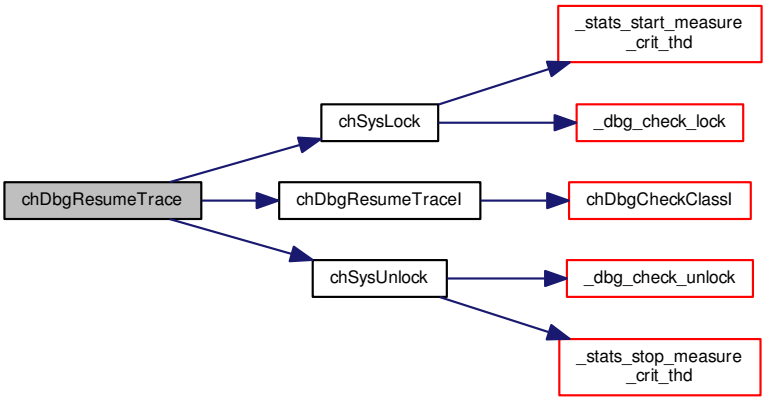
Parameters

in	*mask*	mask of the trace events to be resumed

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:



8.27 Time Measurement

8.27.1 Detailed Description

Time Measurement APIs and services.

Data Structures

- struct tm_calibration_t

  Type of a time measurement calibration data.
- struct time_measurement_t

  Type of a Time Measurement object.

Functions

- void _tm_init (void)

  Initializes the time measurement unit.
- void chTMObjectInit (time_measurement_t *tmp)

  Initializes a `TimeMeasurement` object.
- NOINLINE void chTMStartMeasurementX (time_measurement_t *tmp)

  Starts a measurement.
- NOINLINE void chTMStopMeasurementX (time_measurement_t *tmp)

  Stops a measurement.
- NOINLINE void chTMChainMeasurementToX (time_measurement_t *tmp1, time_measurement_t *tmp2)

  Stops a measurement and chains to the next one using the same time stamp.

8.27.2 Function Documentation

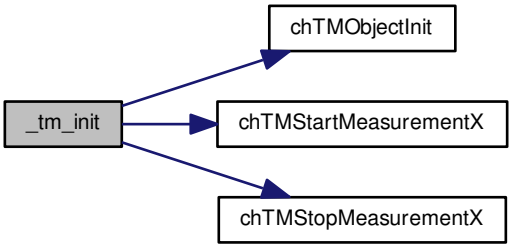
8.27.2.1 void _tm_init (void)

Initializes the time measurement unit.

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:



8.27.2.2 void chTMObjectInit (time_measurement_t * tmp)

Initializes a TimeMeasurement object.

Parameters

out	*tmp*	pointer to a TimeMeasurement structure

Function Class:

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

8.27.2.3 NOINLINE void chTMStartMeasurementX (time_measurement_t * tmp)

Starts a measurement.

Precondition

The time_measurement_t structure must be initialized.

Parameters

in,out	*tmp*	pointer to a TimeMeasurement structure

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.27.2.4 NOINLINE void chTMStopMeasurementX (time_measurement_t * tmp)

Stops a measurement.

Precondition

The time_measurement_t structure must be initialized.

Parameters

in,out	*tmp*	pointer to a time_measurement_t structure

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.27.2.5 NOINLINE void chTMChainMeasurementToX (time_measurement_t * tmp1, time_measurement_t * tmp2)

Stops a measurement and chains to the next one using the same time stamp.

Parameters

in,out	*tmp1*	pointer to the time_measurement_t structure to be stopped
in,out	*tmp2*	pointer to the time_measurement_t structure to be started

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.28 Statistics

8.28.1 Detailed Description

Statistics services.

Data Structures

- struct kernel_stats_t

  Type of a kernel statistics structure.

Functions

- void _stats_init (void)

  Initializes the statistics module.
- void _stats_increase_irq (void)

  Increases the IRQ counter.
- void _stats_ctxswc (thread_t *ntp, thread_t *otp)

  Updates context switch related statistics.
- void _stats_start_measure_crit_thd (void)

  Starts the measurement of a thread critical zone.
- void _stats_stop_measure_crit_thd (void)

  Stops the measurement of a thread critical zone.
- void _stats_start_measure_crit_isr (void)

  Starts the measurement of an ISR critical zone.
- void _stats_stop_measure_crit_isr (void)

  Stops the measurement of an ISR critical zone.

8.28.2 Function Documentation

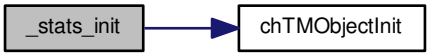
8.28.2.1 void _stats_init (void)

Initializes the statistics module.

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:



8.28.2.2 void _stats_increase_irq (void)

Increases the IRQ counter.

8.28.2.3 void _stats_ctxswc (thread_t * *ntp*, thread_t * *otp*)

Updates context switch related statistics.

Parameters

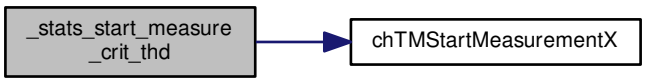
in	*ntp*	the thread to be switched in
in	*otp*	the thread to be switched out

Here is the call graph for this function:

8.28.2.4 void _stats_start_measure_crit_thd (void)

Starts the measurement of a thread critical zone.

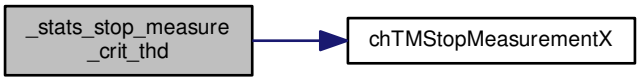
Here is the call graph for this function:



8.28.2.5 void _stats_stop_measure_crit_thd (void)

Stops the measurement of a thread critical zone.

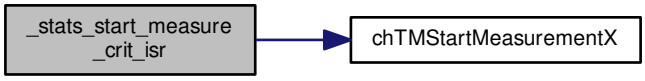
Here is the call graph for this function:



8.28.2.6 void _stats_start_measure_crit_isr (void)

Starts the measurement of an ISR critical zone.

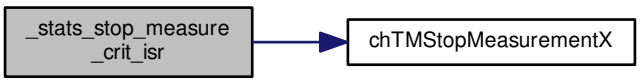
Here is the call graph for this function:



8.28.2.7 void _stats_stop_measure_crit_isr (void)

Stops the measurement of an ISR critical zone.

Here is the call graph for this function:



8.29 Port Layer

8.30 Time_intervals

8.30.1 Detailed Description

Macros

- #define CH_CFG_ST_RESOLUTION 32

  System time counter resolution.
- #define CH_CFG_ST_FREQUENCY 1000

  System tick frequency.
- #define CH_CFG_INTERVALS_SIZE 32

  Time intervals data size.
- #define CH_CFG_TIME_TYPES_SIZE 32

  Time types data size.

Special time constants

- #define TIME_IMMEDIATE ((sysinterval_t)0)

  Zero interval specification for some functions with a timeout specification.
- #define TIME_INFINITE ((sysinterval_t)-1)

  Infinite interval specification for all functions with a timeout specification.
- #define TIME_MAX_INTERVAL ((sysinterval_t)-2)

  Maximum interval constant usable as timeout.
- #define TIME_MAX_SYSTIME ((systime_t)-1)

  Maximum system of system time before it wraps.

Fast time conversion utilities

- #define TIME_S2I(secs) ((sysinterval_t)((time_conv_t)(secs) * (time_conv_t)CH_CFG_ST_FREQUENCY))

  Seconds to time interval.
- #define TIME_MS2I(msecs)

  Milliseconds to time interval.
- #define TIME_US2I(usecs)

  Microseconds to time interval.
- #define TIME_I2S(interval)

  Time interval to seconds.
- #define TIME_I2MS(interval)

  Time interval to milliseconds.
- #define TIME_I2US(interval)

  Time interval to microseconds.

Secure time conversion utilities

- static sysinterval_t chTimeS2I (time_secs_t secs)

  Seconds to time interval.
- static sysinterval_t chTimeMS2I (time_msecs_t msec)

  Milliseconds to time interval.
- static sysinterval_t chTimeUS2I (time_usecs_t usec)

  Microseconds to time interval.
- static time_secs_t chTimeI2S (sysinterval_t interval)

Time interval to seconds.

- static time_msecs_t chTimeI2MS (sysinterval_t interval)

  Time interval to milliseconds.

- static time_usecs_t chTimeI2US (sysinterval_t interval)

  Time interval to microseconds.

- static systime_t chTimeAddX (systime_t systime, sysinterval_t interval)

  Adds an interval to a system time returning a system time.

- static sysinterval_t chTimeDiffX (systime_t start, systime_t end)

  Subtracts two system times returning an interval.

- static bool chTimeIsInRangeX (systime_t time, systime_t start, systime_t end)

  Checks if the specified time is within the specified time range.

Typedefs

- typedef uint64_t systime_t

  Type of system time.

- typedef uint64_t sysinterval_t

  Type of time interval.

- typedef uint32_t time_secs_t

  Type of seconds.

- typedef uint32_t time_msecs_t

  Type of milliseconds.

- typedef uint32_t time_usecs_t

  Type of microseconds.

- typedef uint64_t time_conv_t

  Type of time conversion variable.

8.30.2 Macro Definition Documentation

8.30.2.1 #define TIME_IMMEDIATE ((sysinterval_t)0)

Zero interval specification for some functions with a timeout specification.

Note

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

8.30.2.2 #define TIME_INFINITE ((sysinterval_t)-1)

Infinite interval specification for all functions with a timeout specification.

Note

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

8.30.2.3 #define TIME_MAX_INTERVAL ((sysinterval_t)-2)

Maximum interval constant usable as timeout.

8.30.2.4 #define TIME_MAX_SYSTIME ((systime_t)-1)

Maximum system of system time before it wraps.

8.30.2.5 #define CH_CFG_ST_RESOLUTION 32

System time counter resolution.

Note

Allowed values are 16, 32 or 64 bits.

8.30.2.6 #define CH_CFG_ST_FREQUENCY 1000

System tick frequency.

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

8.30.2.7 #define CH_CFG_INTERVALS_SIZE 32

Time intervals data size.

Note

Allowed values are 16, 32 or 64 bits.

8.30.2.8 #define CH_CFG_TIME_TYPES_SIZE 32

Time types data size.

Note

Allowed values are 16 or 32 bits.

8.30.2.9 #define TIME_S2I(*secs*) ((sysinterval_t)((time_conv_t)(secs) * (time_conv_t)CH_CFG_ST_FREQUEN↩ CY))

Seconds to time interval.

Converts from seconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*secs*	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.

8.30.2.10 #define TIME_MS2I(*msecs*)

Value:

```
((sysinterval_t)((((time_conv_t)(msecs) *                                   \
                   (time_conv_t)CH_CFG_ST_FREQUENCY) +                        \
                  (time_conv_t)999) / (time_conv_t)1000))
```

Milliseconds to time interval.

Converts from milliseconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*msecs*	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.

8.30.2.11 #define TIME_US2I(*usecs*)

Value:

```
((sysinterval_t)((((time_conv_t)(usecs) *                                   \
                   (time_conv_t)CH_CFG_ST_FREQUENCY) +                        \
                  (time_conv_t)999999) / (time_conv_t)1000000))
```

Microseconds to time interval.

Converts from microseconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*usecs*	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.

8.30.2.12 #define TIME_I2S(*interval*)

Value:

```
(time_secs_t)(((time_conv_t)(interval) +                                    \
                  (time_conv_t)CH_CFG_ST_FREQUENCY -                          \
                  (time_conv_t)1) / (time_conv_t)
      CH_CFG_ST_FREQUENCY)
```

Time interval to seconds.

Converts from system ticks number to seconds.

Note

The result is rounded up to the next second boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*interval*	interval in ticks

Returns

The number of seconds.

Function Class:

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

8.30.2.13 #define TIME_I2MS(*interval*)

Value:

```
(time_msecs_t)((((time_conv_t)(interval) * (time_conv_t)1000) +
       \
                  (time_conv_t)CH_CFG_ST_FREQUENCY - (
      time_conv_t)1) /       \
                 (time_conv_t)CH_CFG_ST_FREQUENCY)
```

Time interval to milliseconds.

Converts from system ticks number to milliseconds.

Note

The result is rounded up to the next millisecond boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*interval*	interval in ticks

Returns

The number of milliseconds.

Function Class:

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

8.30.2.14 #define TIME_I2US(*interval*)

Value:

```
(time_msecs_t)((((time_conv_t)(interval) * (time_conv_t)1000000) +      \
                  (time_conv_t)CH_CFG_ST_FREQUENCY - (
      time_conv_t)1) /    \
                 (time_conv_t)CH_CFG_ST_FREQUENCY)
```

Time interval to microseconds.

Converts from system ticks number to microseconds.

Note

The result is rounded up to the next microsecond boundary.
Use of this macro for large values is not secure because integer overflows, make sure your value can be correctly converted.

Parameters

in	*interval*	interval in ticks

Returns

The number of microseconds.

Function Class:

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

8.30.3 Typedef Documentation

8.30.3.1 typedef uint64_t systime_t

Type of system time.

Note

It is selectable in configuration between 16, 32 or 64 bits.

8.30.3.2 typedef uint64_t sysinterval_t

Type of time interval.

Note

It is selectable in configuration between 16, 32 or 64 bits.

8.30.3.3 typedef uint32_t time_secs_t

Type of seconds.

Note

It is selectable in configuration between 16 or 32 bits.

8.30.3.4 typedef uint32_t time_msecs_t

Type of milliseconds.

Note

It is selectable in configuration between 16 or 32 bits.

8.30.3.5 typedef uint32_t time_usecs_t

Type of microseconds.

Note

It is selectable in configuration between 16 or 32 bits.

8.30.3.6 typedef uint64_t time_conv_t

Type of time conversion variable.

Note

This type must have double width than other time types, it is only used internally for conversions.

8.30.4 Function Documentation

8.30.4.1 static sysinterval_t chTimeS2I (time_secs_t *secs*) [inline], [static]

Seconds to time interval.

Converts from seconds to system ticks number.

Note

The result is rounded upward to the next tick boundary.

Parameters

in	*secs*	number of seconds

Returns

The number of ticks.

Function Class:

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

8.30.4.2 static sysinterval_t chTimeMS2I (time_msecs_t *msec*) [inline],[static]

Milliseconds to time interval.

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:

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

8.30.4.3 static sysinterval_t chTimeUS2I (time_usecs_t *usec*) [inline],[static]

Microseconds to time interval.

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:

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

8.30.4.4 static time_secs_t chTimeI2S (sysinterval_t *interval*) [inline],[static]

Time interval to seconds.

Converts from system interval to seconds.

Note

The result is rounded up to the next second boundary.

Parameters

in	*interval*	interval in ticks

Returns

The number of seconds.

Function Class:

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

8.30.4.5 static time_msecs_t chTimeI2MS (sysinterval_t *interval*) [inline],[static]

Time interval to milliseconds.

Converts from system interval to milliseconds.

Note

The result is rounded up to the next millisecond boundary.

Parameters

in	*interval*	interval in ticks

Returns

The number of milliseconds.

Function Class:

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

8.30.4.6 static time_usecs_t chTimeI2US (sysinterval_t *interval*) [inline],[static]

Time interval to microseconds.

Converts from system interval to microseconds.

Note

The result is rounded up to the next microsecond boundary.

Parameters

in	*interval*	interval in ticks

Returns

The number of microseconds.

Function Class:

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

8.30.4.7 static systime_t chTimeAddX (systime_t *systime,* sysinterval_t *interval*) [inline],[static]

Adds an interval to a system time returning a system time.

Parameters

in	*systime*	base system time
in	*interval*	interval to be added

Returns

The new system time.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.30.4.8 static sysinterval_t chTimeDiffX (systime_t *start,* systime_t *end*) [inline],[static]

Subtracts two system times returning an interval.

Parameters

in	*start*	first system time
in	*end*	second system time

Returns

The interval representing the time difference.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.30.4.9 static bool chTimeIsInRangeX (systime_t *time,* systime_t *start,* systime_t *end*) [inline],[static]

Checks if the specified time is within the specified time range.

Note

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

Parameters

in	*time*	the time to be verified
in	*start*	the start of the time window (inclusive)
in	*end*	the end of the time window (non inclusive)

Return values

true	current time within the specified time window.
false	current time not within the specified time window.

Function Class:

This is an **X-Class** API, this function can be invoked from any context.

8.31 Objects_factory

8.31.1 Detailed Description

The object factory is a subsystem that allows to:

- Register static objects by name.
- Dynamically create objects and assign them a name.
- Retrieve existing objects by name.
- Free objects by reference.

Allocated OS objects are handled using a reference counter, only when all references have been released then the object memory is freed in a pool.

Precondition

This subsystem requires the `CH_CFG_USE_MEMCORE` and `CH_CFG_USE_MEMPOOLS` options to be set to `TRUE`. The option `CH_CFG_USE_HEAP` is also required if the support for variable length objects is enabled.

Note

Compatible with RT and NIL.

Macros

- #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8

  Maximum length for object names.
- #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE

  Enables the registry of generic objects.
- #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE

  Enables factory for generic buffers.
- #define CH_CFG_FACTORY_SEMAPHORES TRUE

  Enables factory for semaphores.
- #define CH_CFG_FACTORY_SEMAPHORES FALSE

  Enables factory for semaphores.
- #define CH_CFG_FACTORY_MAILBOXES TRUE

  Enables factory for mailboxes.
- #define CH_CFG_FACTORY_MAILBOXES FALSE

  Enables factory for mailboxes.
- #define CH_CFG_FACTORY_OBJ_FIFOS TRUE

  Enables factory for objects FIFOs.
- #define CH_CFG_FACTORY_OBJ_FIFOS FALSE

  Enables factory for objects FIFOs.

Typedefs

- typedef struct ch_dyn_element dyn_element_t

  Type of a dynamic object list element.
- typedef struct ch_dyn_list dyn_list_t

  Type of a dynamic object list.

- typedef struct ch_registered_static_object registered_object_t
  Type of a registered object.
- typedef struct ch_dyn_object dyn_buffer_t
  Type of a dynamic buffer object.
- typedef struct ch_dyn_semaphore dyn_semaphore_t
  Type of a dynamic semaphore.
- typedef struct ch_dyn_mailbox dyn_mailbox_t
  Type of a dynamic buffer object.
- typedef struct ch_dyn_objects_fifo dyn_objects_fifo_t
  Type of a dynamic buffer object.
- typedef struct ch_objects_factory objects_factory_t
  Type of the factory main object.

Data Structures

- struct ch_dyn_element
  Type of a dynamic object list element.
- struct ch_dyn_list
  Type of a dynamic object list.
- struct ch_registered_static_object
  Type of a registered object.
- struct ch_dyn_object
  Type of a dynamic buffer object.
- struct ch_dyn_semaphore
  Type of a dynamic semaphore.
- struct ch_dyn_mailbox
  Type of a dynamic buffer object.
- struct ch_dyn_objects_fifo
  Type of a dynamic buffer object.
- struct ch_objects_factory
  Type of the factory main object.

Functions

- void _factory_init (void)
  Initializes the objects factory.
- registered_object_t * chFactoryRegisterObject (const char *name, void *objp)
  Registers a generic object.
- registered_object_t * chFactoryFindObject (const char *name)
  Retrieves a registered object.
- registered_object_t * chFactoryFindObjectByPointer (void *objp)
  Retrieves a registered object by pointer.
- void chFactoryReleaseObject (registered_object_t *rop)
  Releases a registered object.
- dyn_buffer_t * chFactoryCreateBuffer (const char *name, size_t size)
  Creates a generic dynamic buffer object.
- dyn_buffer_t * chFactoryFindBuffer (const char *name)
  Retrieves a dynamic buffer object.
- void chFactoryReleaseBuffer (dyn_buffer_t *dbp)
  Releases a dynamic buffer object.

- dyn_semaphore_t * chFactoryCreateSemaphore (const char *name, cnt_t n)

  Creates a dynamic semaphore object.
- dyn_semaphore_t * chFactoryFindSemaphore (const char *name)

  Retrieves a dynamic semaphore object.
- void chFactoryReleaseSemaphore (dyn_semaphore_t *dsp)

  Releases a dynamic semaphore object.
- dyn_mailbox_t * chFactoryCreateMailbox (const char *name, size_t n)

  Creates a dynamic mailbox object.
- dyn_mailbox_t * chFactoryFindMailbox (const char *name)

  Retrieves a dynamic mailbox object.
- void chFactoryReleaseMailbox (dyn_mailbox_t *dmp)

  Releases a dynamic mailbox object.
- dyn_objects_fifo_t * chFactoryCreateObjectsFIFO (const char *name, size_t objsize, size_t objn, unsigned objalign)

  Creates a dynamic "objects FIFO" object.
- dyn_objects_fifo_t * chFactoryFindObjectsFIFO (const char *name)

  Retrieves a dynamic "objects FIFO" object.
- void chFactoryReleaseObjectsFIFO (dyn_objects_fifo_t *dofp)

  Releases a dynamic "objects FIFO" object.
- static dyn_element_t * chFactoryDuplicateReference (dyn_element_t *dep)

  Duplicates an object reference.
- static void * chFactoryGetObject (registered_object_t *rop)

  Returns the pointer to the inner registered object.
- static size_t chFactoryGetBufferSize (dyn_buffer_t *dbp)

  Returns the size of a generic dynamic buffer object.
- static uint8_t * chFactoryGetBuffer (dyn_buffer_t *dbp)

  Returns the pointer to the inner buffer.
- static semaphore_t * chFactoryGetSemaphore (dyn_semaphore_t *dsp)

  Returns the pointer to the inner semaphore.
- static mailbox_t * chFactoryGetMailbox (dyn_mailbox_t *dmp)

  Returns the pointer to the inner mailbox.
- static objects_fifo_t * chFactoryGetObjectsFIFO (dyn_objects_fifo_t *dofp)

  Returns the pointer to the inner objects FIFO.

Variables

- objects_factory_t ch_factory

  Factory object static instance.

8.31.2 Macro Definition Documentation

8.31.2.1 #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8

Maximum length for object names.

If the specified length is zero then the name is stored by pointer but this could have unintended side effects.

8.31.2.2 #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE

Enables the registry of generic objects.

8.31.2.3 #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE

Enables factory for generic buffers.

8.31.2.4 #define CH_CFG_FACTORY_SEMAPHORES TRUE

Enables factory for semaphores.

8.31.2.5 #define CH_CFG_FACTORY_SEMAPHORES FALSE

Enables factory for semaphores.

8.31.2.6 #define CH_CFG_FACTORY_MAILBOXES TRUE

Enables factory for mailboxes.

8.31.2.7 #define CH_CFG_FACTORY_MAILBOXES FALSE

Enables factory for mailboxes.

8.31.2.8 #define CH_CFG_FACTORY_OBJ_FIFOS TRUE

Enables factory for objects FIFOs.

8.31.2.9 #define CH_CFG_FACTORY_OBJ_FIFOS FALSE

Enables factory for objects FIFOs.

8.31.3 Typedef Documentation

8.31.3.1 typedef struct ch_dyn_element dyn_element_t

Type of a dynamic object list element.

8.31.3.2 typedef struct ch_dyn_list dyn_list_t

Type of a dynamic object list.

8.31.3.3 typedef struct ch_registered_static_object registered_object_t

Type of a registered object.

8.31.3.4 typedef struct ch_dyn_object dyn_buffer_t

Type of a dynamic buffer object.

8.31.3.5 typedef struct ch_dyn_semaphore dyn_semaphore_t

Type of a dynamic semaphore.

8.31.3.6 typedef struct ch_dyn_mailbox dyn_mailbox_t

Type of a dynamic buffer object.

8.31.3.7 typedef struct ch_dyn_objects_fifo dyn_objects_fifo_t

Type of a dynamic buffer object.

8.31.3.8 typedef struct ch_objects_factory objects_factory_t

Type of the factory main object.

8.31.4 Function Documentation

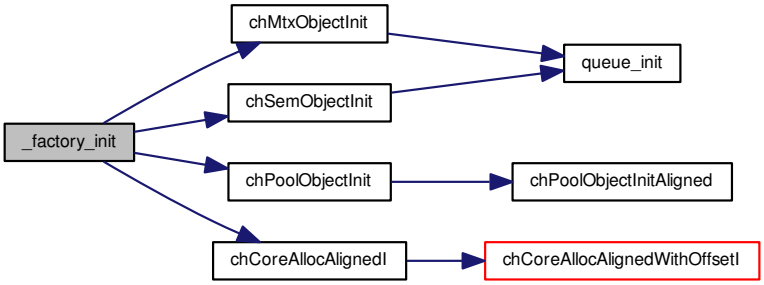
8.31.4.1 void _factory_init (void)

Initializes the objects factory.

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:



8.31.4.2 registered_object_t * chFactoryRegisterObject (const char * *name*, void * *objp*)

Registers a generic object.

Postcondition

A reference to the registered object is returned and the reference counter is initialized to one.

Parameters

in	*name*	name to be assigned to the registered object
in	*objp*	pointer to the object to be registered

Returns

The reference to the registered object.

Return values

NULL	if the object to be registered cannot be allocated or a registered object with the same name exists.

Function Class:

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

8.31.4.3 registered_object_t * chFactoryFindObject (const char * *name*)

Retrieves a registered object.

Postcondition

A reference to the registered object is returned with the reference counter increased by one.

Parameters

in	*name*	name of the registered object

Returns

The reference to the found registered object.

Return values

NULL	if a registered object with the specified name does not exist.

Function Class:

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

8.31.4.4 registered_object_t * chFactoryFindObjectByPointer (void * *objp*)

Retrieves a registered object by pointer.

Postcondition

A reference to the registered object is returned with the reference counter increased by one.

Parameters

in	*objp*	pointer to the object to be retrieved

Returns

The reference to the found registered object.

Return values

NULL	if a registered object with the specified pointer does not exist.

Function Class:

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

8.31.4.5 void chFactoryReleaseObject (registered_object_t * *rop*)

Releases a registered object.

The reference counter of the registered object is decreased by one, if reaches zero then the registered object memory is freed.

Note

The object itself is not freed, it could be static, only the allocated list element is freed.

Parameters

in	*rop*	registered object reference

Function Class:

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

8.31.4.6 dyn_buffer_t * chFactoryCreateBuffer (const char * *name*, size_t *size*)

Creates a generic dynamic buffer object.

Postcondition

A reference to the dynamic buffer object is returned and the reference counter is initialized to one.
The dynamic buffer object is filled with zeros.

Parameters

in	*name*	name to be assigned to the new dynamic buffer object
in	*size*	payload size of the dynamic buffer object to be created

Returns

The reference to the created dynamic buffer object.

Return values

NULL	if the dynamic buffer object cannot be allocated or a dynamic buffer object with the same name exists.

Function Class:

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

8.31.4.7 dyn_buffer_t * chFactoryFindBuffer (const char * *name*)

Retrieves a dynamic buffer object.

Postcondition

A reference to the dynamic buffer object is returned with the reference counter increased by one.

Parameters

in	*name*	name of the dynamic buffer object

Returns

The reference to the found dynamic buffer object.

Return values

NULL	if a dynamic buffer object with the specified name does not exist.

Function Class:

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

8.31.4.8 void chFactoryReleaseBuffer (dyn_buffer_t * *dbp*)

Releases a dynamic buffer object.

The reference counter of the dynamic buffer object is decreased by one, if reaches zero then the dynamic buffer object memory is freed.

Parameters

in	*dbp*	dynamic buffer object reference

Function Class:

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

8.31.4.9 dyn_semaphore_t * chFactoryCreateSemaphore (const char * *name*, cnt_t *n*)

Creates a dynamic semaphore object.

Postcondition

A reference to the dynamic semaphore object is returned and the reference counter is initialized to one.
The dynamic semaphore object is initialized and ready to use.

Parameters

in	*name*	name to be assigned to the new dynamic semaphore object
in	*n*	dynamic semaphore object counter initialization value

Returns

The reference to the created dynamic semaphore object.

Return values

NULL	if the dynamic semaphore object cannot be allocated or a dynamic semaphore with the same name exists.

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:

8.31.4.10 dyn_semaphore_t * chFactoryFindSemaphore (const char * *name*)

Retrieves a dynamic semaphore object.

Postcondition

A reference to the dynamic semaphore object is returned with the reference counter increased by one.

Parameters

in	*name*	name of the dynamic semaphore object

Returns

The reference to the found dynamic semaphore object.

Return values

NULL	if a dynamic semaphore object with the specified name does not exist.

Function Class:

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

8.31.4.11 void chFactoryReleaseSemaphore (dyn_semaphore_t * *dsp*)

Releases a dynamic semaphore object.

The reference counter of the dynamic semaphore object is decreased by one, if reaches zero then the dynamic semaphore object memory is freed.

Parameters

in	*dsp*	dynamic semaphore object reference

Function Class:

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

8.31.4.12 dyn_mailbox_t * chFactoryCreateMailbox (const char * *name*, size_t *n*)

Creates a dynamic mailbox object.

Postcondition

A reference to the dynamic mailbox object is returned and the reference counter is initialized to one.
The dynamic mailbox object is initialized and ready to use.

Parameters

in	*name*	name to be assigned to the new dynamic mailbox object
in	*n*	mailbox buffer size as number of messages

Returns

The reference to the created dynamic mailbox object.

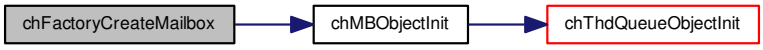
Return values

NULL	if the dynamic mailbox object cannot be allocated or a dynamic mailbox object with the same name exists.

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:



8.31.4.13 dyn_mailbox_t * chFactoryFindMailbox (const char * *name*)

Retrieves a dynamic mailbox object.

Postcondition

A reference to the dynamic mailbox object is returned with the reference counter increased by one.

Parameters

in	*name*	name of the dynamic mailbox object

Returns

The reference to the found dynamic mailbox object.

Return values

NULL	if a dynamic mailbox object with the specified name does not exist.

Function Class:

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

8.31.4.14 void chFactoryReleaseMailbox (dyn_mailbox_t * *dmp*)

Releases a dynamic mailbox object.

The reference counter of the dynamic mailbox object is decreased by one, if reaches zero then the dynamic mailbox object memory is freed.

Parameters

in	*dmp*	dynamic mailbox object reference

Function Class:

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

8.31.4.15 dyn_objects_fifo_t * chFactoryCreateObjectsFIFO (const char * *name*, size_t *objsize*, size_t *objn*, unsigned *objalign*)

Creates a dynamic "objects FIFO" object.

Postcondition

A reference to the dynamic "objects FIFO" object is returned and the reference counter is initialized to one. The dynamic "objects FIFO" object is initialized and ready to use.

Parameters

in	*name*	name to be assigned to the new dynamic "objects FIFO" object
in	*objsize*	size of objects
in	*objn*	number of objects available
in	*objalign*	required objects alignment

Returns

The reference to the created dynamic "objects FIFO" object.

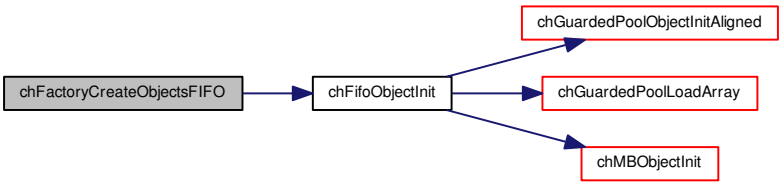
Return values

NULL	if the dynamic "objects FIFO" object cannot be allocated or a dynamic "objects FIFO" object with the same name exists.

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:



8.31.4.16 dyn_objects_fifo_t * chFactoryFindObjectsFIFO (const char * *name*)

Retrieves a dynamic "objects FIFO" object.

Postcondition

A reference to the dynamic "objects FIFO" object is returned with the reference counter increased by one.

Parameters

in	*name*	name of the dynamic "objects FIFO" object

Returns

The reference to the found dynamic "objects FIFO" object.

Return values

NULL	if a dynamic "objects FIFO" object with the specified name does not exist.

Function Class:

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

8.31.4.17 void chFactoryReleaseObjectsFIFO (dyn_objects_fifo_t * *dofp*)

Releases a dynamic "objects FIFO" object.

The reference counter of the dynamic "objects FIFO" object is decreased by one, if reaches zero then the dynamic "objects FIFO" object memory is freed.

Parameters

in	*dofp*	dynamic "objects FIFO" object reference

Function Class:

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

8.31.4.18 static dyn_element_t* chFactoryDuplicateReference (dyn_element_t * *dep*) [inline], [static]

Duplicates an object reference.

Note

This function can be used on any kind of dynamic object.

Parameters

in	*dep*	pointer to the element field of the object

Returns

The duplicated object reference.

Function Class:

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

8.31.4.19 static void* chFactoryGetObject (registered_object_t * rop) [inline], [static]

Returns the pointer to the inner registered object.

Parameters

in	*rop*	registered object reference

Returns

The pointer to the registered object.

Function Class:

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

8.31.4.20 static size_t chFactoryGetBufferSize (dyn_buffer_t * dbp) [inline], [static]

Returns the size of a generic dynamic buffer object.

Parameters

in	*dbp*	dynamic buffer object reference

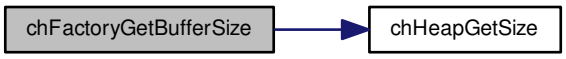
Returns

The size of the buffer object in bytes.

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:



8.31.4.21 static uint8_t* chFactoryGetBuffer (dyn_buffer_t * dbp) [inline], [static]

Returns the pointer to the inner buffer.

Parameters

in	*dbp*	dynamic buffer object reference

Returns

The pointer to the dynamic buffer.

Function Class:

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

8.31.4.22 static semaphore_t* chFactoryGetSemaphore (dyn_semaphore_t * *dsp*) [inline], [static]

Returns the pointer to the inner semaphore.

Parameters

in	*dsp*	dynamic semaphore object reference

Returns

The pointer to the semaphore.

Function Class:

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

8.31.4.23 static mailbox_t* chFactoryGetMailbox (dyn_mailbox_t * *dmp*) [inline], [static]

Returns the pointer to the inner mailbox.

Parameters

in	*dmp*	dynamic mailbox object reference

Returns

The pointer to the mailbox.

Function Class:

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

8.31.4.24 static objects_fifo_t* chFactoryGetObjectsFIFO (dyn_objects_fifo_t * *dofp*) [inline], [static]

Returns the pointer to the inner objects FIFO.

Parameters

in	*dofp*	dynamic "objects FIFO" object reference

Returns

The pointer to the objects FIFO.

Function Class:

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

8.31.5 Variable Documentation

8.31.5.1 objects_factory_t ch_factory

Factory object static instance.

Note

It is a global object because it could be accessed through a specific debugger plugin.

8.32 Objects_fifo

8.32.1 Detailed Description

Typedefs

- typedef struct ch_objects_fifo objects_fifo_t

  Type of an objects FIFO.

Data Structures

- struct ch_objects_fifo

  Type of an objects FIFO.

Functions

- static void chFifoObjectInit (objects_fifo_t *ofp, size_t objsize, size_t objn, unsigned objalign, void *objbuf, msg_t *msgbuf)

  Initializes a FIFO object.
- static void * chFifoTakeObjectI (objects_fifo_t *ofp)

  Allocates a free object.
- static void * chFifoTakeObjectTimeoutS (objects_fifo_t *ofp, sysinterval_t timeout)

  Allocates a free object.
- static void * chFifoTakeObjectTimeout (objects_fifo_t *ofp, sysinterval_t timeout)

  Allocates a free object.
- static void chFifoReturnObjectI (objects_fifo_t *ofp, void *objp)

  Releases a fetched object.
- static void chFifoReturnObject (objects_fifo_t *ofp, void *objp)

  Releases a fetched object.
- static void chFifoSendObjectI (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static void chFifoSendObjectS (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static void chFifoSendObject (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static msg_t chFifoReceiveObjectI (objects_fifo_t *ofp, void **objpp)

  Fetches an object.
- static msg_t chFifoReceiveObjectTimeoutS (objects_fifo_t *ofp, void **objpp, sysinterval_t timeout)

  Fetches an object.
- static msg_t chFifoReceiveObjectTimeout (objects_fifo_t *ofp, void **objpp, sysinterval_t timeout)

  Fetches an object.

8.32.2 Typedef Documentation

8.32.2.1 typedef struct ch_objects_fifo objects_fifo_t

Type of an objects FIFO.

8.32.3 Function Documentation

8.32.3.1 static void chFifoObjectInit (objects_fifo_t * *ofp*, size_t *objsize*, size_t *objn*, unsigned *objalign*, void * *objbuf*, msg_t * *msgbuf*) [inline], [static]

Initializes a FIFO object.

Precondition

The messages size must be a multiple of the alignment requirement.

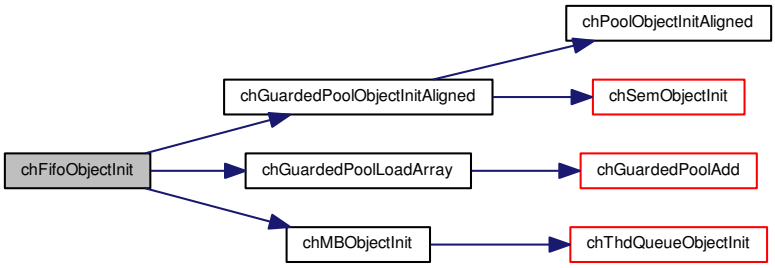
Parameters

out	*ofp*	pointer to a objects_fifo_t structure
in	*objsize*	size of objects
in	*objn*	number of objects available
in	*objalign*	required objects alignment
in	*objbuf*	pointer to the buffer of objects, it must be able to hold objn objects of objsize size with objealign alignment
in	*msgbuf*	pointer to the buffer of messages, it must be able to hold objn messages

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:



8.32.3.2 static void* chFifoTakeObjectI (objects_fifo_t * *ofp*) [inline], [static]

Allocates a free object.

Parameters

in	*ofp*	pointer to a objects_fifo_t structure

Returns

The pointer to the allocated object.

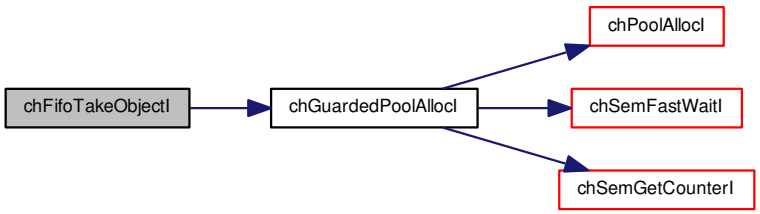
Return values

NULL	if an object is not immediately available.

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:



8.32.3.3 static void* chFifoTakeObjectTimeoutS (objects_fifo_t * *ofp*, sysinterval_t *timeout*) [inline], [static]

Allocates a free object.

Parameters

in	*ofp*	pointer to a `objects_fifo_t` structure
in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The pointer to the allocated object.

Return values

NULL	if an object is not available 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:

8.32.3.4 static void* chFifoTakeObjectTimeout (objects_fifo_t * *ofp*, sysinterval_t *timeout*) [inline], [static]

Allocates a free object.

Parameters

in	*ofp*	pointer to a `objects_fifo_t` structure
in	*timeout*	the number of ticks before the operation timeouts, the following special values are allowed: • *TIME_IMMEDIATE* immediate timeout. • *TIME_INFINITE* no timeout.

Returns

The pointer to the allocated object.

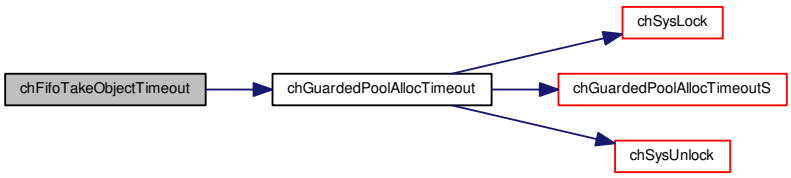
Return values

NULL	if an object is not available 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:



8.32.3.5 static void chFifoReturnObjectI (objects_fifo_t * ofp, void * objp) [inline], [static]

Releases a fetched object.

Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objp*	pointer to the object to be released

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:

8.32.3.6 static void chFifoReturnObject (objects_fifo_t * ofp, void * objp) [inline], [static]

Releases a fetched object.

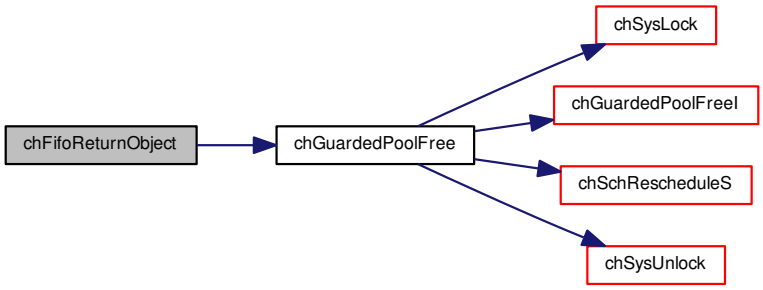
Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objp*	pointer to the object to be released

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:



8.32.3.7 static void chFifoSendObjectI (objects_fifo_t * *ofp*, void * *objp*) [inline], [static]

Posts an object.

Note

By design the object can be always immediately posted.

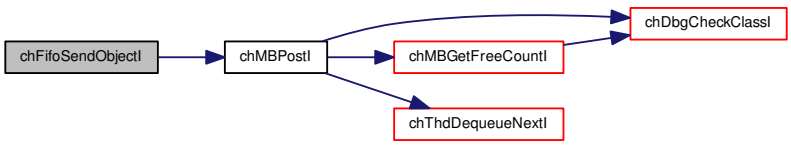
Parameters

in	*ofp*	pointer to a `objects_fifo_t` structure
in	*objp*	pointer to the object to 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:



8.32.3.8 static void chFifoSendObjectS (objects_fifo_t * *ofp*, void * *objp*) [inline], [static]

Posts an object.

Note

By design the object can be always immediately posted.

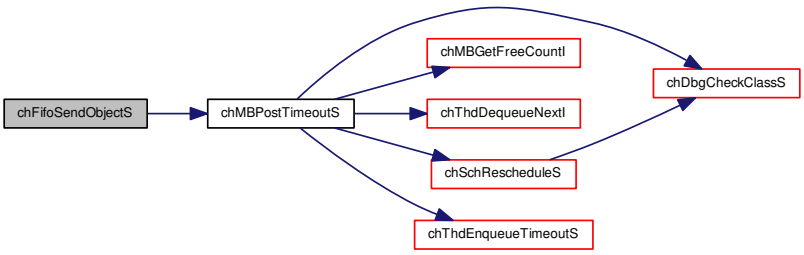
Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objp*	pointer to the object to be posted

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:



8.32.3.9 static void chFifoSendObject (objects_fifo_t * *ofp*, void * *objp*) [inline], [static]

Posts an object.

Note

By design the object can be always immediately posted.

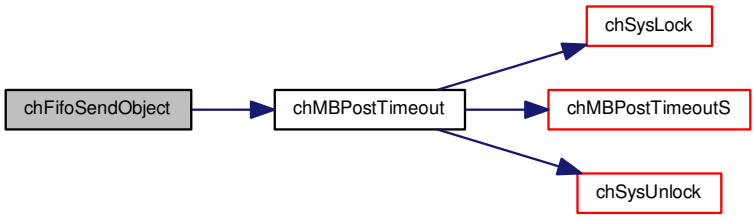
Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objp*	pointer to the object to be released

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:



8.32.3.10 static msg_t chFifoReceiveObjectI (objects_fifo_t * *ofp*, void ** *objpp*) [inline], [static]

Fetches an object.

Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objpp*	pointer to the fetched object reference

Returns

The operation status.

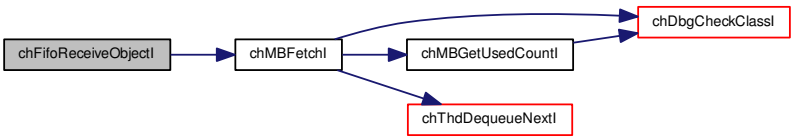
Return values

MSG_OK	if an object has been correctly fetched.
MSG_TIMEOUT	if the FIFO 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:



8.32.3.11 static msg_t chFifoReceiveObjectTimeoutS (objects_fifo_t * *ofp*, void ** *objpp*, sysinterval_t *timeout*) [inline], [static]

Fetches an object.

Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objpp*	pointer to the fetched object reference
in	*timeout*	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.

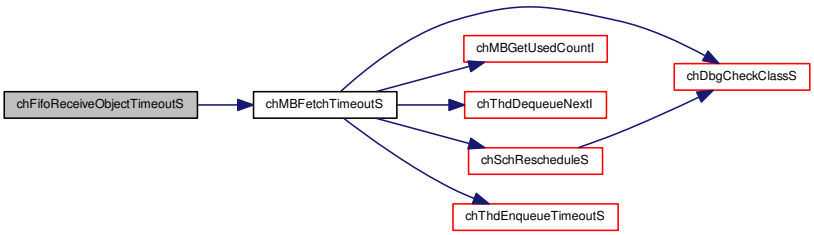
Return values

MSG_OK	if an object has been correctly fetched.
MSG_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:



8.32.3.12 static msg_t chFifoReceiveObjectTimeout (objects_fifo_t * *ofp*, void ** *objpp*, sysinterval_t *timeout*) [inline], [static]

Fetches an object.

Parameters

in	*ofp*	pointer to a objects_fifo_t structure
in	*objpp*	pointer to the fetched object reference

Parameters

in	*timeout*	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.

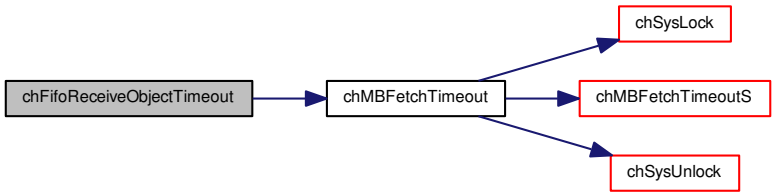
Return values

MSG_OK	if an object has been correctly fetched.
MSG_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:



Chapter 9

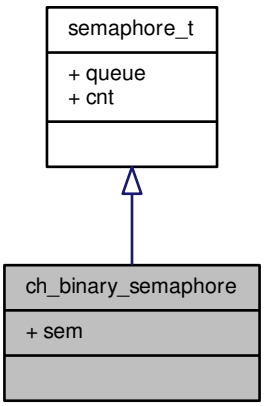
Data Structure Documentation

9.1 ch_binary_semaphore Struct Reference

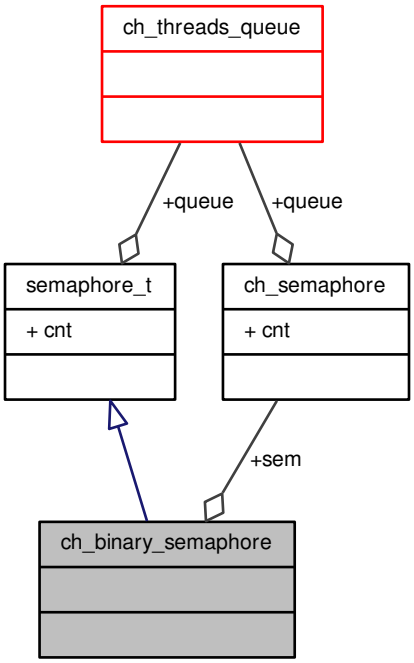
Binary semaphore type.

`#include <chbsem.h>`

Inheritance diagram for ch_binary_semaphore:



Collaboration diagram for ch_binary_semaphore:



Additional Inherited Members

9.1.1 Detailed Description

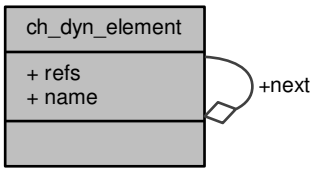
Binary semaphore type.

9.2 ch_dyn_element Struct Reference

Type of a dynamic object list element.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_element:



Data Fields

- struct ch_dyn_element * next

  Next dynamic object in the list.

- ucnt_t refs

  Number of references to this object.

9.2.1 Detailed Description

Type of a dynamic object list element.

9.2.2 Field Documentation

9.2.2.1 struct ch_dyn_element* ch_dyn_element::next

Next dynamic object in the list.

9.2.2.2 ucnt_t ch_dyn_element::refs

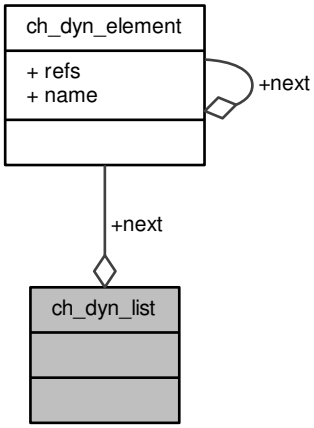
Number of references to this object.

9.3 ch_dyn_list Struct Reference

Type of a dynamic object list.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_list:



9.3.1 Detailed Description

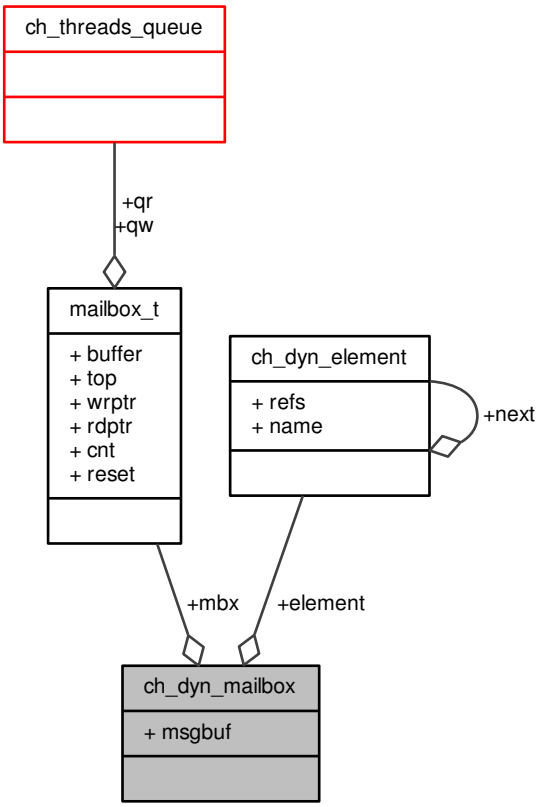
Type of a dynamic object list.

9.4 ch_dyn_mailbox Struct Reference

Type of a dynamic buffer object.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_mailbox:



Data Fields

- dyn_element_t element

  List element of the dynamic buffer object.
- mailbox_t mbx

  The mailbox.
- msg_t msgbuf [ ]

  Messages buffer.

9.4.1 Detailed Description

Type of a dynamic buffer object.

9.4.2 Field Documentation

9.4.2.1 dyn_element_t ch_dyn_mailbox::element

List element of the dynamic buffer object.

9.4.2.2 mailbox_t ch_dyn_mailbox::mbx

The mailbox.

9.4.2.3 msg_t ch_dyn_mailbox::msgbuf[ ]

Messages buffer.

Note

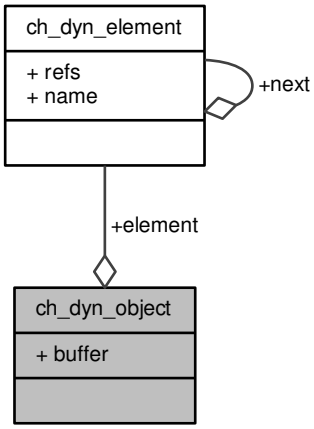
This requires C99.

9.5 ch_dyn_object Struct Reference

Type of a dynamic buffer object.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_object:



Data Fields

- dyn_element_t element

  List element of the dynamic buffer object.
- uint8_t buffer [ ]

  The buffer.

9.5.1 Detailed Description

Type of a dynamic buffer object.

9.5.2 Field Documentation

9.5.2.1 dyn_element_t ch_dyn_object::element

List element of the dynamic buffer object.

9.5.2.2 uint8_t ch_dyn_object::buffer[ ]

The buffer.

Note

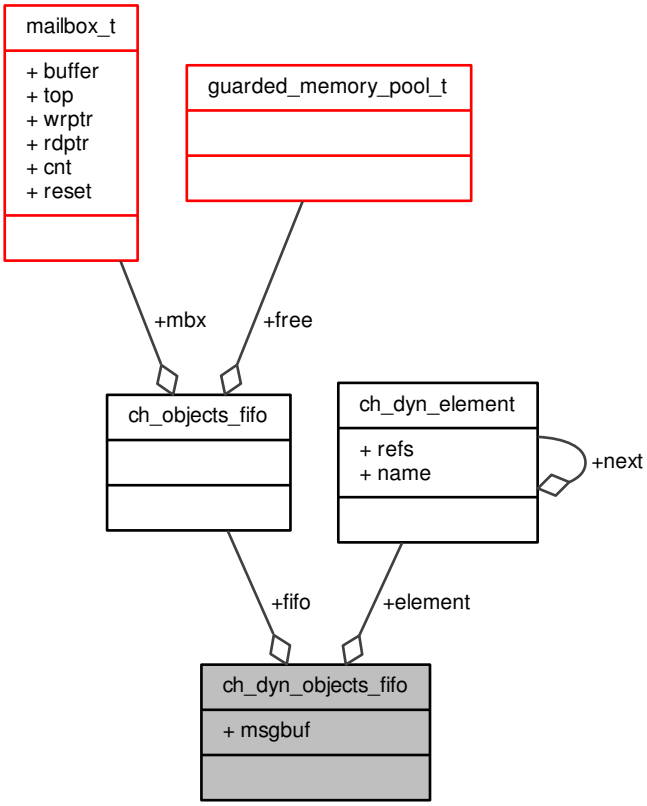
This requires C99.

9.6 ch_dyn_objects_fifo Struct Reference

Type of a dynamic buffer object.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_objects_fifo:



Data Fields

- dyn_element_t element

  List element of the dynamic buffer object.

- objects_fifo_t fifo

  The objects FIFO.

- msg_t msgbuf [ ]

  Messages buffer.

9.6.1 Detailed Description

Type of a dynamic buffer object.

9.6.2 Field Documentation

9.6.2.1 dyn_element_t ch_dyn_objects_fifo::element

List element of the dynamic buffer object.

9.6.2.2 objects_fifo_t ch_dyn_objects_fifo::fifo

The objects FIFO.

9.6.2.3 msg_t ch_dyn_objects_fifo::msgbuf[ ]

Messages buffer.

Note

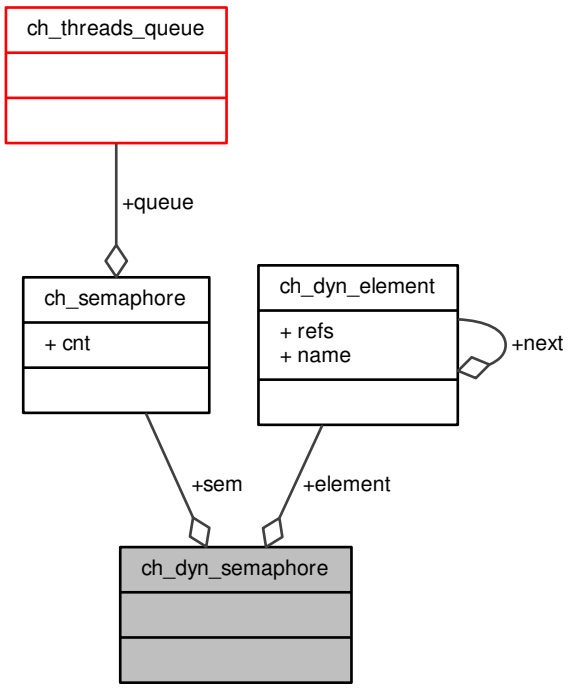
This open array is followed by another area containing the objects, this area is not represented in this structure. This requires C99.

9.7 ch_dyn_semaphore Struct Reference

Type of a dynamic semaphore.

`#include <chfactory.h>`

Collaboration diagram for ch_dyn_semaphore:



Data Fields

- dyn_element_t element

  List element of the dynamic semaphore.
- semaphore_t sem

  The semaphore.

9.7.1 Detailed Description

Type of a dynamic semaphore.

9.7.2 Field Documentation

9.7.2.1 dyn_element_t ch_dyn_semaphore::element

List element of the dynamic semaphore.

9.7.2.2 semaphore_t ch_dyn_semaphore::sem

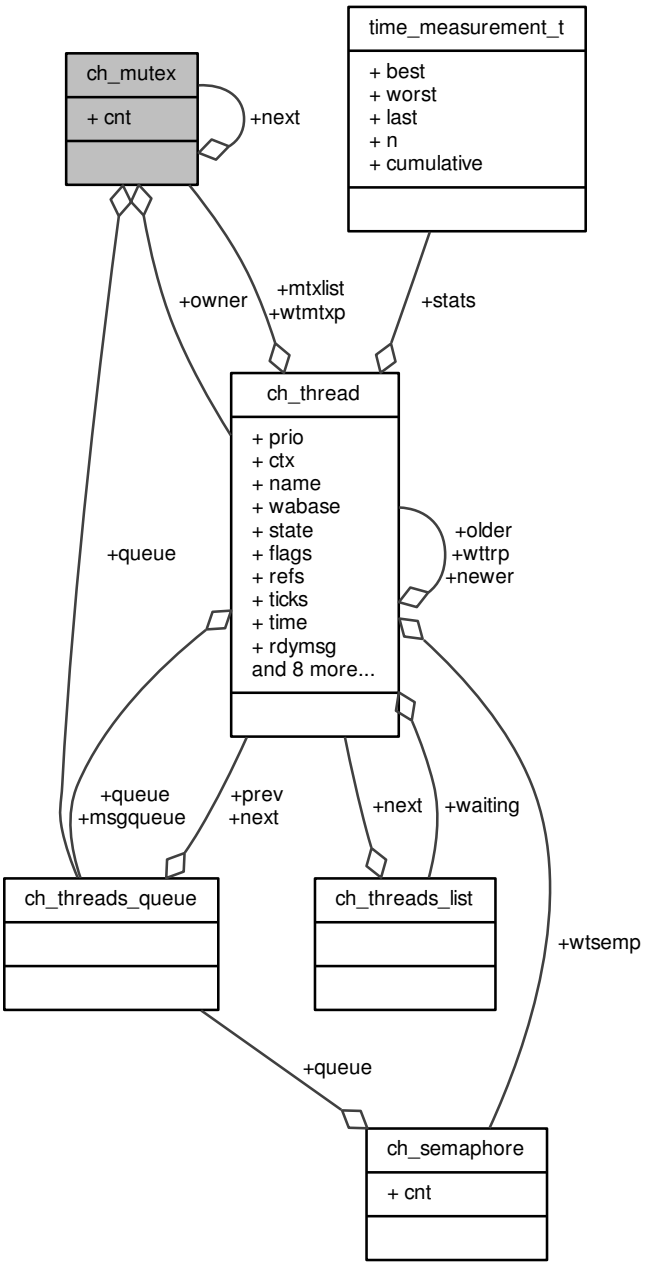
The semaphore.

9.8 ch_mutex Struct Reference

Mutex structure.

`#include <chmtx.h>`

Collaboration diagram for ch_mutex:



Data Fields

- threads_queue_t queue

  Queue of the threads sleeping on this mutex.
- thread_t * owner

  Owner `thread_t` *pointer or* `NULL`.
- mutex_t * next

Next `mutex_t` *into an owner-list or* `NULL`*.*

- cnt_t cnt

  Mutex recursion counter.

9.8.1 Detailed Description

Mutex structure.

9.8.2 Field Documentation

9.8.2.1 threads_queue_t ch_mutex::queue

Queue of the threads sleeping on this mutex.

9.8.2.2 thread_t* ch_mutex::owner

Owner `thread_t` pointer or `NULL`.

9.8.2.3 mutex_t* ch_mutex::next

Next `mutex_t` into an owner-list or `NULL`.

9.8.2.4 cnt_t ch_mutex::cnt

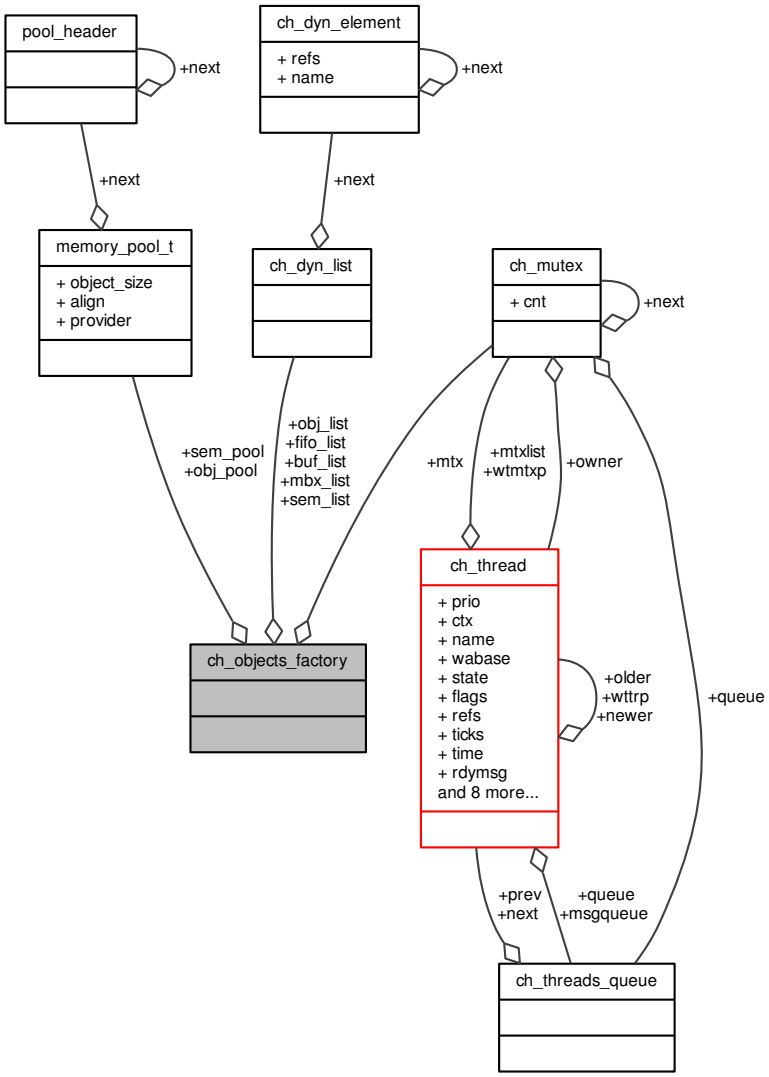
Mutex recursion counter.

9.9 ch_objects_factory Struct Reference

Type of the factory main object.

`#include <chfactory.h>`

Collaboration diagram for ch_objects_factory:



Data Fields

- mutex_t mtx

  Factory access mutex or semaphore.
- dyn_list_t obj_list

  List of the registered objects.
- memory_pool_t obj_pool

  Pool of the available registered objects.
- dyn_list_t buf_list

  List of the allocated buffer objects.
- dyn_list_t sem_list

  List of the allocated semaphores.
- memory_pool_t sem_pool

Pool of the available semaphores.

- dyn_list_t mbx_list

  List of the allocated buffer objects.

- dyn_list_t fifo_list

  List of the allocated "objects FIFO" objects.

9.9.1 Detailed Description

Type of the factory main object.

9.9.2 Field Documentation

9.9.2.1 mutex_t ch_objects_factory::mtx

Factory access mutex or semaphore.

9.9.2.2 dyn_list_t ch_objects_factory::obj_list

List of the registered objects.

9.9.2.3 memory_pool_t ch_objects_factory::obj_pool

Pool of the available registered objects.

9.9.2.4 dyn_list_t ch_objects_factory::buf_list

List of the allocated buffer objects.

9.9.2.5 dyn_list_t ch_objects_factory::sem_list

List of the allocated semaphores.

9.9.2.6 memory_pool_t ch_objects_factory::sem_pool

Pool of the available semaphores.

9.9.2.7 dyn_list_t ch_objects_factory::mbx_list

List of the allocated buffer objects.

9.9.2.8 dyn_list_t ch_objects_factory::fifo_list

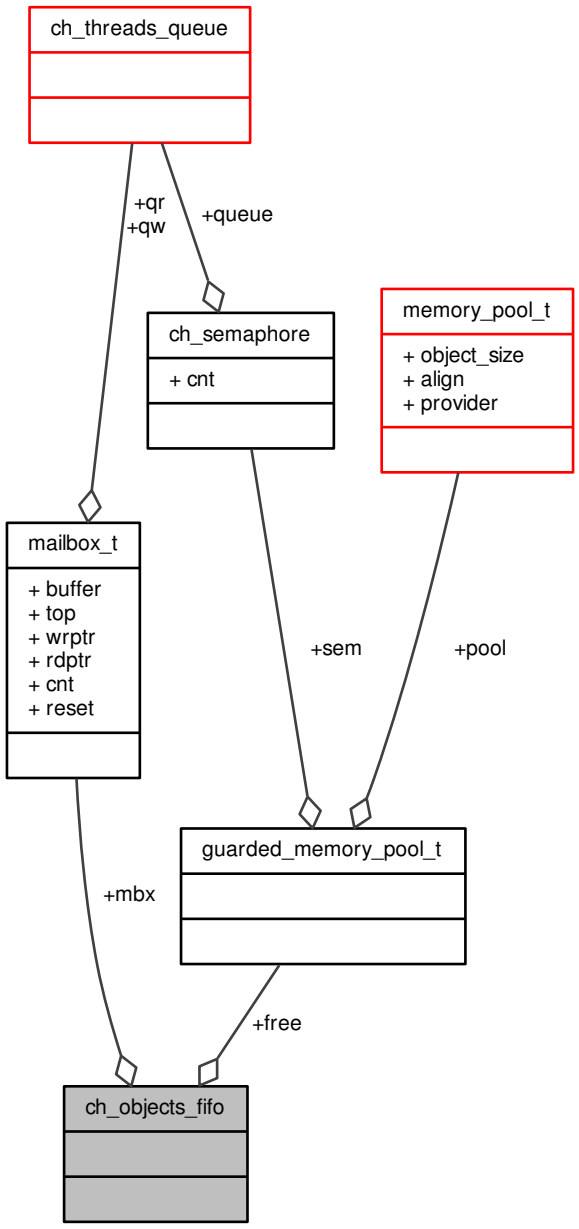
List of the allocated "objects FIFO" objects.

9.10 ch_objects_fifo Struct Reference

Type of an objects FIFO.

`#include <chfifo.h>`

Collaboration diagram for ch_objects_fifo:



Data Fields

- guarded_memory_pool_t free

  Pool of the free objects.

- mailbox_t mbx

  Mailbox of the sent objects.

9.10.1 Detailed Description

Type of an objects FIFO.

9.10.2 Field Documentation

9.10.2.1 guarded_memory_pool_t ch_objects_fifo::free

Pool of the free objects.

9.10.2.2 mailbox_t ch_objects_fifo::mbx

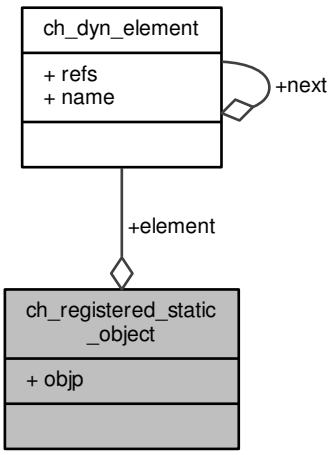
Mailbox of the sent objects.

9.11 ch_registered_static_object Struct Reference

Type of a registered object.

`#include <chfactory.h>`

Collaboration diagram for ch_registered_static_object:



Data Fields

- dyn_element_t element

  List element of the registered object.

- void * objp

  Pointer to the object.

9.11.1 Detailed Description

Type of a registered object.

9.11.2 Field Documentation

9.11.2.1 dyn_element_t ch_registered_static_object::element

List element of the registered object.

9.11.2.2 void* ch_registered_static_object::objp

Pointer to the object.

Note

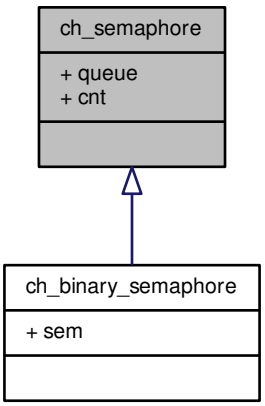
The type of the object is not stored in anyway.

9.12 ch_semaphore Struct Reference

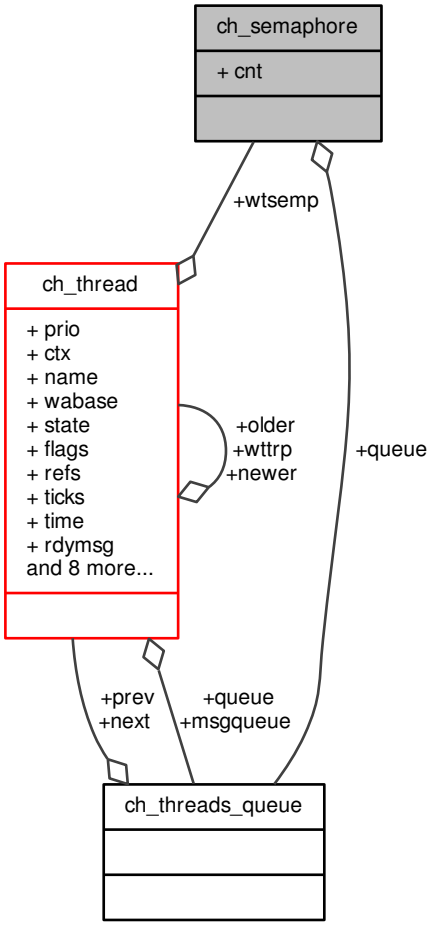
Semaphore structure.

`#include <chsem.h>`

Inheritance diagram for ch_semaphore:



Collaboration diagram for ch_semaphore:



Data Fields

- threads_queue_t queue

  Queue of the threads sleeping on this semaphore.
- cnt_t cnt

  The semaphore counter.

9.12.1 Detailed Description

Semaphore structure.

9.12.2 Field Documentation

9.12.2.1 threads_queue_t ch_semaphore::queue

Queue of the threads sleeping on this semaphore.

9.12.2.2 cnt_t ch_semaphore::cnt

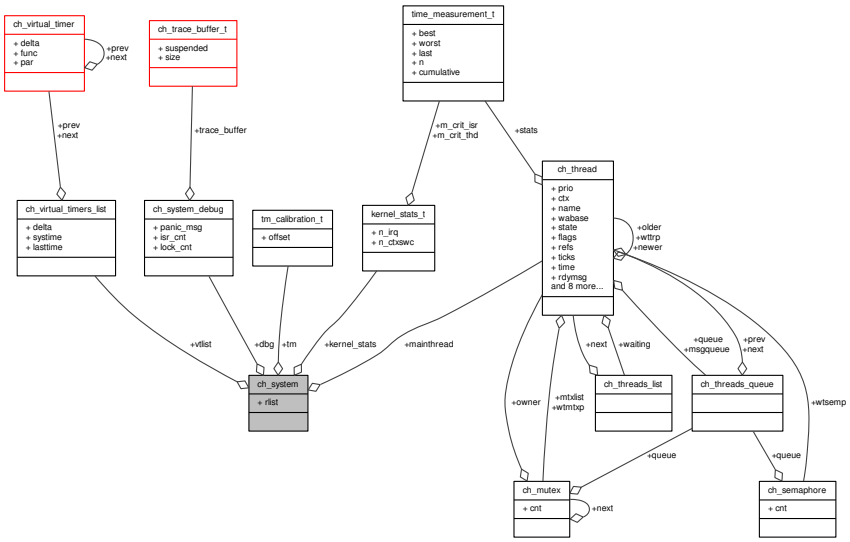
The semaphore counter.

9.13 ch_system Struct Reference

System data structure.

`#include <chschd.h>`

Collaboration diagram for ch_system:



Data Fields

- ready_list_t rlist

  Ready list header.
- virtual_timers_list_t vtlist

  Virtual timers delta list header.
- system_debug_t dbg

  System debug.
- thread_t mainthread

  Main thread descriptor.
- tm_calibration_t tm

  Time measurement calibration data.
- kernel_stats_t kernel_stats

  Global kernel statistics.

9.13.1 Detailed Description

System data structure.

Note

This structure contain all the data areas used by the OS except stacks.

9.13.2 Field Documentation

9.13.2.1 ready_list_t ch_system::rlist

Ready list header.

9.13.2.2 virtual_timers_list_t ch_system::vtlist

Virtual timers delta list header.

9.13.2.3 system_debug_t ch_system::dbg

System debug.

9.13.2.4 thread_t ch_system::mainthread

Main thread descriptor.

9.13.2.5 tm_calibration_t ch_system::tm

Time measurement calibration data.

9.13.2.6 kernel_stats_t ch_system::kernel_stats

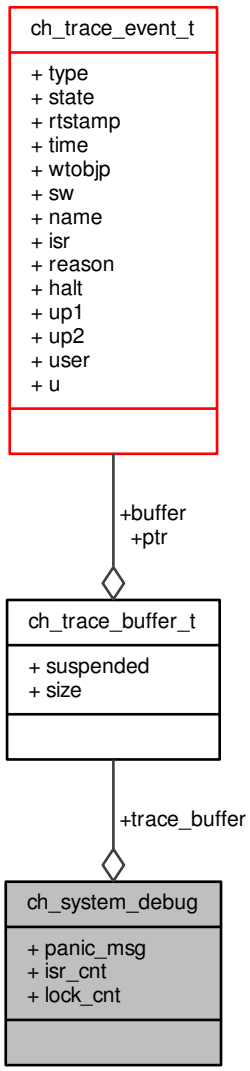
Global kernel statistics.

9.14 ch_system_debug Struct Reference

System debug data structure.

`#include <chschd.h>`

Collaboration diagram for ch_system_debug:



Data Fields

- const char *volatile panic_msg

  Pointer to the panic message.

- cnt_t isr_cnt

  ISR nesting level.

- cnt_t lock_cnt

  Lock nesting level.

- ch_trace_buffer_t trace_buffer

  Public trace buffer.

9.14.1 Detailed Description

System debug data structure.

9.14.2 Field Documentation

9.14.2.1 const char* volatile ch_system_debug::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.

Note

Accesses to this pointer must never be optimized out so the field itself is declared volatile.

9.14.2.2 cnt_t ch_system_debug::isr_cnt

ISR nesting level.

9.14.2.3 cnt_t ch_system_debug::lock_cnt

Lock nesting level.

9.14.2.4 ch_trace_buffer_t ch_system_debug::trace_buffer

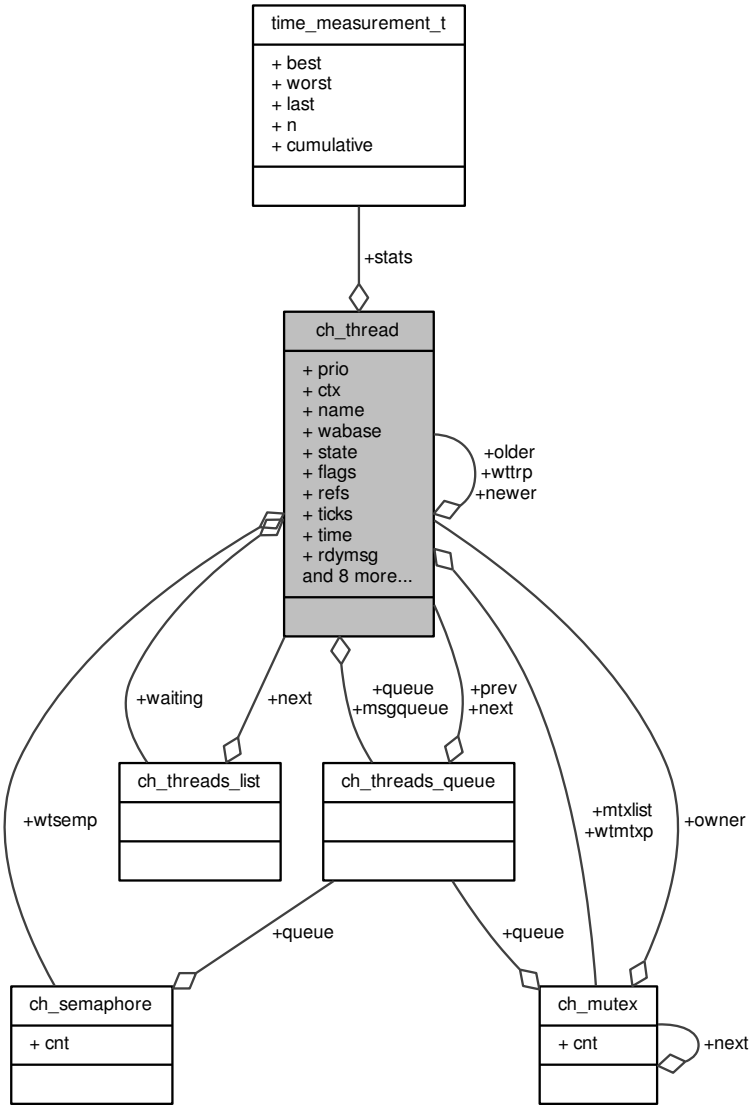
Public trace buffer.

9.15 ch_thread Struct Reference

Structure representing a thread.

`#include <chschd.h>`

Collaboration diagram for ch_thread:



Data Fields

- threads_queue_t queue

  Threads queue header.
- tprio_t prio

  Thread priority.
- struct port_context ctx

  Processor context.
- thread_t * newer

  Newer registry element.
- thread_t * older

  Older registry element.

- const char * name

  Thread name or NULL.
- stkalign_t * wabase

  Working area base address.
- tstate_t state

  Current thread state.
- tmode_t flags

  Various thread flags.
- trefs_t refs

  References to this thread.
- tslices_t ticks

  Number of ticks remaining to this thread.
- volatile systime_t time

  Thread consumed time in ticks.
- union {
  msg_t rdymsg
  Thread wakeup code.
  msg_t exitcode
  Thread exit code.
  void * wtobjp
  Pointer to a generic "wait" object.
  thread_reference_t * wttrp
  Pointer to a generic thread reference object.
  msg_t sentmsg
  Thread sent message.
  struct ch_semaphore * wtsemp
  Pointer to a generic semaphore object.
  struct ch_mutex * wtmtxp
  Pointer to a generic mutex object.
  eventmask_t ewmask
  Enabled events mask.
  } u

  State-specific fields.
- threads_list_t waiting

  Termination waiting list.
- threads_queue_t msgqueue

  Messages queue.
- eventmask_t epending

  Pending events mask.
- struct ch_mutex * mtxlist

  List of the mutexes owned by this thread.
- tprio_t realprio

  Thread's own, non-inherited, priority.
- void * mpool

  Memory Pool where the thread workspace is returned.
- time_measurement_t stats

  Thread statistics.

9.15.1 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 this structure.

9.15.2 Field Documentation

9.15.2.1 threads_queue_t ch_thread::queue

Threads queue header.

9.15.2.2 tprio_t ch_thread::prio

Thread priority.

9.15.2.3 struct port_context ch_thread::ctx

Processor context.

9.15.2.4 thread_t* ch_thread::newer

Newer registry element.

9.15.2.5 thread_t* ch_thread::older

Older registry element.

9.15.2.6 const char* ch_thread::name

Thread name or `NULL`.

9.15.2.7 stkalign_t* ch_thread::wabase

Working area base address.

Note

This pointer is used for stack overflow checks and for dynamic threading.

9.15.2.8 tstate_t ch_thread::state

Current thread state.

9.15.2.9 tmode_t ch_thread::flags

Various thread flags.

9.15.2.10 trefs_t ch_thread::refs

References to this thread.

9.15.2.11 tslices_t ch_thread::ticks

Number of ticks remaining to this thread.

9.15.2.12 volatile systime_t ch_thread::time

Thread consumed time in ticks.

Note

This field can overflow.

9.15.2.13 msg_t ch_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.

9.15.2.14 msg_t ch_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.

9.15.2.15 void* ch_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.

9.15.2.16 thread_reference_t* ch_thread::wttrp

Pointer to a generic thread reference object.

Note

This field is used to get a pointer to a synchronization object and is valid when the thread is in `CH_STATE_SUSPENDED` state.

9.15.2.17 msg_t ch_thread::sentmsg

Thread sent message.

9.15.2.18 struct ch_semaphore* ch_thread::wtsemp

Pointer to a generic semaphore object.

Note

This field is used to get a pointer to a synchronization object and is valid when the thread is in CH_STATE_WTSEM state.

9.15.2.19 struct ch_mutex* ch_thread::wtmtxp

Pointer to a generic mutex object.

Note

This field is used to get a pointer to a synchronization object and is valid when the thread is in CH_STATE_WTMTX state.

9.15.2.20 eventmask_t ch_thread::ewmask

Enabled events mask.

Note

This field is only valid while the thread is in the CH_STATE_WTOREVT or CH_STATE_WTANDEVT states.

9.15.2.21 union { ... } ch_thread::u

State-specific fields.

Note

All the fields declared in this union are only valid in the specified state or condition and are thus volatile.

9.15.2.22 threads_list_t ch_thread::waiting

Termination waiting list.

9.15.2.23 threads_queue_t ch_thread::msgqueue

Messages queue.

9.15.2.24 eventmask_t ch_thread::epending

Pending events mask.

9.15.2.25 struct ch_mutex* ch_thread::mtxlist

List of the mutexes owned by this thread.

Note

The list is terminated by a `NULL` in this field.

9.15.2.26 tprio_t ch_thread::realprio

Thread's own, non-inherited, priority.

9.15.2.27 void* ch_thread::mpool

Memory Pool where the thread workspace is returned.

9.15.2.28 time_measurement_t ch_thread::stats

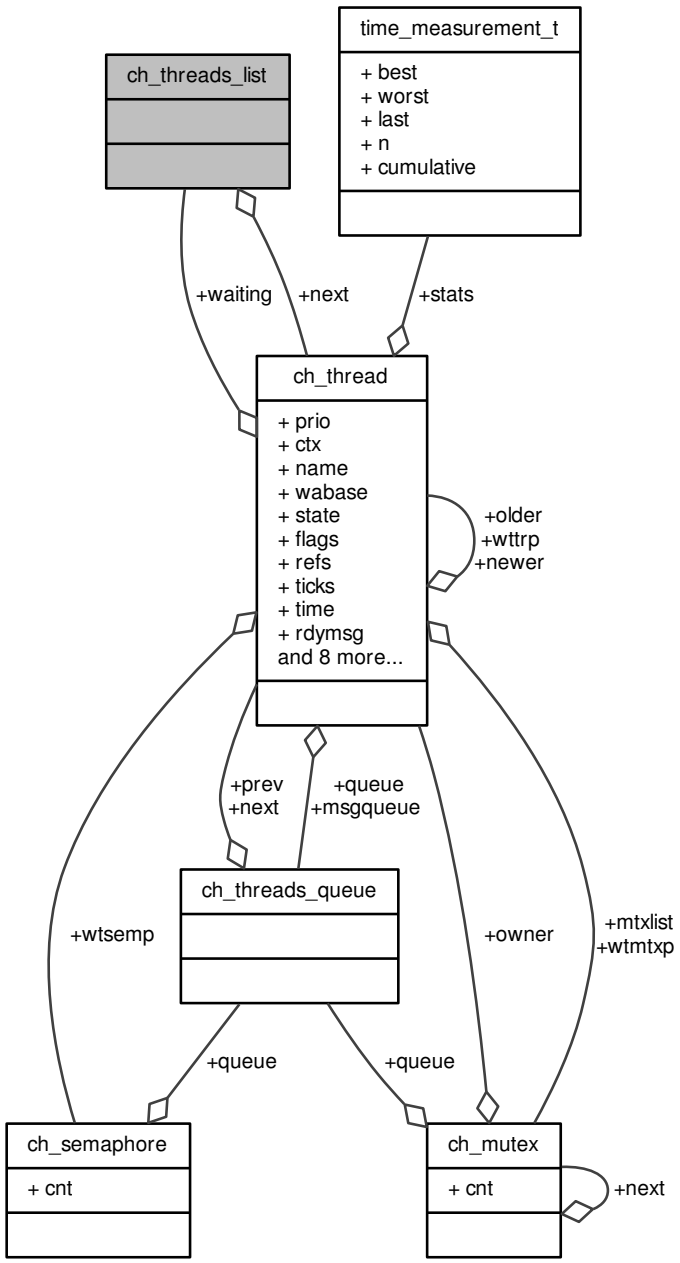
Thread statistics.

9.16 ch_threads_list Struct Reference

Generic threads single link list, it works like a stack.

`#include <chschd.h>`

Collaboration diagram for ch_threads_list:



Data Fields

- thread_t * next

  Next in the list/queue.

9.16.1 Detailed Description

Generic threads single link list, it works like a stack.

9.16.2 Field Documentation

9.16.2.1 thread_t* ch_threads_list::next

Next in the list/queue.

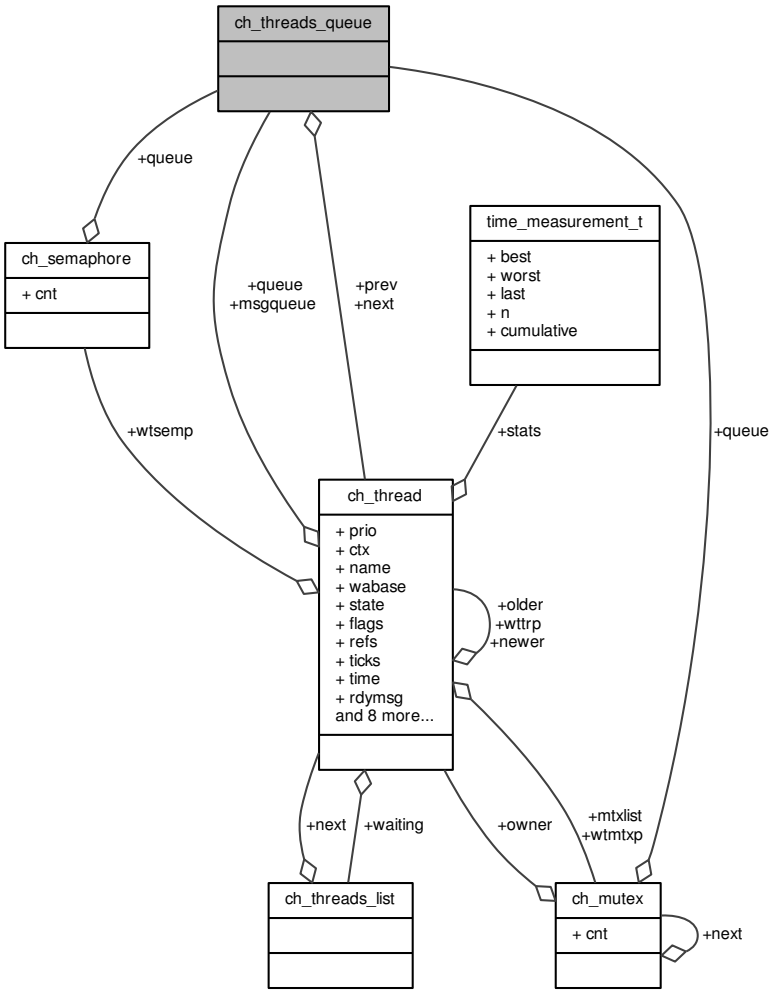
9.17 ch_threads_queue Struct Reference

Generic threads bidirectional linked list header and element.

`#include <chschd.h>`

Inherited by ch_ready_list.

Collaboration diagram for ch_threads_queue:



Data Fields

- thread_t * next

  Next in the list/queue.

- thread_t * prev

  Previous in the queue.

9.17.1 Detailed Description

Generic threads bidirectional linked list header and element.

9.17.2 Field Documentation

9.17.2.1 thread_t* ch_threads_queue::next

Next in the list/queue.

9.17.2.2 thread_t* ch_threads_queue::prev

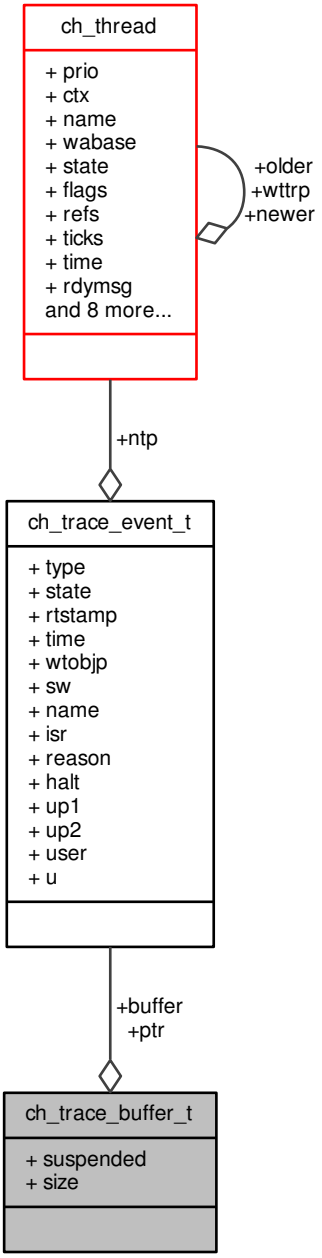
Previous in the queue.

9.18 ch_trace_buffer_t Struct Reference

Trace buffer header.

`#include <chtrace.h>`

Collaboration diagram for ch_trace_buffer_t:



Data Fields

- uint16_t suspended

  Suspended trace sources mask.
- uint16_t size

  Trace buffer size (entries).
- ch_trace_event_t * ptr

Pointer to the buffer front.

- ch_trace_event_t buffer [CH_DBG_TRACE_BUFFER_SIZE]

  Ring buffer.

9.18.1 Detailed Description

Trace buffer header.

9.18.2 Field Documentation

9.18.2.1 uint16_t ch_trace_buffer_t::suspended

Suspended trace sources mask.

9.18.2.2 uint16_t ch_trace_buffer_t::size

Trace buffer size (entries).

9.18.2.3 ch_trace_event_t* ch_trace_buffer_t::ptr

Pointer to the buffer front.

9.18.2.4 ch_trace_event_t ch_trace_buffer_t::buffer[CH_DBG_TRACE_BUFFER_SIZE]

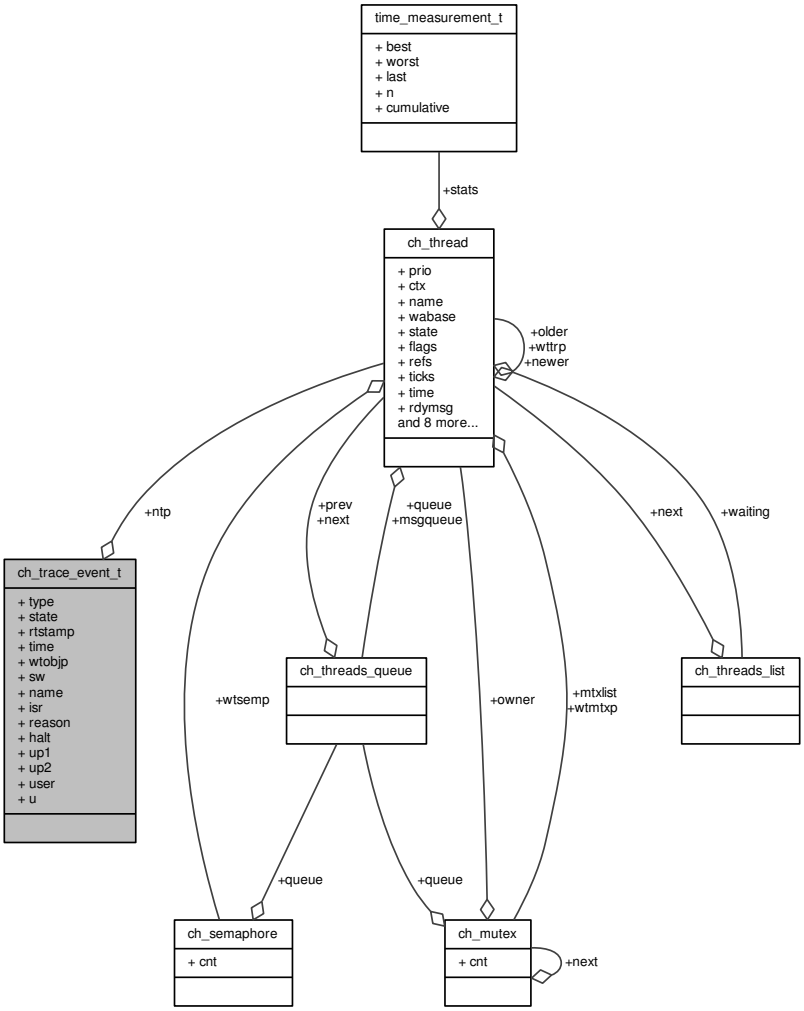
Ring buffer.

9.19 ch_trace_event_t Struct Reference

Trace buffer record.

`#include <chtrace.h>`

Collaboration diagram for ch_trace_event_t:



Data Fields

- uint32_t type:3

  Record type.
- uint32_t state:5

  Switched out thread state.
- uint32_t rtstamp:24

  Accurate time stamp.
- systime_t time

  System time stamp of the switch event.
- thread_t * ntp

  Switched in thread.
- void * wtobjp

  Object where going to sleep.

- struct {
  thread_t * ntp
  Switched in thread.
  void * wtobjp
  Object where going to sleep.
  } sw

  Structure representing a context switch.
- const char * name

  ISR function name taken using ***func****.*
- struct {
  const char * name
  ISR function name taken using ***func****.*
  } isr

  Structure representing an ISR enter.
- const char * reason

  Halt error string.
- struct {
  const char * reason
  Halt error string.
  } halt

  Structure representing an halt.
- void * up1

  Trace user parameter 1.
- void * up2

  Trace user parameter 2.
- struct {
  void * up1
  Trace user parameter 1.
  void * up2
  Trace user parameter 2.
  } user

  User trace structure.

9.19.1 Detailed Description

Trace buffer record.

9.19.2 Field Documentation

9.19.2.1 uint32_t ch_trace_event_t::type

Record type.

9.19.2.2 uint32_t ch_trace_event_t::state

Switched out thread state.

9.19.2.3 uint32_t ch_trace_event_t::rtstamp

Accurate time stamp.

Note

This field only available if the post supports `PORT_SUPPORTS_RT` else it is set to zero.

9.19.2.4 systime_t ch_trace_event_t::time

System time stamp of the switch event.

9.19.2.5 thread_t* ch_trace_event_t::ntp

Switched in thread.

9.19.2.6 void* ch_trace_event_t::wtobjp

Object where going to sleep.

9.19.2.7 struct { ... } ch_trace_event_t::sw

Structure representing a context switch.

9.19.2.8 const char* ch_trace_event_t::name

ISR function name taken using **`func`**.

9.19.2.9 struct { ... } ch_trace_event_t::isr

Structure representing an ISR enter.

9.19.2.10 const char* ch_trace_event_t::reason

Halt error string.

9.19.2.11 struct { ... } ch_trace_event_t::halt

Structure representing an halt.

9.19.2.12 void* ch_trace_event_t::up1

Trace user parameter 1.

9.19.2.13 void* ch_trace_event_t::up2

Trace user parameter 2.

9.19.2.14 struct { ... } ch_trace_event_t::user

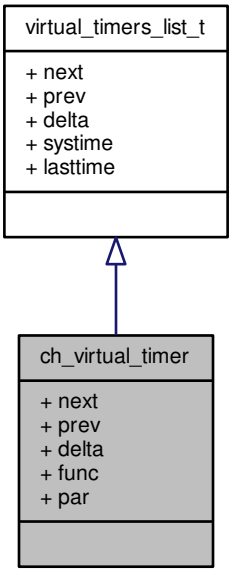
User trace structure.

9.20 ch_virtual_timer Struct Reference

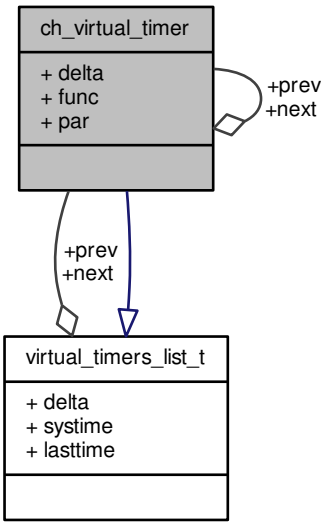
Virtual Timer descriptor structure.

`#include <chschd.h>`

Inheritance diagram for ch_virtual_timer:



Collaboration diagram for ch_virtual_timer:



Data Fields

- virtual_timer_t * next

  Next timer in the list.
- virtual_timer_t * prev

  Previous timer in the list.
- sysinterval_t delta

  Time delta before timeout.
- vtfunc_t func

  Timer callback function pointer.
- void * par

  Timer callback function parameter.

9.20.1 Detailed Description

Virtual Timer descriptor structure.

9.20.2 Field Documentation

9.20.2.1 virtual_timer_t* ch_virtual_timer::next

Next timer in the list.

9.20.2.2 virtual_timer_t* ch_virtual_timer::prev

Previous timer in the list.

9.20.2.3 sysinterval_t ch_virtual_timer::delta

Time delta before timeout.

9.20.2.4 vtfunc_t ch_virtual_timer::func

Timer callback function pointer.

9.20.2.5 void* ch_virtual_timer::par

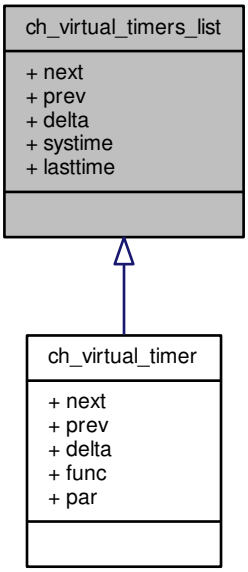
Timer callback function parameter.

9.21 ch_virtual_timers_list Struct Reference

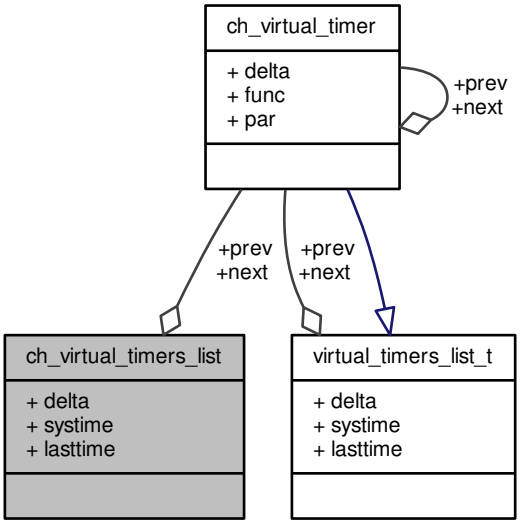
Virtual timers list header.

`#include <chschd.h>`

Inheritance diagram for ch_virtual_timers_list:



Collaboration diagram for ch_virtual_timers_list:



Data Fields

- virtual_timer_t * next

  Next timer in the delta list.
- virtual_timer_t * prev

  Last timer in the delta list.
- sysinterval_t delta

  Must be initialized to -1.
- volatile systime_t systime

  System Time counter.
- systime_t lasttime

  System time of the last tick event.

9.21.1 Detailed Description

Virtual timers list header.

Note

The timers 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.

9.21.2 Field Documentation

9.21.2.1 virtual_timer_t* ch_virtual_timers_list::next

Next timer in the delta list.

9.21.2.2 virtual_timer_t* ch_virtual_timers_list::prev

Last timer in the delta list.

9.21.2.3 sysinterval_t ch_virtual_timers_list::delta

Must be initialized to -1.

9.21.2.4 volatile systime_t ch_virtual_timers_list::systime

System Time counter.

9.21.2.5 systime_t ch_virtual_timers_list::lasttime

System time of the last tick event.

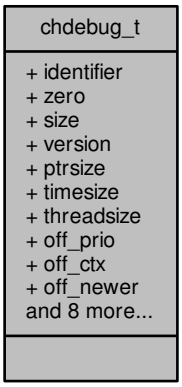
System time of the last tick event.

9.22 chdebug_t Struct Reference

ChibiOS/RT memory signature record.

`#include <chregistry.h>`

Collaboration diagram for chdebug_t:



Data Fields

- char identifier [4]

  Always set to "main".
- uint8_t zero

  Must be zero.
- uint8_t size

  Size of this structure.

- uint16_t version

  Encoded ChibiOS/RT version.
- uint8_t ptrsize

  Size of a pointer.
- uint8_t timesize

  Size of a `systime_t`*.*
- uint8_t threadsize

  Size of a `thread_t`*.*
- uint8_t off_prio

  Offset of `prio` *field.*
- uint8_t off_ctx

  Offset of `ctx` *field.*
- uint8_t off_newer

  Offset of `newer` *field.*
- uint8_t off_older

  Offset of `older` *field.*
- uint8_t off_name

  Offset of `name` *field.*
- uint8_t off_stklimit

  Offset of `stklimit` *field.*
- uint8_t off_state

  Offset of `state` *field.*
- uint8_t off_flags

  Offset of `flags` *field.*
- uint8_t off_refs

  Offset of `refs` *field.*
- uint8_t off_preempt

  Offset of `preempt` *field.*
- uint8_t off_time

  Offset of `time` *field.*

9.22.1 Detailed Description

ChibiOS/RT memory signature record.

9.22.2 Field Documentation

9.22.2.1 char chdebug_t::identifier[4]

Always set to "main".

9.22.2.2 uint8_t chdebug_t::zero

Must be zero.

9.22.2.3 uint8_t chdebug_t::size

Size of this structure.

9.22.2.4 uint16_t chdebug_t::version

Encoded ChibiOS/RT version.

9.22.2.5 uint8_t chdebug_t::ptrsize

Size of a pointer.

9.22.2.6 uint8_t chdebug_t::timesize

Size of a `systime_t`.

9.22.2.7 uint8_t chdebug_t::threadsize

Size of a `thread_t`.

9.22.2.8 uint8_t chdebug_t::off_prio

Offset of `prio` field.

9.22.2.9 uint8_t chdebug_t::off_ctx

Offset of `ctx` field.

9.22.2.10 uint8_t chdebug_t::off_newer

Offset of `newer` field.

9.22.2.11 uint8_t chdebug_t::off_older

Offset of `older` field.

9.22.2.12 uint8_t chdebug_t::off_name

Offset of `name` field.

9.22.2.13 uint8_t chdebug_t::off_stklimit

Offset of `stklimit` field.

9.22.2.14 uint8_t chdebug_t::off_state

Offset of `state` field.

9.22.2.15 uint8_t chdebug_t::off_flags

Offset of `flags` field.

9.22.2.16 uint8_t chdebug_t::off_refs

Offset of `refs` field.

9.22.2.17 uint8_t chdebug_t::off_preempt

Offset of `preempt` field.

9.22.2.18 uint8_t chdebug_t::off_time

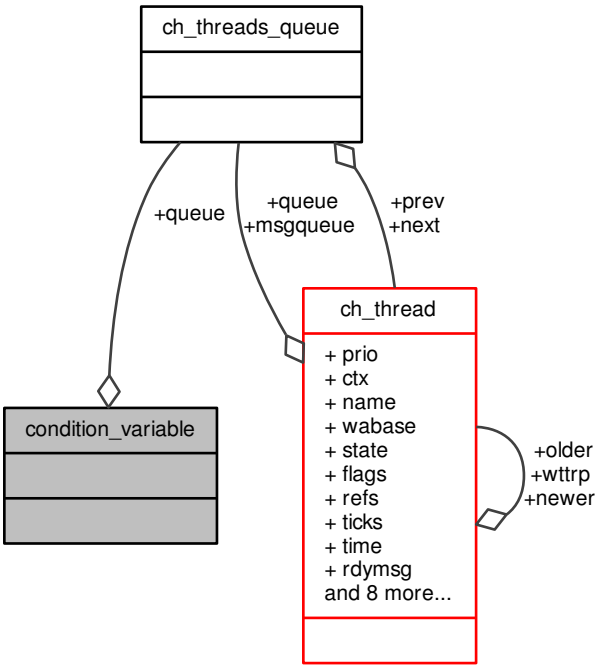
Offset of `time` field.

9.23 condition_variable Struct Reference

condition_variable_t structure.

`#include <chcond.h>`

Collaboration diagram for condition_variable:



Data Fields

- threads_queue_t queue

  Condition variable threads queue.

9.23.1 Detailed Description

condition_variable_t structure.

9.23.2 Field Documentation

9.23.2.1 threads_queue_t condition_variable::queue

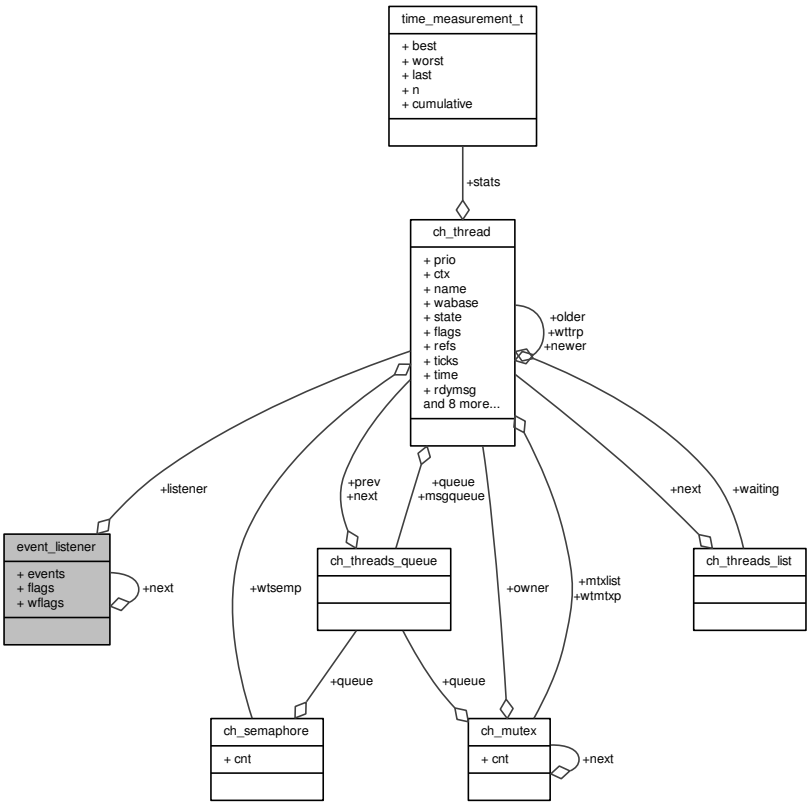
Condition variable threads queue.

9.24 event_listener Struct Reference

Event Listener structure.

`#include <chevents.h>`

Collaboration diagram for event_listener:



Data Fields

- event_listener_t * next

  Next Event Listener registered on the event source.

- thread_t * listener

  Thread interested in the event source.

- eventmask_t events

  Events to be set in the listening thread.

- eventflags_t flags

  Flags added to the listener by the event source.

- eventflags_t wflags

  Flags that this listener interested in.

9.24.1 Detailed Description

Event Listener structure.

9.24.2 Field Documentation

9.24.2.1 event_listener_t* event_listener::next

Next Event Listener registered on the event source.

9.24.2.2 thread_t* event_listener::listener

Thread interested in the event source.

9.24.2.3 eventmask_t event_listener::events

Events to be set in the listening thread.

9.24.2.4 eventflags_t event_listener::flags

Flags added to the listener by the event source.

9.24.2.5 eventflags_t event_listener::wflags

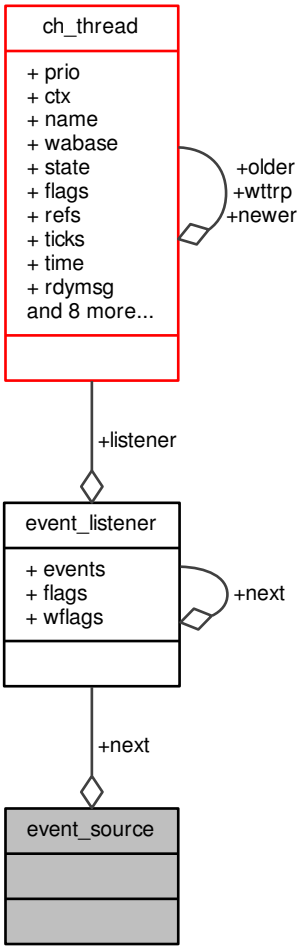
Flags that this listener interested in.

9.25 event_source Struct Reference

Event Source structure.

`#include <chevents.h>`

Collaboration diagram for event_source:



Data Fields

- event_listener_t * next

  First Event Listener registered on the Event Source.

9.25.1 Detailed Description

Event Source structure.

9.25.2 Field Documentation

9.25.2.1 event_listener_t* event_source::next

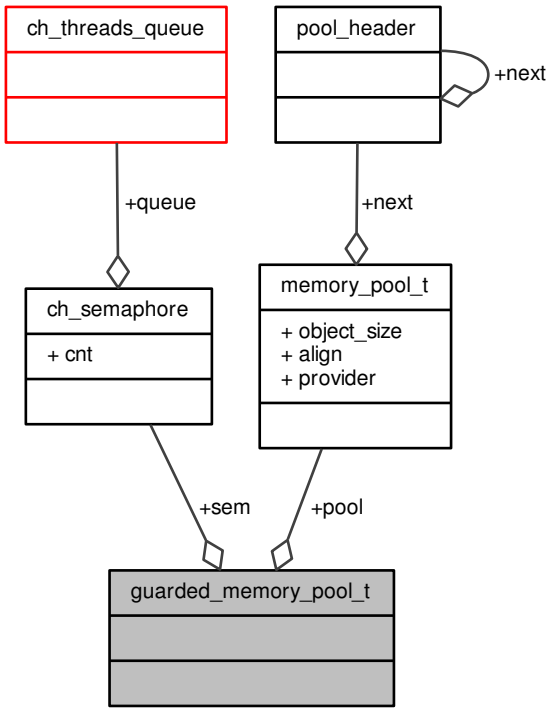
First Event Listener registered on the Event Source.

9.26 guarded_memory_pool_t Struct Reference

Guarded memory pool descriptor.

`#include <chmempools.h>`

Collaboration diagram for guarded_memory_pool_t:



Data Fields

- semaphore_t sem

  Counter semaphore guarding the memory pool.
- memory_pool_t pool

  The memory pool itself.

9.26.1 Detailed Description

Guarded memory pool descriptor.

9.26.2 Field Documentation

9.26.2.1 semaphore_t guarded_memory_pool_t::sem

Counter semaphore guarding the memory pool.

9.26.2.2 memory_pool_t guarded_memory_pool_t::pool

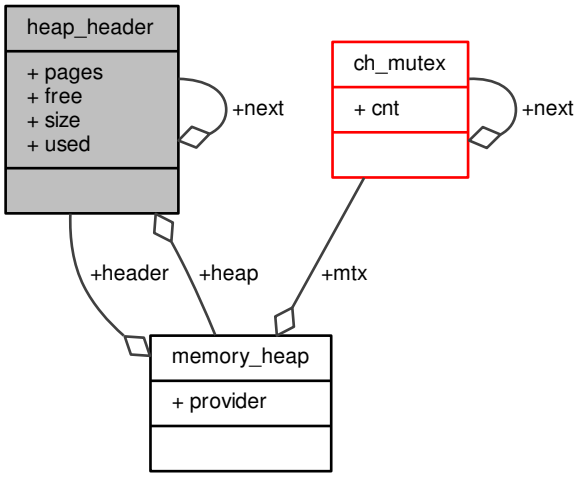
The memory pool itself.

9.27 heap_header Union Reference

Memory heap block header.

`#include <chheap.h>`

Collaboration diagram for heap_header:



9.27.1 Detailed Description

Memory heap block header.

9.27.2 Field Documentation

9.27.2.1 heap_header_t* heap_header::next

Next block in free list.

9.27.2.2 size_t heap_header::pages

Size of the area in pages.

9.27.2.3 memory_heap_t* heap_header::heap

Block owner heap.

9.27.2.4 size_t heap_header::size

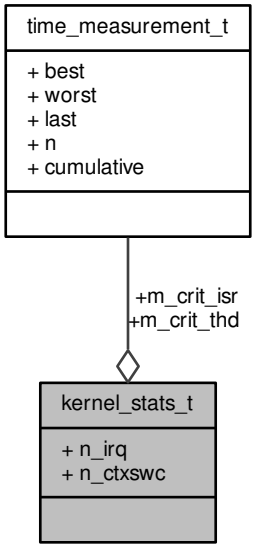
Size of the area in bytes.

9.28 kernel_stats_t Struct Reference

Type of a kernel statistics structure.

`#include <chstats.h>`

Collaboration diagram for kernel_stats_t:



Data Fields

- ucnt_t n_irq

  Number of IRQs.
- ucnt_t n_ctxswc

  Number of context switches.
- time_measurement_t m_crit_thd

  Measurement of threads critical zones duration.
- time_measurement_t m_crit_isr

  Measurement of ISRs critical zones duration.

9.28.1 Detailed Description

Type of a kernel statistics structure.

9.28.2 Field Documentation

9.28.2.1 ucnt_t kernel_stats_t::n_irq

Number of IRQs.

9.28.2.2 ucnt_t kernel_stats_t::n_ctxswc

Number of context switches.

9.28.2.3 time_measurement_t kernel_stats_t::m_crit_thd

Measurement of threads critical zones duration.

9.28.2.4 time_measurement_t kernel_stats_t::m_crit_isr

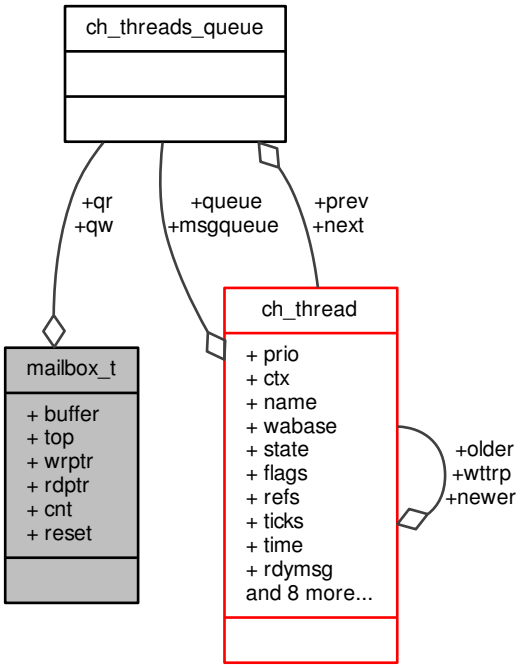
Measurement of ISRs critical zones duration.

9.29 mailbox_t Struct Reference

Structure representing a mailbox object.

`#include <chmboxes.h>`

Collaboration diagram for mailbox_t:



Data Fields

- msg_t * buffer

  Pointer to the mailbox buffer.
- msg_t * top

  Pointer to the location after the buffer.
- msg_t * wrptr

  Write pointer.
- msg_t * rdptr

  Read pointer.
- size_t cnt

  Messages in queue.
- bool reset

  True in reset state.
- threads_queue_t qw

  Queued writers.
- threads_queue_t qr

  Queued readers.

9.29.1 Detailed Description

Structure representing a mailbox object.

9.29.2 Field Documentation

9.29.2.1 msg_t* mailbox_t::buffer

Pointer to the mailbox buffer.

9.29.2.2 msg_t* mailbox_t::top

Pointer to the location after the buffer.

9.29.2.3 msg_t* mailbox_t::wrptr

Write pointer.

9.29.2.4 msg_t* mailbox_t::rdptr

Read pointer.

9.29.2.5 size_t mailbox_t::cnt

Messages in queue.

9.29.2.6 bool mailbox_t::reset

True in reset state.

9.29.2.7 threads_queue_t mailbox_t::qw

Queued writers.

9.29.2.8 threads_queue_t mailbox_t::qr

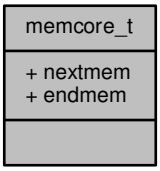
Queued readers.

9.30 memcore_t Struct Reference

Type of memory core object.

`#include <chmemcore.h>`

Collaboration diagram for memcore_t:



Data Fields

- uint8_t * nextmem

  Next free address.
- uint8_t * endmem

  Final address.

9.30.1 Detailed Description

Type of memory core object.

9.30.2 Field Documentation

9.30.2.1 uint8_t* memcore_t::nextmem

Next free address.

9.30.2.2 uint8_t* memcore_t::endmem

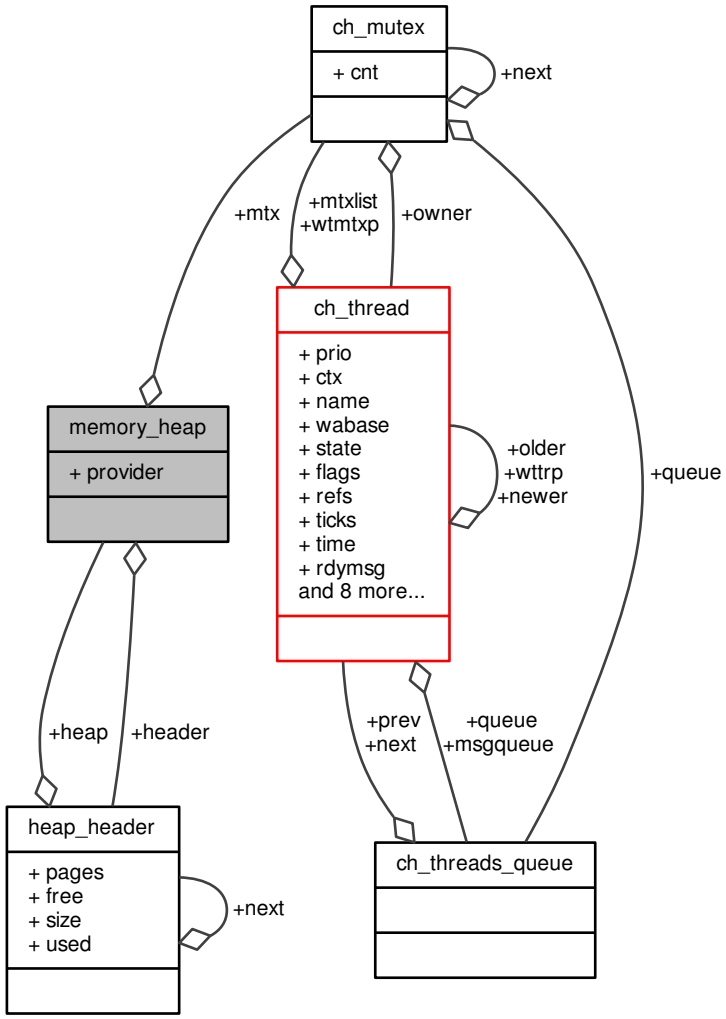
Final address.

9.31 memory_heap Struct Reference

Structure describing a memory heap.

`#include <chheap.h>`

Collaboration diagram for memory_heap:



Data Fields

- memgetfunc2_t provider

  Memory blocks provider for this heap.

- heap_header_t header

  Free blocks list header.

- mutex_t mtx

  Heap access mutex.

9.31.1 Detailed Description

Structure describing a memory heap.

9.31.2 Field Documentation

9.31.2.1 memgetfunc2_t memory_heap::provider

Memory blocks provider for this heap.

9.31.2.2 heap_header_t memory_heap::header

Free blocks list header.

9.31.2.3 mutex_t memory_heap::mtx

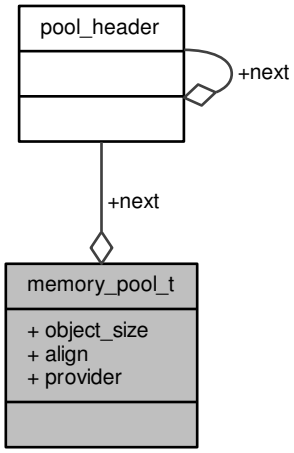
Heap access mutex.

9.32 memory_pool_t Struct Reference

Memory pool descriptor.

`#include <chmempools.h>`

Collaboration diagram for memory_pool_t:



Data Fields

- struct pool_header * next

  Pointer to the header.
- size_t object_size

Memory pool objects size.

- unsigned align

  Required alignment.

- memgetfunc_t provider

  Memory blocks provider for this pool.

9.32.1 Detailed Description

Memory pool descriptor.

9.32.2 Field Documentation

9.32.2.1 struct pool_header* memory_pool_t::next

Pointer to the header.

9.32.2.2 size_t memory_pool_t::object_size

Memory pool objects size.

9.32.2.3 unsigned memory_pool_t::align

Required alignment.

9.32.2.4 memgetfunc_t memory_pool_t::provider

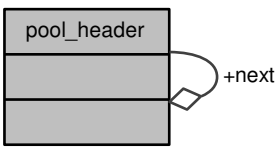
Memory blocks provider for this pool.

9.33 pool_header Struct Reference

Memory pool free object header.

`#include <chmempools.h>`

Collaboration diagram for pool_header:



Data Fields

- struct pool_header * next
  Pointer to the next pool header in the list.

9.33.1 Detailed Description

Memory pool free object header.

9.33.2 Field Documentation

9.33.2.1 struct pool_header* pool_header::next

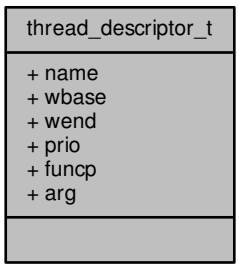
Pointer to the next pool header in the list.

9.34 thread_descriptor_t Struct Reference

Type of a thread descriptor.

`#include <chthreads.h>`

Collaboration diagram for thread_descriptor_t:



Data Fields

- const char * name
  Thread name.
- stkalign_t * wbase
  Pointer to the working area base.
- stkalign_t * wend
  End of the working area.
- tprio_t prio
  Thread priority.
- tfunc_t funcp
  Thread function pointer.
- void * arg
  Thread argument.

9.34.1 Detailed Description

Type of a thread descriptor.

9.34.2 Field Documentation

9.34.2.1 const char* thread_descriptor_t::name

Thread name.

9.34.2.2 stkalign_t* thread_descriptor_t::wbase

Pointer to the working area base.

9.34.2.3 stkalign_t* thread_descriptor_t::wend

End of the working area.

9.34.2.4 tprio_t thread_descriptor_t::prio

Thread priority.

9.34.2.5 tfunc_t thread_descriptor_t::funcp

Thread function pointer.

9.34.2.6 void* thread_descriptor_t::arg

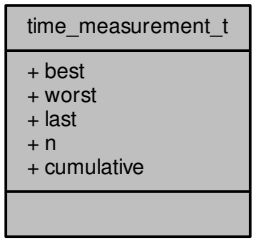
Thread argument.

9.35 time_measurement_t Struct Reference

Type of a Time Measurement object.

`#include <chtm.h>`

Collaboration diagram for time_measurement_t:



Data Fields

- rtcnt_t best

  Best measurement.
- rtcnt_t worst

  Worst measurement.
- rtcnt_t last

  Last measurement.
- ucnt_t n

  Number of measurements.
- rttime_t cumulative

  Cumulative measurement.

9.35.1 Detailed Description

Type of a Time Measurement object.

Note

The maximum measurable time period depends on the implementation of the realtime counter and its clock frequency.
The measurement is not 100% cycle-accurate, it can be in excess of few cycles depending on the compiler and target architecture.
Interrupts can affect measurement if the measurement is performed with interrupts enabled.

9.35.2 Field Documentation

9.35.2.1 rtcnt_t time_measurement_t::best

Best measurement.

9.35.2.2 rtcnt_t time_measurement_t::worst

Worst measurement.

9.35.2.3 rtcnt_t time_measurement_t::last

Last measurement.

9.35.2.4 ucnt_t time_measurement_t::n

Number of measurements.

9.35.2.5 rttime_t time_measurement_t::cumulative

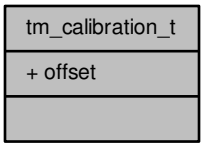
Cumulative measurement.

9.36 tm_calibration_t Struct Reference

Type of a time measurement calibration data.

`#include <chtm.h>`

Collaboration diagram for tm_calibration_t:



Data Fields

- rtcnt_t offset

  Measurement calibration value.

9.36.1 Detailed Description

Type of a time measurement calibration data.

9.36.2 Field Documentation

9.36.2.1 rtcnt_t tm_calibration_t::offset

Measurement calibration value.

Chapter 10

File Documentation

10.1 ch.h File Reference

ChibiOS/RT main include file.

```
#include "chconf.h"
#include "chchecks.h"
#include "chlicense.h"
#include "chrestrictions.h"
#include "chtypes.h"
#include "chsystypes.h"
#include "chdebug.h"
#include "chtime.h"
#include "chalign.h"
#include "chcore.h"
#include "chtrace.h"
#include "chtm.h"
#include "chstats.h"
#include "chschd.h"
#include "chsys.h"
#include "chvt.h"
#include "chthreads.h"
#include "chregistry.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 "chfifo.h"
#include "chfactory.h"
#include "chdynamic.h"
```

Macros

- #define _CHIBIOS_RT_

  ChibiOS/RT identification macro.

- #define CH_KERNEL_STABLE 1
  Stable release flag.

ChibiOS/RT version identification

- #define CH_KERNEL_VERSION "5.0.0"
  Kernel version string.
- #define CH_KERNEL_MAJOR 5
  Kernel version major number.
- #define CH_KERNEL_MINOR 0
  Kernel version minor number.
- #define CH_KERNEL_PATCH 0
  Kernel version patch number.

Constants for configuration options

- #define FALSE 0
  Generic 'false' preprocessor boolean constant.
- #define TRUE 1
  Generic 'true' preprocessor boolean constant.

Functions

- void chSysHalt (const char *reason)
  Halts the system.

10.1.1 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.

10.2 chalign.h File Reference

Memory alignment macros and structures.

Macros

Memory alignment support macros

- #define MEM_ALIGN_MASK(a) ((size_t)(a) - 1U)
  Alignment mask constant.
- #define MEM_ALIGN_PREV(p, a)
  Aligns to the previous aligned memory address.
- #define MEM_ALIGN_NEXT(p, a)
  Aligns to the next aligned memory address.
- #define MEM_IS_ALIGNED(p, a) (((size_t)(p) & MEM_ALIGN_MASK(a)) == 0U)
  Returns whatever a pointer or memory size is aligned.
- #define MEM_IS_VALID_ALIGNMENT(a) (((size_t)(a) != 0U) && (((size_t)(a) & ((size_t)(a) - 1U)) == 0U))
  Returns whatever a constant is a valid alignment.

10.2.1 Detailed Description

Memory alignment macros and structures.

10.3 chbsem.h File Reference

Binary semaphores structures and macros.

Data Structures

- struct ch_binary_semaphore

  Binary semaphore type.

Macros

- #define _BSEMAPHORE_DATA(name, taken) {_SEMAPHORE_DATA(name.sem, ((taken) ? 0 : 1))}

  Data part of a static semaphore initializer.
- #define BSEMAPHORE_DECL(name, taken) binary_semaphore_t name = _BSEMAPHORE_DATA(name, taken)

  Static semaphore initializer.

Typedefs

- typedef struct ch_binary_semaphore binary_semaphore_t

  Binary semaphore type.

Functions

- static void chBSemObjectInit (binary_semaphore_t *bsp, bool taken)

  Initializes a binary semaphore.
- static msg_t chBSemWait (binary_semaphore_t *bsp)

  Wait operation on the binary semaphore.
- static msg_t chBSemWaitS (binary_semaphore_t *bsp)

  Wait operation on the binary semaphore.
- static msg_t chBSemWaitTimeoutS (binary_semaphore_t *bsp, sysinterval_t timeout)

  Wait operation on the binary semaphore.
- static msg_t chBSemWaitTimeout (binary_semaphore_t *bsp, sysinterval_t timeout)

  Wait operation on the binary semaphore.
- static void chBSemResetI (binary_semaphore_t *bsp, bool taken)

  Reset operation on the binary semaphore.
- static void chBSemReset (binary_semaphore_t *bsp, bool taken)

  Reset operation on the binary semaphore.
- static void chBSemSignalI (binary_semaphore_t *bsp)

  Performs a signal operation on a binary semaphore.
- static void chBSemSignal (binary_semaphore_t *bsp)

  Performs a signal operation on a binary semaphore.
- static bool chBSemGetStateI (const binary_semaphore_t *bsp)

  Returns the binary semaphore current state.

10.3.1 Detailed Description

Binary semaphores structures and macros.

10.4 chchecks.h File Reference

Configuration file checks header.

10.4.1 Detailed Description

Configuration file checks header.

10.5 chcond.c File Reference

Condition Variables code.

```
#include "ch.h"
```

Functions

- void chCondObjectInit (condition_variable_t *cp)

  Initializes s `condition_variable_t` structure.
- void chCondSignal (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondSignalI (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondBroadcast (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- void chCondBroadcastI (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- msg_t chCondWait (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitS (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitTimeout (condition_variable_t *cp, sysinterval_t timeout)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitTimeoutS (condition_variable_t *cp, sysinterval_t timeout)

  Waits on the condition variable releasing the mutex lock.

10.5.1 Detailed Description

Condition Variables code.

10.6 chcond.h File Reference

Condition Variables macros and structures.

Data Structures

- struct condition_variable

  condition_variable_t structure.

Macros

- #define _CONDVAR_DATA(name) {_THREADS_QUEUE_DATA(name.queue)}

  Data part of a static condition variable initializer.
- #define CONDVAR_DECL(name) condition_variable_t name = _CONDVAR_DATA(name)

  Static condition variable initializer.

Typedefs

- typedef struct condition_variable condition_variable_t

  condition_variable_t structure.

Functions

- void chCondObjectInit (condition_variable_t *cp)

  Initializes s `condition_variable_t` *structure.*
- void chCondSignal (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondSignalI (condition_variable_t *cp)

  Signals one thread that is waiting on the condition variable.
- void chCondBroadcast (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- void chCondBroadcastI (condition_variable_t *cp)

  Signals all threads that are waiting on the condition variable.
- msg_t chCondWait (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitS (condition_variable_t *cp)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitTimeout (condition_variable_t *cp, sysinterval_t timeout)

  Waits on the condition variable releasing the mutex lock.
- msg_t chCondWaitTimeoutS (condition_variable_t *cp, sysinterval_t timeout)

  Waits on the condition variable releasing the mutex lock.

10.6.1 Detailed Description

Condition Variables macros and structures.

10.7 chconf.h File Reference

Configuration file template.

Macros

System timers settings

- #define CH_CFG_ST_RESOLUTION 32

  System time counter resolution.
- #define CH_CFG_ST_FREQUENCY 10000

  System tick frequency.
- #define CH_CFG_INTERVALS_SIZE 32

Time intervals data size.

- #define CH_CFG_TIME_TYPES_SIZE 32
  Time types data size.
- #define CH_CFG_ST_TIMEDELTA 2
  Time delta constant for the tick-less mode.

Kernel parameters and options

- #define CH_CFG_TIME_QUANTUM 0
  Round robin interval.
- #define CH_CFG_MEMCORE_SIZE 0
  Managed RAM size.
- #define CH_CFG_NO_IDLE_THREAD FALSE
  Idle thread automatic spawn suppression.

Performance options

- #define CH_CFG_OPTIMIZE_SPEED TRUE
  OS optimization.

Subsystem options

- #define CH_CFG_USE_TM TRUE
  Time Measurement APIs.
- #define CH_CFG_USE_REGISTRY TRUE
  Threads registry APIs.
- #define CH_CFG_USE_WAITEXIT TRUE
  Threads synchronization APIs.
- #define CH_CFG_USE_SEMAPHORES TRUE
  Semaphores APIs.
- #define CH_CFG_USE_SEMAPHORES_PRIORITY FALSE
  Semaphores queuing mode.
- #define CH_CFG_USE_MUTEXES TRUE
  Mutexes APIs.
- #define CH_CFG_USE_MUTEXES_RECURSIVE FALSE
  Enables recursive behavior on mutexes.
- #define CH_CFG_USE_CONDVARS TRUE
  Conditional Variables APIs.
- #define CH_CFG_USE_CONDVARS_TIMEOUT TRUE
  Conditional Variables APIs with timeout.
- #define CH_CFG_USE_EVENTS TRUE
  Events Flags APIs.
- #define CH_CFG_USE_EVENTS_TIMEOUT TRUE
  Events Flags APIs with timeout.
- #define CH_CFG_USE_MESSAGES TRUE
  Synchronous Messages APIs.
- #define CH_CFG_USE_MESSAGES_PRIORITY FALSE
  Synchronous Messages queuing mode.
- #define CH_CFG_USE_MAILBOXES TRUE
  Mailboxes APIs.
- #define CH_CFG_USE_MEMCORE TRUE
  Core Memory Manager APIs.
- #define CH_CFG_USE_HEAP TRUE
  Heap Allocator APIs.
- #define CH_CFG_USE_MEMPOOLS TRUE
  Memory Pools Allocator APIs.
- #define CH_CFG_USE_OBJ_FIFOS TRUE
  Objects FIFOs APIs.
- #define CH_CFG_USE_DYNAMIC TRUE

Dynamic Threads APIs.

Objects factory options

- #define CH_CFG_USE_FACTORY TRUE
  Objects Factory APIs.
- #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8
  Maximum length for object names.
- #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE
  Enables the registry of generic objects.
- #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE
  Enables factory for generic buffers.
- #define CH_CFG_FACTORY_SEMAPHORES TRUE
  Enables factory for semaphores.
- #define CH_CFG_FACTORY_MAILBOXES TRUE
  Enables factory for mailboxes.
- #define CH_CFG_FACTORY_OBJ_FIFOS TRUE
  Enables factory for objects FIFOs.

Debug options

- #define CH_DBG_STATISTICS FALSE
  Debug option, kernel statistics.
- #define CH_DBG_SYSTEM_STATE_CHECK TRUE
  Debug option, system state check.
- #define CH_DBG_ENABLE_CHECKS TRUE
  Debug option, parameters checks.
- #define CH_DBG_ENABLE_ASSERTS TRUE
  Debug option, consistency checks.
- #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_ALL
  Debug option, trace buffer.
- #define CH_DBG_TRACE_BUFFER_SIZE 128
  Trace buffer entries.
- #define CH_DBG_ENABLE_STACK_CHECK TRUE
  Debug option, stack checks.
- #define CH_DBG_FILL_THREADS TRUE
  Debug option, stacks initialization.
- #define CH_DBG_THREADS_PROFILING FALSE
  Debug option, threads profiling.

Kernel hooks

- #define CH_CFG_SYSTEM_EXTRA_FIELDS /* Add threads custom fields here.*/
  System structure extension.
- #define CH_CFG_SYSTEM_INIT_HOOK(tp)
  System initialization hook.
- #define CH_CFG_THREAD_EXTRA_FIELDS /* Add threads custom fields here.*/
  Threads descriptor structure extension.
- #define CH_CFG_THREAD_INIT_HOOK(tp)
  Threads initialization hook.
- #define CH_CFG_THREAD_EXIT_HOOK(tp)
  Threads finalization hook.
- #define CH_CFG_CONTEXT_SWITCH_HOOK(ntp, otp)
  Context switch hook.
- #define CH_CFG_IRQ_PROLOGUE_HOOK()
  ISR enter hook.
- #define CH_CFG_IRQ_EPILOGUE_HOOK()
  ISR exit hook.
- #define CH_CFG_IDLE_ENTER_HOOK()

Idle thread enter hook.

- #define CH_CFG_IDLE_LEAVE_HOOK()
  Idle thread leave hook.
- #define CH_CFG_IDLE_LOOP_HOOK()
  Idle Loop hook.
- #define CH_CFG_SYSTEM_TICK_HOOK()
  System tick event hook.
- #define CH_CFG_SYSTEM_HALT_HOOK(reason)
  System halt hook.
- #define CH_CFG_TRACE_HOOK(tep)
  Trace hook.

10.7.1 Detailed Description

Configuration file template.

A copy of this file must be placed in each project directory, it contains the application specific kernel settings.

10.8 chdebug.c File Reference

Debug support 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.

10.8.1 Detailed Description

Debug support code.

10.9 chdebug.h File Reference

Debug support macros and structures.

Macros

Debug related settings

- #define CH_DBG_STACK_FILL_VALUE 0x55
  Fill value for thread stack area in debug mode.

Macro Functions

- #define chDbgCheck(c)
  Function parameters check.
- #define chDbgAssert(c, r)
  Condition assertion.

10.9.1 Detailed Description

Debug support macros and structures.

10.10 chdynamic.c File Reference

Dynamic threads code.

```
#include "ch.h"
```

Functions

- thread_t * chThdCreateFromHeap (memory_heap_t *heapp, size_t size, const char *name, tprio_t prio, tfunc_t pf, void *arg)
  Creates a new thread allocating the memory from the heap.
- thread_t * chThdCreateFromMemoryPool (memory_pool_t *mp, const char *name, tprio_t prio, tfunc_t pf, void *arg)
  Creates a new thread allocating the memory from the specified memory pool.

10.10.1 Detailed Description

Dynamic threads code.

10.11 chdynamic.h File Reference

Dynamic threads macros and structures.

Functions

- thread_t * chThdCreateFromHeap (memory_heap_t *heapp, size_t size, const char *name, tprio_t prio, tfunc_t pf, void *arg)

  Creates a new thread allocating the memory from the heap.
- thread_t * chThdCreateFromMemoryPool (memory_pool_t *mp, const char *name, tprio_t prio, tfunc_t pf, void *arg)

  Creates a new thread allocating the memory from the specified memory pool.

10.11.1 Detailed Description

Dynamic threads macros and structures.

10.12 chevents.c File Reference

Events code.

```
#include "ch.h"
```

Functions

- void chEvtRegisterMaskWithFlags (event_source_t *esp, event_listener_t *elp, eventmask_t events, eventflags_t wflags)

  Registers an Event Listener on an Event Source.
- void chEvtUnregister (event_source_t *esp, event_listener_t *elp)

  Unregisters an Event Listener from its Event Source.
- eventmask_t chEvtGetAndClearEventsI (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtGetAndClearEvents (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtAddEvents (eventmask_t events)

  *Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvtSignal()`.*
- void chEvtBroadcastFlagsI (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- eventflags_t chEvtGetAndClearFlags (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- void chEvtSignal (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtSignalI (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtBroadcastFlags (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- eventflags_t chEvtGetAndClearFlagsI (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- void chEvtDispatch (const evhandler_t *handlers, eventmask_t events)

  Invokes the event handlers associated to an event flags mask.
- eventmask_t chEvtWaitOne (eventmask_t events)

  Waits for exactly one of the specified events.
- eventmask_t chEvtWaitAny (eventmask_t events)

Waits for any of the specified events.

- eventmask_t chEvtWaitAll (eventmask_t events)

  Waits for all the specified events.

- eventmask_t chEvtWaitOneTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for exactly one of the specified events.

- eventmask_t chEvtWaitAnyTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for any of the specified events.

- eventmask_t chEvtWaitAllTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for all the specified events.

10.12.1 Detailed Description

Events code.

10.13 chevents.h File Reference

Events macros and structures.

Data Structures

- struct event_listener

  Event Listener structure.

- struct event_source

  Event Source structure.

Macros

- #define ALL_EVENTS ((eventmask_t)-1)

  All events allowed mask.

- #define EVENT_MASK(eid) ((eventmask_t)1 << (eventmask_t)(eid))

  Returns an event mask from an event identifier.

- #define _EVENTSOURCE_DATA(name) {(event_listener_t *)(&name)}

  Data part of a static event source initializer.

- #define EVENTSOURCE_DECL(name) event_source_t name = _EVENTSOURCE_DATA(name)

  Static event source initializer.

Typedefs

- typedef struct event_source event_source_t

  Event Source structure.

- typedef void(* evhandler_t) (eventid_t id)

  Event Handler callback function.

Functions

- void chEvtRegisterMaskWithFlags (event_source_t *esp, event_listener_t *elp, eventmask_t events, eventflags_t wflags)

  Registers an Event Listener on an Event Source.
- void chEvtUnregister (event_source_t *esp, event_listener_t *elp)

  Unregisters an Event Listener from its Event Source.
- eventmask_t chEvtGetAndClearEventsI (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtGetAndClearEvents (eventmask_t events)

  Clears the pending events specified in the events mask.
- eventmask_t chEvtAddEvents (eventmask_t events)

  *Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvtSignal()`.*
- eventflags_t chEvtGetAndClearFlags (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- eventflags_t chEvtGetAndClearFlagsI (event_listener_t *elp)

  Returns the flags associated to an `event_listener_t`.
- void chEvtSignal (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtSignalI (thread_t *tp, eventmask_t events)

  Adds a set of event flags directly to the specified `thread_t`.
- void chEvtBroadcastFlags (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- void chEvtBroadcastFlagsI (event_source_t *esp, eventflags_t flags)

  Signals all the Event Listeners registered on the specified Event Source.
- void chEvtDispatch (const evhandler_t *handlers, eventmask_t events)

  Invokes the event handlers associated to an event flags mask.
- eventmask_t chEvtWaitOne (eventmask_t events)

  Waits for exactly one of the specified events.
- eventmask_t chEvtWaitAny (eventmask_t events)

  Waits for any of the specified events.
- eventmask_t chEvtWaitAll (eventmask_t events)

  Waits for all the specified events.
- eventmask_t chEvtWaitOneTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for exactly one of the specified events.
- eventmask_t chEvtWaitAnyTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for any of the specified events.
- eventmask_t chEvtWaitAllTimeout (eventmask_t events, sysinterval_t timeout)

  Waits for all the specified events.
- static void chEvtObjectInit (event_source_t *esp)

  Initializes an Event Source.
- static void chEvtRegisterMask (event_source_t *esp, event_listener_t *elp, eventmask_t events)

  Registers an Event Listener on an Event Source.
- static void chEvtRegister (event_source_t *esp, event_listener_t *elp, eventid_t event)

  Registers an Event Listener on an Event Source.
- static bool chEvtIsListeningI (event_source_t *esp)

  Verifies if there is at least one `event_listener_t` registered.
- static void chEvtBroadcast (event_source_t *esp)

  Signals all the Event Listeners registered on the specified Event Source.
- static void chEvtBroadcastI (event_source_t *esp)

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

- static eventmask_t chEvtAddEventsI (eventmask_t events)

  *Adds (OR) a set of events to the current thread, this is **much** faster than using `chEvtBroadcast()` or `chEvt↩Signal()`.*

- static eventmask_t chEvtGetEventsX (void)

  Returns the events mask.

10.13.1 Detailed Description

Events macros and structures.

10.14 chfactory.c File Reference

ChibiOS objects factory and registry code.

```
#include <string.h>
#include "ch.h"
```

Functions

- void _factory_init (void)

  Initializes the objects factory.

- registered_object_t * chFactoryRegisterObject (const char *name, void *objp)

  Registers a generic object.

- registered_object_t * chFactoryFindObject (const char *name)

  Retrieves a registered object.

- registered_object_t * chFactoryFindObjectByPointer (void *objp)

  Retrieves a registered object by pointer.

- void chFactoryReleaseObject (registered_object_t *rop)

  Releases a registered object.

- dyn_buffer_t * chFactoryCreateBuffer (const char *name, size_t size)

  Creates a generic dynamic buffer object.

- dyn_buffer_t * chFactoryFindBuffer (const char *name)

  Retrieves a dynamic buffer object.

- void chFactoryReleaseBuffer (dyn_buffer_t *dbp)

  Releases a dynamic buffer object.

- dyn_semaphore_t * chFactoryCreateSemaphore (const char *name, cnt_t n)

  Creates a dynamic semaphore object.

- dyn_semaphore_t * chFactoryFindSemaphore (const char *name)

  Retrieves a dynamic semaphore object.

- void chFactoryReleaseSemaphore (dyn_semaphore_t *dsp)

  Releases a dynamic semaphore object.

- dyn_mailbox_t * chFactoryCreateMailbox (const char *name, size_t n)

  Creates a dynamic mailbox object.

- dyn_mailbox_t * chFactoryFindMailbox (const char *name)

  Retrieves a dynamic mailbox object.

- void chFactoryReleaseMailbox (dyn_mailbox_t *dmp)

  Releases a dynamic mailbox object.

- dyn_objects_fifo_t * chFactoryCreateObjectsFIFO (const char *name, size_t objsize, size_t objn, unsigned objalign)

Creates a dynamic "objects FIFO" object.

- dyn_objects_fifo_t * chFactoryFindObjectsFIFO (const char *name)

  Retrieves a dynamic "objects FIFO" object.

- void chFactoryReleaseObjectsFIFO (dyn_objects_fifo_t *dofp)

  Releases a dynamic "objects FIFO" object.

Variables

- objects_factory_t ch_factory

  Factory object static instance.

10.14.1 Detailed Description

ChibiOS objects factory and registry code.

10.15 chfactory.h File Reference

ChibiOS objects factory structures and macros.

Data Structures

- struct ch_dyn_element

  Type of a dynamic object list element.

- struct ch_dyn_list

  Type of a dynamic object list.

- struct ch_registered_static_object

  Type of a registered object.

- struct ch_dyn_object

  Type of a dynamic buffer object.

- struct ch_dyn_semaphore

  Type of a dynamic semaphore.

- struct ch_dyn_mailbox

  Type of a dynamic buffer object.

- struct ch_dyn_objects_fifo

  Type of a dynamic buffer object.

- struct ch_objects_factory

  Type of the factory main object.

Macros

- #define CH_CFG_FACTORY_MAX_NAMES_LENGTH 8

  Maximum length for object names.

- #define CH_CFG_FACTORY_OBJECTS_REGISTRY TRUE

  Enables the registry of generic objects.

- #define CH_CFG_FACTORY_GENERIC_BUFFERS TRUE

  Enables factory for generic buffers.

- #define CH_CFG_FACTORY_SEMAPHORES TRUE

  Enables factory for semaphores.

- #define CH_CFG_FACTORY_MAILBOXES TRUE

Enables factory for mailboxes.

- #define CH_CFG_FACTORY_OBJ_FIFOS TRUE

  Enables factory for objects FIFOs.
- #define CH_CFG_FACTORY_SEMAPHORES FALSE

  Enables factory for semaphores.
- #define CH_CFG_FACTORY_MAILBOXES FALSE

  Enables factory for mailboxes.
- #define CH_CFG_FACTORY_OBJ_FIFOS FALSE

  Enables factory for objects FIFOs.

Typedefs

- typedef struct ch_dyn_element dyn_element_t

  Type of a dynamic object list element.
- typedef struct ch_dyn_list dyn_list_t

  Type of a dynamic object list.
- typedef struct ch_registered_static_object registered_object_t

  Type of a registered object.
- typedef struct ch_dyn_object dyn_buffer_t

  Type of a dynamic buffer object.
- typedef struct ch_dyn_semaphore dyn_semaphore_t

  Type of a dynamic semaphore.
- typedef struct ch_dyn_mailbox dyn_mailbox_t

  Type of a dynamic buffer object.
- typedef struct ch_dyn_objects_fifo dyn_objects_fifo_t

  Type of a dynamic buffer object.
- typedef struct ch_objects_factory objects_factory_t

  Type of the factory main object.

Functions

- void _factory_init (void)

  Initializes the objects factory.
- registered_object_t * chFactoryRegisterObject (const char *name, void *objp)

  Registers a generic object.
- registered_object_t * chFactoryFindObject (const char *name)

  Retrieves a registered object.
- registered_object_t * chFactoryFindObjectByPointer (void *objp)

  Retrieves a registered object by pointer.
- void chFactoryReleaseObject (registered_object_t *rop)

  Releases a registered object.
- dyn_buffer_t * chFactoryCreateBuffer (const char *name, size_t size)

  Creates a generic dynamic buffer object.
- dyn_buffer_t * chFactoryFindBuffer (const char *name)

  Retrieves a dynamic buffer object.
- void chFactoryReleaseBuffer (dyn_buffer_t *dbp)

  Releases a dynamic buffer object.
- dyn_semaphore_t * chFactoryCreateSemaphore (const char *name, cnt_t n)

  Creates a dynamic semaphore object.
- dyn_semaphore_t * chFactoryFindSemaphore (const char *name)

Retrieves a dynamic semaphore object.

- void chFactoryReleaseSemaphore (dyn_semaphore_t *dsp)

  Releases a dynamic semaphore object.
- dyn_mailbox_t * chFactoryCreateMailbox (const char *name, size_t n)

  Creates a dynamic mailbox object.
- dyn_mailbox_t * chFactoryFindMailbox (const char *name)

  Retrieves a dynamic mailbox object.
- void chFactoryReleaseMailbox (dyn_mailbox_t *dmp)

  Releases a dynamic mailbox object.
- dyn_objects_fifo_t * chFactoryCreateObjectsFIFO (const char *name, size_t objsize, size_t objn, unsigned objalign)

  Creates a dynamic "objects FIFO" object.
- dyn_objects_fifo_t * chFactoryFindObjectsFIFO (const char *name)

  Retrieves a dynamic "objects FIFO" object.
- void chFactoryReleaseObjectsFIFO (dyn_objects_fifo_t *dofp)

  Releases a dynamic "objects FIFO" object.
- static dyn_element_t * chFactoryDuplicateReference (dyn_element_t *dep)

  Duplicates an object reference.
- static void * chFactoryGetObject (registered_object_t *rop)

  Returns the pointer to the inner registered object.
- static size_t chFactoryGetBufferSize (dyn_buffer_t *dbp)

  Returns the size of a generic dynamic buffer object.
- static uint8_t * chFactoryGetBuffer (dyn_buffer_t *dbp)

  Returns the pointer to the inner buffer.
- static semaphore_t * chFactoryGetSemaphore (dyn_semaphore_t *dsp)

  Returns the pointer to the inner semaphore.
- static mailbox_t * chFactoryGetMailbox (dyn_mailbox_t *dmp)

  Returns the pointer to the inner mailbox.
- static objects_fifo_t * chFactoryGetObjectsFIFO (dyn_objects_fifo_t *dofp)

  Returns the pointer to the inner objects FIFO.

10.15.1 Detailed Description

ChibiOS objects factory structures and macros.

10.16 chfifo.h File Reference

Objects FIFO structures and macros.

Data Structures

- struct ch_objects_fifo

  Type of an objects FIFO.

Typedefs

- typedef struct ch_objects_fifo objects_fifo_t

  Type of an objects FIFO.

Functions

- static void chFifoObjectInit (objects_fifo_t *ofp, size_t objsize, size_t objn, unsigned objalign, void *objbuf, msg_t *msgbuf)

  Initializes a FIFO object.
- static void * chFifoTakeObjectI (objects_fifo_t *ofp)

  Allocates a free object.
- static void * chFifoTakeObjectTimeoutS (objects_fifo_t *ofp, sysinterval_t timeout)

  Allocates a free object.
- static void * chFifoTakeObjectTimeout (objects_fifo_t *ofp, sysinterval_t timeout)

  Allocates a free object.
- static void chFifoReturnObjectI (objects_fifo_t *ofp, void *objp)

  Releases a fetched object.
- static void chFifoReturnObject (objects_fifo_t *ofp, void *objp)

  Releases a fetched object.
- static void chFifoSendObjectI (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static void chFifoSendObjectS (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static void chFifoSendObject (objects_fifo_t *ofp, void *objp)

  Posts an object.
- static msg_t chFifoReceiveObjectI (objects_fifo_t *ofp, void **objpp)

  Fetches an object.
- static msg_t chFifoReceiveObjectTimeoutS (objects_fifo_t *ofp, void **objpp, sysinterval_t timeout)

  Fetches an object.
- static msg_t chFifoReceiveObjectTimeout (objects_fifo_t *ofp, void **objpp, sysinterval_t timeout)

  Fetches an object.

10.16.1 Detailed Description

Objects FIFO structures and macros.

This module implements a generic FIFO queue of objects by coupling a Guarded Memory Pool (for objects storage) and a MailBox.
On the sender side free objects are taken from the pool, filled and then sent to the receiver, on the receiver side objects are fetched, used and then returned to the pool. Operations defined for object FIFOs:

- **Take**: An object is taken from the pool of the free objects, can be blocking.
- **Return**: An object is returned to the pool of the free objects, it is guaranteed to be non-blocking.
- **Send**: An object is sent through the mailbox, it is guaranteed to be non-blocking
- **Receive**: An object is received from the mailbox, can be blocking.

10.17 chheap.c File Reference

Heaps code.

```
#include "ch.h"
```

Functions

- void _heap_init (void)

  Initializes the default heap.
- void chHeapObjectInit (memory_heap_t *heapp, void *buf, size_t size)

  Initializes a memory heap from a static memory area.
- void * chHeapAllocAligned (memory_heap_t *heapp, size_t size, unsigned align)

  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 (memory_heap_t *heapp, size_t *totalp, size_t *largestp)

  Reports the heap status.

Variables

- static memory_heap_t default_heap

  Default heap descriptor.

10.17.1 Detailed Description

Heaps code.

10.18 chheap.h File Reference

Heaps macros and structures.

Data Structures

- union heap_header

  Memory heap block header.
- struct memory_heap

  Structure describing a memory heap.

Macros

- #define CH_HEAP_ALIGNMENT 8U

  Minimum alignment used for heap.
- #define CH_HEAP_AREA(name, size)

  Allocation of an aligned static heap buffer.

Typedefs

- typedef struct memory_heap memory_heap_t

  Type of a memory heap.
- typedef union heap_header heap_header_t

  Type of a memory heap header.

Functions

- void _heap_init (void)

  Initializes the default heap.
- void chHeapObjectInit (memory_heap_t *heapp, void *buf, size_t size)

  Initializes a memory heap from a static memory area.
- void * chHeapAllocAligned (memory_heap_t *heapp, size_t size, unsigned align)

  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 (memory_heap_t *heapp, size_t *totalp, size_t *largestp)

  Reports the heap status.
- static void * chHeapAlloc (memory_heap_t *heapp, size_t size)

  Allocates a block of memory from the heap by using the first-fit algorithm.
- static size_t chHeapGetSize (const void *p)

  Returns the size of an allocated block.

10.18.1 Detailed Description

Heaps macros and structures.

10.19 chmboxes.c File Reference

Mailboxes code.

```
#include "ch.h"
```

Functions

- void chMBObjectInit (mailbox_t *mbp, msg_t *buf, size_t n)

  Initializes a `mailbox_t` object.
- void chMBReset (mailbox_t *mbp)

  Resets a `mailbox_t` object.
- void chMBResetI (mailbox_t *mbp)

  Resets a `mailbox_t` object.
- msg_t chMBPostTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts a message into a mailbox.
- msg_t chMBPostTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts a message into a mailbox.
- msg_t chMBPostI (mailbox_t *mbp, msg_t msg)

  Posts a message into a mailbox.
- msg_t chMBPostAheadTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)

  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadI (mailbox_t *mbp, msg_t msg)

  Posts an high priority message into a mailbox.
- msg_t chMBFetchTimeout (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)

  Retrieves a message from a mailbox.

- msg_t chMBFetchTimeoutS (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)
  Retrieves a message from a mailbox.
- msg_t chMBFetchI (mailbox_t *mbp, msg_t *msgp)
  Retrieves a message from a mailbox.

10.19.1 Detailed Description

Mailboxes code.

10.20 chmboxes.h File Reference

Mailboxes macros and structures.

Data Structures

- struct mailbox_t
  Structure representing a mailbox object.

Macros

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

Functions

- void chMBObjectInit (mailbox_t *mbp, msg_t *buf, size_t n)
  Initializes a `mailbox_t` object.
- void chMBReset (mailbox_t *mbp)
  Resets a `mailbox_t` object.
- void chMBResetI (mailbox_t *mbp)
  Resets a `mailbox_t` object.
- msg_t chMBPostTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)
  Posts a message into a mailbox.
- msg_t chMBPostTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)
  Posts a message into a mailbox.
- msg_t chMBPostI (mailbox_t *mbp, msg_t msg)
  Posts a message into a mailbox.
- msg_t chMBPostAheadTimeout (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)
  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadTimeoutS (mailbox_t *mbp, msg_t msg, sysinterval_t timeout)
  Posts an high priority message into a mailbox.
- msg_t chMBPostAheadI (mailbox_t *mbp, msg_t msg)
  Posts an high priority message into a mailbox.
- msg_t chMBFetchTimeout (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)
  Retrieves a message from a mailbox.
- msg_t chMBFetchTimeoutS (mailbox_t *mbp, msg_t *msgp, sysinterval_t timeout)
  Retrieves a message from a mailbox.

- msg_t chMBFetchI (mailbox_t *mbp, msg_t *msgp)

  Retrieves a message from a mailbox.
- static size_t chMBGetSizeI (const mailbox_t *mbp)

  Returns the mailbox buffer size as number of messages.
- static size_t chMBGetUsedCountI (const mailbox_t *mbp)

  Returns the number of used message slots into a mailbox.
- static size_t chMBGetFreeCountI (const mailbox_t *mbp)

  Returns the number of free message slots into a mailbox.
- static msg_t chMBPeekI (const mailbox_t *mbp)

  Returns the next message in the queue without removing it.
- static void chMBResumeX (mailbox_t *mbp)

  Terminates the reset state.

10.20.1 Detailed Description

Mailboxes macros and structures.

10.21 chmemcore.c File Reference

Core memory manager code.

```
#include "ch.h"
```

Functions

- void _core_init (void)

  Low level memory manager initialization.
- void * chCoreAllocAlignedWithOffsetI (size_t size, unsigned align, size_t offset)

  Allocates a memory block.
- void * chCoreAllocAlignedWithOffset (size_t size, unsigned align, size_t offset)

  Allocates a memory block.
- size_t chCoreGetStatusX (void)

  Core memory status.

Variables

- memcore_t ch_memcore

  Memory core descriptor.

10.21.1 Detailed Description

Core memory manager code.

10.22 chmemcore.h File Reference

Core memory manager macros and structures.

Data Structures

- struct memcore_t

  Type of memory core object.

Macros

- #define CH_CFG_MEMCORE_SIZE 0

  Managed RAM size.

Typedefs

- typedef void *(* memgetfunc_t) (size_t size, unsigned align)

  Memory get function.
- typedef void *(* memgetfunc2_t) (size_t size, unsigned align, size_t offset)

  Enhanced memory get function.

Functions

- void _core_init (void)

  Low level memory manager initialization.
- void * chCoreAllocAlignedWithOffsetI (size_t size, unsigned align, size_t offset)

  Allocates a memory block.
- void * chCoreAllocAlignedWithOffset (size_t size, unsigned align, size_t offset)

  Allocates a memory block.
- size_t chCoreGetStatusX (void)

  Core memory status.
- static void * chCoreAllocAlignedI (size_t size, unsigned align)

  Allocates a memory block.
- static void * chCoreAllocAligned (size_t size, unsigned align)

  Allocates a memory block.
- static void * chCoreAllocI (size_t size)

  Allocates a memory block.
- static void * chCoreAlloc (size_t size)

  Allocates a memory block.

10.22.1 Detailed Description

Core memory manager macros and structures.

10.23 chmempools.c File Reference

Memory Pools code.

```
#include "ch.h"
```

Functions

- void chPoolObjectInitAligned (memory_pool_t *mp, size_t size, unsigned align, memgetfunc_t provider)

  Initializes an empty memory pool.
- void chPoolLoadArray (memory_pool_t *mp, void *p, size_t n)

  Loads a memory pool with an array of static objects.
- void * chPoolAllocI (memory_pool_t *mp)

  Allocates an object from a memory pool.
- void * chPoolAlloc (memory_pool_t *mp)

  Allocates an object from a memory pool.
- void chPoolFreeI (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chPoolFree (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chGuardedPoolObjectInitAligned (guarded_memory_pool_t *gmp, size_t size, unsigned align)

  Initializes an empty guarded memory pool.
- void chGuardedPoolLoadArray (guarded_memory_pool_t *gmp, void *p, size_t n)

  Loads a guarded memory pool with an array of static objects.
- void * chGuardedPoolAllocTimeoutS (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void * chGuardedPoolAllocTimeout (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void chGuardedPoolFreeI (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.
- void chGuardedPoolFree (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.

10.23.1 Detailed Description

Memory Pools code.

10.24 chmempools.h File Reference

Memory Pools macros and structures.

Data Structures

- struct pool_header

  Memory pool free object header.
- struct memory_pool_t

  Memory pool descriptor.
- struct guarded_memory_pool_t

  Guarded memory pool descriptor.

Macros

- #define _MEMORYPOOL_DATA(name, size, align, provider) {NULL, size, align, provider}

  Data part of a static memory pool initializer.
- #define MEMORYPOOL_DECL(name, size, align, provider) memory_pool_t name = _MEMORYPOOL_DATA(name, size, align, provider)

  Static memory pool initializer.
- #define _GUARDEDMEMORYPOOL_DATA(name, size, align)

  Data part of a static guarded memory pool initializer.
- #define GUARDEDMEMORYPOOL_DECL(name, size, align) guarded_memory_pool_t name = _GUARDEDMEMORYPOOL_DATA(name, size, align)

  Static guarded memory pool initializer.

Functions

- void chPoolObjectInitAligned (memory_pool_t *mp, size_t size, unsigned align, memgetfunc_t provider)

  Initializes an empty memory pool.
- void chPoolLoadArray (memory_pool_t *mp, void *p, size_t n)

  Loads a memory pool with an array of static objects.
- void * chPoolAllocI (memory_pool_t *mp)

  Allocates an object from a memory pool.
- void * chPoolAlloc (memory_pool_t *mp)

  Allocates an object from a memory pool.
- void chPoolFreeI (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chPoolFree (memory_pool_t *mp, void *objp)

  Releases an object into a memory pool.
- void chGuardedPoolObjectInitAligned (guarded_memory_pool_t *gmp, size_t size, unsigned align)

  Initializes an empty guarded memory pool.
- void chGuardedPoolLoadArray (guarded_memory_pool_t *gmp, void *p, size_t n)

  Loads a guarded memory pool with an array of static objects.
- void * chGuardedPoolAllocTimeoutS (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void * chGuardedPoolAllocTimeout (guarded_memory_pool_t *gmp, sysinterval_t timeout)

  Allocates an object from a guarded memory pool.
- void chGuardedPoolFreeI (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.
- void chGuardedPoolFree (guarded_memory_pool_t *gmp, void *objp)

  Releases an object into a guarded memory pool.
- static void chPoolObjectInit (memory_pool_t *mp, size_t size, memgetfunc_t provider)

  Initializes an empty memory pool.
- static void chPoolAdd (memory_pool_t *mp, void *objp)

  Adds an object to a memory pool.
- static void chPoolAddI (memory_pool_t *mp, void *objp)

  Adds an object to a memory pool.
- static void chGuardedPoolObjectInit (guarded_memory_pool_t *gmp, size_t size)

  Initializes an empty guarded memory pool.
- static void chGuardedPoolAdd (guarded_memory_pool_t *gmp, void *objp)

  Adds an object to a guarded memory pool.
- static void chGuardedPoolAddI (guarded_memory_pool_t *gmp, void *objp)

  Adds an object to a guarded memory pool.
- static void * chGuardedPoolAllocI (guarded_memory_pool_t *gmp)

  Allocates an object from a guarded memory pool.

10.24.1 Detailed Description

Memory Pools macros and structures.

10.25 chmsg.c File Reference

Messages code.

```
#include "ch.h"
```

Functions

- msg_t chMsgSend (thread_t *tp, msg_t msg)

  Sends a message to the specified thread.
- thread_t * chMsgWait (void)

  Suspends the thread and waits for an incoming message.
- void chMsgRelease (thread_t *tp, msg_t msg)

  Releases a sender thread specifying a response message.

10.25.1 Detailed Description

Messages code.

10.26 chmsg.h File Reference

Messages macros and structures.

Functions

- msg_t chMsgSend (thread_t *tp, msg_t msg)

  Sends a message to the specified thread.
- thread_t * chMsgWait (void)

  Suspends the thread and waits for an incoming message.
- void chMsgRelease (thread_t *tp, msg_t msg)

  Releases a sender thread specifying a response message.
- static bool chMsgIsPendingI (thread_t *tp)

  Evaluates to `true` if the thread has pending messages.
- static msg_t chMsgGet (thread_t *tp)

  Returns the message carried by the specified thread.
- static void chMsgReleaseS (thread_t *tp, msg_t msg)

  Releases the thread waiting on top of the messages queue.

10.26.1 Detailed Description

Messages macros and structures.

10.27 chmtx.c File Reference

Mutexes code.

```
#include "ch.h"
```

Functions

- void chMtxObjectInit (mutex_t *mp)

  Initializes s `mutex_t` structure.
- void chMtxLock (mutex_t *mp)

  Locks the specified mutex.
- void chMtxLockS (mutex_t *mp)

  Locks the specified mutex.
- bool chMtxTryLock (mutex_t *mp)

  Tries to lock a mutex.
- bool chMtxTryLockS (mutex_t *mp)

  Tries to lock a mutex.
- void chMtxUnlock (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockS (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockAllS (void)

  Unlocks all mutexes owned by the invoking thread.
- void chMtxUnlockAll (void)

  Unlocks all mutexes owned by the invoking thread.

10.27.1 Detailed Description

Mutexes code.

10.28 chmtx.h File Reference

Mutexes macros and structures.

Data Structures

- struct ch_mutex

  Mutex structure.

Macros

- #define _MUTEX_DATA(name) {_THREADS_QUEUE_DATA(name.queue), NULL, NULL, 0}

  Data part of a static mutex initializer.
- #define MUTEX_DECL(name) mutex_t name = _MUTEX_DATA(name)

  Static mutex initializer.

Typedefs

- typedef struct ch_mutex mutex_t

  Type of a mutex structure.

Functions

- void chMtxObjectInit (mutex_t *mp)

  Initializes s `mutex_t` structure.
- void chMtxLock (mutex_t *mp)

  Locks the specified mutex.
- void chMtxLockS (mutex_t *mp)

  Locks the specified mutex.
- bool chMtxTryLock (mutex_t *mp)

  Tries to lock a mutex.
- bool chMtxTryLockS (mutex_t *mp)

  Tries to lock a mutex.
- void chMtxUnlock (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockS (mutex_t *mp)

  Unlocks the specified mutex.
- void chMtxUnlockAll (void)

  Unlocks all mutexes owned by the invoking thread.
- void chMtxUnlockAllS (void)

  Unlocks all mutexes owned by the invoking thread.
- static bool chMtxQueueNotEmptyS (mutex_t *mp)

  Returns `true` if the mutex queue contains at least a waiting thread.
- static mutex_t * chMtxGetNextMutexS (void)

  Returns the next mutex in the mutexes stack of the current thread.

10.28.1 Detailed Description

Mutexes macros and structures.

10.29 chregistry.c File Reference

Threads registry code.

```
#include <string.h>
#include "ch.h"
```

Functions

- thread_t * chRegFirstThread (void)

  Returns the first thread in the system.
- thread_t * chRegNextThread (thread_t *tp)

  Returns the thread next to the specified one.
- thread_t * chRegFindThreadByName (const char *name)

  Retrieves a thread pointer by name.

- thread_t * chRegFindThreadByPointer (thread_t *tp)

  Confirms that a pointer is a valid thread pointer.
- thread_t * chRegFindThreadByWorkingArea (stkalign_t *wa)

  Confirms that a working area is being used by some active thread.

10.29.1 Detailed Description

Threads registry code.

10.30 chregistry.h File Reference

Threads registry macros and structures.

Data Structures

- struct chdebug_t

  ChibiOS/RT memory signature record.

Macros

- #define REG_REMOVE(tp)

  Removes a thread from the registry list.
- #define REG_INSERT(tp)

  Adds a thread to the registry list.

Functions

- thread_t * chRegFirstThread (void)

  Returns the first thread in the system.
- thread_t * chRegNextThread (thread_t *tp)

  Returns the thread next to the specified one.
- thread_t * chRegFindThreadByName (const char *name)

  Retrieves a thread pointer by name.
- thread_t * chRegFindThreadByPointer (thread_t *tp)

  Confirms that a pointer is a valid thread pointer.
- thread_t * chRegFindThreadByWorkingArea (stkalign_t *wa)

  Confirms that a working area is being used by some active thread.
- static void chRegSetThreadName (const char *name)

  Sets the current thread name.
- static const char * chRegGetThreadNameX (thread_t *tp)

  Returns the name of the specified thread.
- static void chRegSetThreadNameX (thread_t *tp, const char *name)

  Changes the name of the specified thread.

10.30.1 Detailed Description

Threads registry macros and structures.

10.31 chrestrictions.h File Reference

Licensing restrictions header.

10.31.1 Detailed Description

Licensing restrictions header.

10.32 chschd.c File Reference

Scheduler code.

```
#include "ch.h"
```

Functions

- void _scheduler_init (void)

  Scheduler initialization.
- void queue_prio_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a priority ordered queue.
- void queue_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a queue.
- thread_t * queue_fifo_remove (threads_queue_t *tqp)

  Removes the first-out thread from a queue and returns it.
- thread_t * queue_lifo_remove (threads_queue_t *tqp)

  Removes the last-out thread from a queue and returns it.
- thread_t * queue_dequeue (thread_t *tp)

  Removes a thread from a queue and returns it.
- void list_insert (thread_t *tp, threads_list_t *tlp)

  Pushes a thread_t on top of a stack list.
- thread_t * list_remove (threads_list_t *tlp)

  Pops a thread from the top of a stack list and returns it.
- thread_t * chSchReadyI (thread_t *tp)

  Inserts a thread in the Ready List placing it behind its peers.
- thread_t * chSchReadyAheadI (thread_t *tp)

  Inserts a thread in the Ready List placing it ahead its peers.
- void chSchGoSleepS (tstate_t newstate)

  Puts the current thread to sleep into the specified state.
- msg_t chSchGoSleepTimeoutS (tstate_t newstate, sysinterval_t timeout)

  Puts the current thread to sleep into the specified state with timeout specification.
- void chSchWakeupS (thread_t *ntp, msg_t msg)

  Wakes up a thread.
- void chSchRescheduleS (void)

  Performs a reschedule if a higher priority thread is runnable.
- bool chSchIsPreemptionRequired (void)

  Evaluates if preemption is required.
- void chSchDoRescheduleBehind (void)

  Switches to the first thread on the runnable queue.
- void chSchDoRescheduleAhead (void)

Switches to the first thread on the runnable queue.

- void chSchDoReschedule (void)

  Switches to the first thread on the runnable queue.

Variables

- ch_system_t ch

  System data structures.

10.32.1 Detailed Description

Scheduler code.

10.33 chschd.h File Reference

Scheduler macros and structures.

Data Structures

- struct ch_threads_list

  Generic threads single link list, it works like a stack.
- struct ch_threads_queue

  Generic threads bidirectional linked list header and element.
- struct ch_thread

  Structure representing a thread.
- struct ch_virtual_timer

  Virtual Timer descriptor structure.
- struct ch_virtual_timers_list

  Virtual timers list header.
- struct ch_system_debug

  System debug data structure.
- struct ch_system

  System data structure.

Macros

- #define firstprio(rlp) ((rlp)->next->prio)

  Returns the priority of the first thread on the given ready list.
- #define currp ch.rlist.current

  Current thread pointer access macro.

Wakeup status codes

- #define MSG_OK (msg_t)0

  Normal wakeup message.
- #define MSG_TIMEOUT (msg_t)-1

  Wakeup caused by a timeout condition.
- #define MSG_RESET (msg_t)-2

  Wakeup caused by a reset condition.

Priority constants

- #define NOPRIO (tprio_t)0
  Ready list header priority.
- #define IDLEPRIO (tprio_t)1
  Idle priority.
- #define LOWPRIO (tprio_t)2
  Lowest priority.
- #define NORMALPRIO (tprio_t)128
  Normal priority.
- #define HIGHPRIO (tprio_t)255
  Highest priority.

Thread states

- #define CH_STATE_READY (tstate_t)0
  Waiting on the ready list.
- #define CH_STATE_CURRENT (tstate_t)1
  Currently running.
- #define CH_STATE_WTSTART (tstate_t)2
  Just created.
- #define CH_STATE_SUSPENDED (tstate_t)3
  Suspended state.
- #define CH_STATE_QUEUED (tstate_t)4
  On an I/O queue.
- #define CH_STATE_WTSEM (tstate_t)5
  On a semaphore.
- #define CH_STATE_WTMTX (tstate_t)6
  On a mutex.
- #define CH_STATE_WTCOND (tstate_t)7
  On a cond.variable.
- #define CH_STATE_SLEEPING (tstate_t)8
  Sleeping.
- #define CH_STATE_WTEXIT (tstate_t)9
  Waiting a thread.
- #define CH_STATE_WTOREVT (tstate_t)10
  One event.
- #define CH_STATE_WTANDEVT (tstate_t)11
  Several events.
- #define CH_STATE_SNDMSGQ (tstate_t)12
  Sending a message, in queue.
- #define CH_STATE_SNDMSG (tstate_t)13
  Sent a message, waiting answer.
- #define CH_STATE_WTMSG (tstate_t)14
  Waiting for a message.
- #define CH_STATE_FINAL (tstate_t)15
  Thread terminated.
- #define CH_STATE_NAMES
  Thread states as array of strings.

Thread flags and attributes

- #define CH_FLAG_MODE_MASK (tmode_t)3U
  Thread memory mode mask.
- #define CH_FLAG_MODE_STATIC (tmode_t)0U
  Static thread.
- #define CH_FLAG_MODE_HEAP (tmode_t)1U
  Thread allocated from a Memory Heap.
- #define CH_FLAG_MODE_MPOOL (tmode_t)2U
  Thread allocated from a Memory Pool.
- #define CH_FLAG_TERMINATE (tmode_t)4U
  Termination requested flag.

Functions

- void _scheduler_init (void)

  Scheduler initialization.
- thread_t * chSchReadyI (thread_t *tp)

  Inserts a thread in the Ready List placing it behind its peers.
- thread_t * chSchReadyAheadI (thread_t *tp)

  Inserts a thread in the Ready List placing it ahead its peers.
- void chSchGoSleepS (tstate_t newstate)

  Puts the current thread to sleep into the specified state.
- msg_t chSchGoSleepTimeoutS (tstate_t newstate, sysinterval_t timeout)

  Puts the current thread to sleep into the specified state with timeout specification.
- void chSchWakeupS (thread_t *ntp, msg_t msg)

  Wakes up a thread.
- void chSchRescheduleS (void)

  Performs a reschedule if a higher priority thread is runnable.
- bool chSchIsPreemptionRequired (void)

  Evaluates if preemption is required.
- void chSchDoRescheduleBehind (void)

  Switches to the first thread on the runnable queue.
- void chSchDoRescheduleAhead (void)

  Switches to the first thread on the runnable queue.
- void chSchDoReschedule (void)

  Switches to the first thread on the runnable queue.
- void queue_prio_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a priority ordered queue.
- void queue_insert (thread_t *tp, threads_queue_t *tqp)

  Inserts a thread into a queue.
- thread_t * queue_fifo_remove (threads_queue_t *tqp)

  Removes the first-out thread from a queue and returns it.
- thread_t * queue_lifo_remove (threads_queue_t *tqp)

  Removes the last-out thread from a queue and returns it.
- thread_t * queue_dequeue (thread_t *tp)

  Removes a thread from a queue and returns it.
- void list_insert (thread_t *tp, threads_list_t *tlp)

  Pushes a thread_t on top of a stack list.
- thread_t * list_remove (threads_list_t *tlp)

  Pops a thread from the top of a stack list and returns it.
- static void list_init (threads_list_t *tlp)

  Threads list initialization.
- static bool list_isempty (threads_list_t *tlp)

  Evaluates to `true` if the specified threads list is empty.
- static bool list_notempty (threads_list_t *tlp)

  Evaluates to `true` if the specified threads list is not empty.
- static void queue_init (threads_queue_t *tqp)

  Threads queue initialization.
- static bool queue_isempty (const threads_queue_t *tqp)

  Evaluates to `true` if the specified threads queue is empty.
- static bool queue_notempty (const threads_queue_t *tqp)

  Evaluates to `true` if the specified threads queue is not empty.
- static bool chSchIsRescRequiredI (void)

Determines if the current thread must reschedule.

- static bool chSchCanYieldS (void)

  Determines if yielding is possible.

- static void chSchDoYieldS (void)

  Yields the time slot.

- static void chSchPreemption (void)

  Inline-able preemption code.

10.33.1 Detailed Description

Scheduler macros and structures.

10.34 chsem.c File Reference

Semaphores code.

```
#include "ch.h"
```

Functions

- void chSemObjectInit (semaphore_t *sp, cnt_t n)

  Initializes a semaphore with the specified counter value.

- void chSemReset (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.

- void chSemResetI (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.

- msg_t chSemWait (semaphore_t *sp)

  Performs a wait operation on a semaphore.

- msg_t chSemWaitS (semaphore_t *sp)

  Performs a wait operation on a semaphore.

- msg_t chSemWaitTimeout (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.

- msg_t chSemWaitTimeoutS (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.

- void chSemSignal (semaphore_t *sp)

  Performs a signal operation on a semaphore.

- void chSemSignalI (semaphore_t *sp)

  Performs a signal operation on a semaphore.

- void chSemAddCounterI (semaphore_t *sp, cnt_t n)

  Adds the specified value to the semaphore counter.

- msg_t chSemSignalWait (semaphore_t *sps, semaphore_t *spw)

  Performs atomic signal and wait operations on two semaphores.

10.34.1 Detailed Description

Semaphores code.

10.35 chsem.h File Reference

Semaphores macros and structures.

Data Structures

- struct ch_semaphore

  Semaphore structure.

Macros

- #define _SEMAPHORE_DATA(name, n) {_THREADS_QUEUE_DATA(name.queue), n}

  Data part of a static semaphore initializer.
- #define SEMAPHORE_DECL(name, n) semaphore_t name = _SEMAPHORE_DATA(name, n)

  Static semaphore initializer.

Typedefs

- typedef struct ch_semaphore semaphore_t

  Semaphore structure.

Functions

- void chSemObjectInit (semaphore_t *sp, cnt_t n)

  Initializes a semaphore with the specified counter value.
- void chSemReset (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.
- void chSemResetI (semaphore_t *sp, cnt_t n)

  Performs a reset operation on the semaphore.
- msg_t chSemWait (semaphore_t *sp)

  Performs a wait operation on a semaphore.
- msg_t chSemWaitS (semaphore_t *sp)

  Performs a wait operation on a semaphore.
- msg_t chSemWaitTimeout (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.
- msg_t chSemWaitTimeoutS (semaphore_t *sp, sysinterval_t timeout)

  Performs a wait operation on a semaphore with timeout specification.
- void chSemSignal (semaphore_t *sp)

  Performs a signal operation on a semaphore.
- void chSemSignalI (semaphore_t *sp)

  Performs a signal operation on a semaphore.
- void chSemAddCounterI (semaphore_t *sp, cnt_t n)

  Adds the specified value to the semaphore counter.
- msg_t chSemSignalWait (semaphore_t *sps, semaphore_t *spw)

  Performs atomic signal and wait operations on two semaphores.
- static void chSemFastWaitI (semaphore_t *sp)

  Decreases the semaphore counter.
- static void chSemFastSignalI (semaphore_t *sp)

  Increases the semaphore counter.
- static cnt_t chSemGetCounterI (const semaphore_t *sp)

  Returns the semaphore counter current value.

10.35.1 Detailed Description

Semaphores macros and structures.

10.36 chstats.c File Reference

Statistics module code.

```
#include "ch.h"
```

Functions

- void _stats_init (void)
  Initializes the statistics module.
- void _stats_increase_irq (void)
  Increases the IRQ counter.
- void _stats_ctxswc (thread_t *ntp, thread_t *otp)
  Updates context switch related statistics.
- void _stats_start_measure_crit_thd (void)
  Starts the measurement of a thread critical zone.
- void _stats_stop_measure_crit_thd (void)
  Stops the measurement of a thread critical zone.
- void _stats_start_measure_crit_isr (void)
  Starts the measurement of an ISR critical zone.
- void _stats_stop_measure_crit_isr (void)
  Stops the measurement of an ISR critical zone.

10.36.1 Detailed Description

Statistics module code.

10.37 chstats.h File Reference

Statistics module macros and structures.

Data Structures

- struct kernel_stats_t
  Type of a kernel statistics structure.

Functions

- void _stats_init (void)
  Initializes the statistics module.
- void _stats_increase_irq (void)
  Increases the IRQ counter.
- void _stats_ctxswc (thread_t *ntp, thread_t *otp)
  Updates context switch related statistics.

- void _stats_start_measure_crit_thd (void)
  Starts the measurement of a thread critical zone.
- void _stats_stop_measure_crit_thd (void)
  Stops the measurement of a thread critical zone.
- void _stats_start_measure_crit_isr (void)
  Starts the measurement of an ISR critical zone.
- void _stats_stop_measure_crit_isr (void)
  Stops the measurement of an ISR critical zone.

10.37.1 Detailed Description

Statistics module macros and structures.

10.38 chsys.c File Reference

System related code.

```
#include "ch.h"
```

Functions

- THD_WORKING_AREA (ch_idle_thread_wa, PORT_IDLE_THREAD_STACK_SIZE)
  Idle thread working area.
- static void _idle_thread (void *p)
  This function implements the idle thread infinite loop.
- void chSysInit (void)
  ChibiOS/RT initialization.
- void chSysHalt (const char *reason)
  Halts the system.
- bool chSysIntegrityCheckI (unsigned testmask)
  System integrity check.
- void chSysTimerHandlerI (void)
  Handles time ticks for round robin preemption and timer increments.
- syssts_t chSysGetStatusAndLockX (void)
  Returns the execution status and enters a critical zone.
- void chSysRestoreStatusX (syssts_t sts)
  Restores the specified execution status and leaves a critical zone.
- bool chSysIsCounterWithinX (rtcnt_t cnt, rtcnt_t start, rtcnt_t end)
  Realtime window test.
- void chSysPolledDelayX (rtcnt_t cycles)
  Polled delay.

10.38.1 Detailed Description

System related code.

10.39 chsys.h File Reference

System related macros and structures.

Macros

- #define chSysGetRealtimeCounterX() (rtcnt_t)port_rt_get_counter_value()

  Returns the current value of the system real time counter.
- #define chSysSwitch(ntp, otp)

  Performs a context switch.

Masks of executable integrity checks.

- #define **CH_INTEGRITY_RLIST** 1U
- #define **CH_INTEGRITY_VTLIST** 2U
- #define **CH_INTEGRITY_REGISTRY** 4U
- #define **CH_INTEGRITY_PORT** 8U

ISRs abstraction macros

- #define CH_IRQ_IS_VALID_PRIORITY(prio) PORT_IRQ_IS_VALID_PRIORITY(prio)

  Priority level validation macro.
- #define CH_IRQ_IS_VALID_KERNEL_PRIORITY(prio) PORT_IRQ_IS_VALID_KERNEL_PRIORITY(prio)

  Priority level validation macro.
- #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.

Time conversion utilities for the realtime counter

- #define S2RTC(freq, sec) ((freq) * (sec))

  Seconds to realtime counter.
- #define MS2RTC(freq, msec) (rtcnt_t)((((freq) + 999UL) / 1000UL) * (msec))

  Milliseconds to realtime counter.
- #define US2RTC(freq, usec) (rtcnt_t)((((freq) + 999999UL) / 1000000UL) * (usec))

  Microseconds to realtime counter.
- #define RTC2S(freq, n) ((((n) - 1UL) / (freq)) + 1UL)

  Realtime counter cycles to seconds.
- #define RTC2MS(freq, n) ((((n) - 1UL) / ((freq) / 1000UL)) + 1UL)

  Realtime counter cycles to milliseconds.
- #define RTC2US(freq, n) ((((n) - 1UL) / ((freq) / 1000000UL)) + 1UL)

  Realtime counter cycles to microseconds.

Functions

- void chSysInit (void)

  ChibiOS/RT initialization.
- bool chSysIntegrityCheckI (unsigned testmask)

  System integrity check.
- void chSysTimerHandlerI (void)

  Handles time ticks for round robin preemption and timer increments.

- syssts_t chSysGetStatusAndLockX (void)

  Returns the execution status and enters a critical zone.
- void chSysRestoreStatusX (syssts_t sts)

  Restores the specified execution status and leaves a critical zone.
- bool chSysIsCounterWithinX (rtcnt_t cnt, rtcnt_t start, rtcnt_t end)

  Realtime window test.
- void chSysPolledDelayX (rtcnt_t cycles)

  Polled delay.
- static void chSysDisable (void)

  Raises the system interrupt priority mask to the maximum level.
- static void chSysSuspend (void)

  Raises the system interrupt priority mask to system level.
- static void chSysEnable (void)

  Lowers the system interrupt priority mask to user level.
- static void chSysLock (void)

  Enters the kernel lock state.
- static void chSysUnlock (void)

  Leaves the kernel lock state.
- static void chSysLockFromISR (void)

  Enters the kernel lock state from within an interrupt handler.
- static void chSysUnlockFromISR (void)

  Leaves the kernel lock state from within an interrupt handler.
- static void chSysUnconditionalLock (void)

  Unconditionally enters the kernel lock state.
- static void chSysUnconditionalUnlock (void)

  Unconditionally leaves the kernel lock state.
- static thread_t * chSysGetIdleThreadX (void)

  Returns a pointer to the idle thread.

10.39.1 Detailed Description

System related macros and structures.

10.40 chsystypes.h File Reference

System types header.

Macros

- #define __CH_STRINGIFY(a) #a

  Utility to make the parameter a quoted string.

Typedefs

- typedef struct ch_thread thread_t

  Type of a thread structure.
- typedef thread_t * thread_reference_t

  Type of a thread reference.
- typedef struct ch_threads_list threads_list_t

Type of a generic threads single link list, it works like a stack.

- typedef struct ch_threads_queue threads_queue_t

  Type of a generic threads bidirectional linked list header and element.
- typedef struct ch_ready_list ready_list_t

  Type of a ready list header.
- typedef void(∗ vtfunc_t) (void ∗p)

  Type of a Virtual Timer callback function.
- typedef struct ch_virtual_timer virtual_timer_t

  Type of a Virtual Timer structure.
- typedef struct ch_virtual_timers_list virtual_timers_list_t

  Type of virtual timers list header.
- typedef struct ch_system_debug system_debug_t

  Type of a system debug structure.
- typedef struct ch_system ch_system_t

  Type of system data structure.

10.40.1 Detailed Description

System types header.

10.41 chthreads.c File Reference

Threads code.

```
#include "ch.h"
```

Functions

- thread_t ∗ _thread_init (thread_t ∗tp, const char ∗name, tprio_t prio)

  Initializes a thread structure.
- void _thread_memfill (uint8_t ∗startp, uint8_t ∗endp, uint8_t v)

  Memory fill utility.
- thread_t ∗ chThdCreateSuspendedI (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateSuspended (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateI (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreate (const thread_descriptor_t ∗tdp)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdCreateStatic (void ∗wsp, size_t size, tprio_t prio, tfunc_t pf, void ∗arg)

  Creates a new thread into a static memory area.
- thread_t ∗ chThdStart (thread_t ∗tp)

  Resumes a thread created with `chThdCreateI()`.
- thread_t ∗ chThdAddRef (thread_t ∗tp)

  Adds a reference to a thread object.
- void chThdRelease (thread_t ∗tp)

  Releases a reference to a thread object.
- void chThdExit (msg_t msg)

Terminates the current thread.

- void chThdExitS (msg_t msg)

  Terminates the current thread.

- msg_t chThdWait (thread_t *tp)

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

- tprio_t chThdSetPriority (tprio_t newprio)

  Changes the running thread priority level then reschedules if necessary.

- void chThdTerminate (thread_t *tp)

  Requests a thread termination.

- void chThdSleep (sysinterval_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.

- systime_t chThdSleepUntilWindowed (systime_t prev, systime_t next)

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

- void chThdYield (void)

  Yields the time slot.

- msg_t chThdSuspendS (thread_reference_t *trp)

  Sends the current thread sleeping and sets a reference variable.

- msg_t chThdSuspendTimeoutS (thread_reference_t *trp, sysinterval_t timeout)

  Sends the current thread sleeping and sets a reference variable.

- void chThdResumeI (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.

- void chThdResumeS (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.

- void chThdResume (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.

- msg_t chThdEnqueueTimeoutS (threads_queue_t *tqp, sysinterval_t timeout)

  Enqueues the caller thread on a threads queue object.

- void chThdDequeueNextI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up one thread from the threads queue object, if any.

- void chThdDequeueAllI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up all threads from the threads queue object.

10.41.1 Detailed Description

Threads code.

10.42 chthreads.h File Reference

Threads module macros and structures.

Data Structures

- struct thread_descriptor_t

  Type of a thread descriptor.

Macros

Threads queues

- #define _THREADS_QUEUE_DATA(name) {(thread_t *)&name, (thread_t *)&name}
  Data part of a static threads queue object initializer.
- #define _THREADS_QUEUE_DECL(name) threads_queue_t name = _THREADS_QUEUE_DATA(name)
  Static threads queue object initializer.

Working Areas

- #define THD_WORKING_AREA_SIZE(n) MEM_ALIGN_NEXT(sizeof(thread_t) + PORT_WA_SIZE(n), PORT_STACK_ALIGN)
  Calculates the total Working Area size.
- #define THD_WORKING_AREA(s, n) PORT_WORKING_AREA(s, n)
  Static working area allocation.
- #define THD_WORKING_AREA_BASE(s) ((stkalign_t *)(s))
  Base of a working area casted to the correct type.
- #define THD_WORKING_AREA_END(s)
  End of a working area casted to the correct type.

Threads abstraction macros

- #define THD_FUNCTION(tname, arg) PORT_THD_FUNCTION(tname, arg)
  Thread declaration macro.

Macro Functions

- #define chThdSleepSeconds(sec) chThdSleep(TIME_S2I(sec))
  Delays the invoking thread for the specified number of seconds.
- #define chThdSleepMilliseconds(msec) chThdSleep(TIME_MS2I(msec))
  Delays the invoking thread for the specified number of milliseconds.
- #define chThdSleepMicroseconds(usec) chThdSleep(TIME_US2I(usec))
  Delays the invoking thread for the specified number of microseconds.

Typedefs

- typedef void(* tfunc_t) (void *p)
  Thread function.

Functions

- thread_t * _thread_init (thread_t *tp, const char *name, tprio_t prio)
  Initializes a thread structure.
- void _thread_memfill (uint8_t *startp, uint8_t *endp, uint8_t v)
  Memory fill utility.
- thread_t * chThdCreateSuspendedI (const thread_descriptor_t *tdp)
  Creates a new thread into a static memory area.
- thread_t * chThdCreateSuspended (const thread_descriptor_t *tdp)
  Creates a new thread into a static memory area.
- thread_t * chThdCreateI (const thread_descriptor_t *tdp)
  Creates a new thread into a static memory area.
- thread_t * chThdCreate (const thread_descriptor_t *tdp)
  Creates a new thread into a static memory area.
- thread_t * chThdCreateStatic (void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)

Creates a new thread into a static memory area.

- thread_t * chThdStart (thread_t *tp)

  Resumes a thread created with `chThdCreateI()`.
- thread_t * chThdAddRef (thread_t *tp)

  Adds a reference to a thread object.
- void chThdRelease (thread_t *tp)

  Releases a reference to a thread object.
- void chThdExit (msg_t msg)

  Terminates the current thread.
- void chThdExitS (msg_t msg)

  Terminates the current thread.
- msg_t chThdWait (thread_t *tp)

  Blocks the execution of the invoking thread until the specified thread terminates then the exit code is returned.
- tprio_t chThdSetPriority (tprio_t newprio)

  Changes the running thread priority level then reschedules if necessary.
- void chThdTerminate (thread_t *tp)

  Requests a thread termination.
- msg_t chThdSuspendS (thread_reference_t *trp)

  Sends the current thread sleeping and sets a reference variable.
- msg_t chThdSuspendTimeoutS (thread_reference_t *trp, sysinterval_t timeout)

  Sends the current thread sleeping and sets a reference variable.
- void chThdResumeI (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- void chThdResumeS (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- void chThdResume (thread_reference_t *trp, msg_t msg)

  Wakes up a thread waiting on a thread reference object.
- msg_t chThdEnqueueTimeoutS (threads_queue_t *tqp, sysinterval_t timeout)

  Enqueues the caller thread on a threads queue object.
- void chThdDequeueNextI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up one thread from the threads queue object, if any.
- void chThdDequeueAllI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up all threads from the threads queue object.
- void chThdSleep (sysinterval_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.
- systime_t chThdSleepUntilWindowed (systime_t prev, systime_t next)

  Suspends the invoking thread until the system time arrives to the specified value.
- void chThdYield (void)

  Yields the time slot.
- static thread_t * chThdGetSelfX (void)

  Returns a pointer to the current `thread_t`.
- static tprio_t chThdGetPriorityX (void)

  Returns the current thread priority.
- static systime_t chThdGetTicksX (thread_t *tp)

  Returns the number of ticks consumed by the specified thread.
- static stkalign_t * chThdGetWorkingAreaX (thread_t *tp)

  Returns the working area base of the specified thread.
- static bool chThdTerminatedX (thread_t *tp)

  Verifies if the specified thread is in the `CH_STATE_FINAL` state.

- static bool chThdShouldTerminateX (void)

  Verifies if the current thread has a termination request pending.
- static thread_t * chThdStartI (thread_t *tp)

  Resumes a thread created with `chThdCreateI()`.
- static void chThdSleepS (sysinterval_t ticks)

  Suspends the invoking thread for the specified number of ticks.
- static void chThdQueueObjectInit (threads_queue_t *tqp)

  Initializes a threads queue object.
- static bool chThdQueueIsEmptyI (threads_queue_t *tqp)

  Evaluates to `true` if the specified queue is empty.
- static void chThdDoDequeueNextI (threads_queue_t *tqp, msg_t msg)

  Dequeues and wakes up one thread from the threads queue object.

10.42.1 Detailed Description

Threads module macros and structures.

10.43 chtime.h File Reference

Time and intervals macros and structures.

Macros

- #define CH_CFG_ST_RESOLUTION 32

  System time counter resolution.
- #define CH_CFG_ST_FREQUENCY 1000

  System tick frequency.
- #define CH_CFG_INTERVALS_SIZE 32

  Time intervals data size.
- #define CH_CFG_TIME_TYPES_SIZE 32

  Time types data size.

Special time constants

- #define TIME_IMMEDIATE ((sysinterval_t)0)

  Zero interval specification for some functions with a timeout specification.
- #define TIME_INFINITE ((sysinterval_t)-1)

  Infinite interval specification for all functions with a timeout specification.
- #define TIME_MAX_INTERVAL ((sysinterval_t)-2)

  Maximum interval constant usable as timeout.
- #define TIME_MAX_SYSTIME ((systime_t)-1)

  Maximum system of system time before it wraps.

Fast time conversion utilities

- #define TIME_S2I(secs) ((sysinterval_t)((time_conv_t)(secs) * (time_conv_t)CH_CFG_ST_FREQUEN↩CY))

  Seconds to time interval.
- #define TIME_MS2I(msecs)

  Milliseconds to time interval.
- #define TIME_US2I(usecs)

  Microseconds to time interval.

- #define TIME_I2S(interval)
  Time interval to seconds.
- #define TIME_I2MS(interval)
  Time interval to milliseconds.
- #define TIME_I2US(interval)
  Time interval to microseconds.

Typedefs

- typedef uint64_t systime_t
  Type of system time.
- typedef uint64_t sysinterval_t
  Type of time interval.
- typedef uint32_t time_secs_t
  Type of seconds.
- typedef uint32_t time_msecs_t
  Type of milliseconds.
- typedef uint32_t time_usecs_t
  Type of microseconds.
- typedef uint64_t time_conv_t
  Type of time conversion variable.

Functions

Secure time conversion utilities

- static sysinterval_t chTimeS2I (time_secs_t secs)
  Seconds to time interval.
- static sysinterval_t chTimeMS2I (time_msecs_t msec)
  Milliseconds to time interval.
- static sysinterval_t chTimeUS2I (time_usecs_t usec)
  Microseconds to time interval.
- static time_secs_t chTimeI2S (sysinterval_t interval)
  Time interval to seconds.
- static time_msecs_t chTimeI2MS (sysinterval_t interval)
  Time interval to milliseconds.
- static time_usecs_t chTimeI2US (sysinterval_t interval)
  Time interval to microseconds.
- static systime_t chTimeAddX (systime_t systime, sysinterval_t interval)
  Adds an interval to a system time returning a system time.
- static sysinterval_t chTimeDiffX (systime_t start, systime_t end)
  Subtracts two system times returning an interval.
- static bool chTimeIsInRangeX (systime_t time, systime_t start, systime_t end)
  Checks if the specified time is within the specified time range.

10.43.1 Detailed Description

Time and intervals macros and structures.

10.44 chtm.c File Reference

Time Measurement module code.

```
#include "ch.h"
```

Functions

- void _tm_init (void)

  Initializes the time measurement unit.
- void chTMObjectInit (time_measurement_t *tmp)

  Initializes a `TimeMeasurement` object.
- NOINLINE void chTMStartMeasurementX (time_measurement_t *tmp)

  Starts a measurement.
- NOINLINE void chTMStopMeasurementX (time_measurement_t *tmp)

  Stops a measurement.
- NOINLINE void chTMChainMeasurementToX (time_measurement_t *tmp1, time_measurement_t *tmp2)

  Stops a measurement and chains to the next one using the same time stamp.

10.44.1 Detailed Description

Time Measurement module code.

10.45 chtm.h File Reference

Time Measurement module macros and structures.

Data Structures

- struct tm_calibration_t

  Type of a time measurement calibration data.
- struct time_measurement_t

  Type of a Time Measurement object.

Functions

- void _tm_init (void)

  Initializes the time measurement unit.
- void chTMObjectInit (time_measurement_t *tmp)

  Initializes a `TimeMeasurement` object.
- NOINLINE void chTMStartMeasurementX (time_measurement_t *tmp)

  Starts a measurement.
- NOINLINE void chTMStopMeasurementX (time_measurement_t *tmp)

  Stops a measurement.
- NOINLINE void chTMChainMeasurementToX (time_measurement_t *tmp1, time_measurement_t *tmp2)

  Stops a measurement and chains to the next one using the same time stamp.

10.45.1 Detailed Description

Time Measurement module macros and structures.

10.46 chtrace.c File Reference

Tracer code.

```
#include "ch.h"
```

Functions

- static NOINLINE void trace_next (void)

  Writes a time stamp and increases the trace buffer pointer.
- void _trace_init (void)

  Trace circular buffer subsystem initialization.
- void _trace_switch (thread_t *ntp, thread_t *otp)

  Inserts in the circular debug trace buffer a context switch record.
- void _trace_isr_enter (const char *isr)

  Inserts in the circular debug trace buffer an ISR-enter record.
- void _trace_isr_leave (const char *isr)

  Inserts in the circular debug trace buffer an ISR-leave record.
- void _trace_halt (const char *reason)

  Inserts in the circular debug trace buffer an halt record.
- void chDbgWriteTraceI (void *up1, void *up2)

  Adds an user trace record to the trace buffer.
- void chDbgWriteTrace (void *up1, void *up2)

  Adds an user trace record to the trace buffer.
- void chDbgSuspendTraceI (uint16_t mask)

  Suspends one or more trace events.
- void chDbgSuspendTrace (uint16_t mask)

  Suspends one or more trace events.
- void chDbgResumeTraceI (uint16_t mask)

  Resumes one or more trace events.
- void chDbgResumeTrace (uint16_t mask)

  Resumes one or more trace events.

10.46.1 Detailed Description

Tracer code.

10.47 chtrace.h File Reference

Tracer macros and structures.

Data Structures

- struct ch_trace_event_t

  Trace buffer record.
- struct ch_trace_buffer_t

  Trace buffer header.

Macros

Trace record types

- #define **CH_TRACE_TYPE_UNUSED** 0U
- #define **CH_TRACE_TYPE_SWITCH** 1U
- #define **CH_TRACE_TYPE_ISR_ENTER** 2U
- #define **CH_TRACE_TYPE_ISR_LEAVE** 3U
- #define **CH_TRACE_TYPE_HALT** 4U
- #define **CH_TRACE_TYPE_USER** 5U

Events to trace

- #define **CH_DBG_TRACE_MASK_DISABLED** 255U
- #define **CH_DBG_TRACE_MASK_NONE** 0U
- #define **CH_DBG_TRACE_MASK_SWITCH** 1U
- #define **CH_DBG_TRACE_MASK_ISR** 2U
- #define **CH_DBG_TRACE_MASK_HALT** 4U
- #define **CH_DBG_TRACE_MASK_USER** 8U
- #define **CH_DBG_TRACE_MASK_SLOW**
- #define **CH_DBG_TRACE_MASK_ALL**

Debug related settings

- #define CH_DBG_TRACE_MASK CH_DBG_TRACE_MASK_DISABLED
  Trace buffer entries.
- #define CH_DBG_TRACE_BUFFER_SIZE 128
  Trace buffer entries.

Functions

- void _trace_switch (thread_t *ntp, thread_t *otp)
  Inserts in the circular debug trace buffer a context switch record.
- void _trace_isr_enter (const char *isr)
  Inserts in the circular debug trace buffer an ISR-enter record.
- void _trace_isr_leave (const char *isr)
  Inserts in the circular debug trace buffer an ISR-leave record.
- void _trace_halt (const char *reason)
  Inserts in the circular debug trace buffer an halt record.
- void chDbgWriteTraceI (void *up1, void *up2)
  Adds an user trace record to the trace buffer.
- void chDbgWriteTrace (void *up1, void *up2)
  Adds an user trace record to the trace buffer.
- void chDbgSuspendTraceI (uint16_t mask)
  Suspends one or more trace events.
- void chDbgSuspendTrace (uint16_t mask)
  Suspends one or more trace events.
- void chDbgResumeTraceI (uint16_t mask)
  Resumes one or more trace events.
- void chDbgResumeTrace (uint16_t mask)
  Resumes one or more trace events.

10.47.1 Detailed Description

Tracer macros and structures.

10.48 chvt.c File Reference

Time and Virtual Timers module code.

```
#include "ch.h"
```

Functions

- void _vt_init (void)
  Virtual Timers initialization.
- void chVTDoSetI (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)
  Enables a virtual timer.
- void chVTDoResetI (virtual_timer_t *vtp)
  Disables a Virtual Timer.

10.48.1 Detailed Description

Time and Virtual Timers module code.

10.49 chvt.h File Reference

Time and Virtual Timers module macros and structures.

Functions

- void _vt_init (void)
  Virtual Timers initialization.
- void chVTDoSetI (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)
  Enables a virtual timer.
- void chVTDoResetI (virtual_timer_t *vtp)
  Disables a Virtual Timer.
- static void chVTObjectInit (virtual_timer_t *vtp)
  Initializes a `virtual_timer_t` object.
- static systime_t chVTGetSystemTimeX (void)
  Current system time.
- static systime_t chVTGetSystemTime (void)
  Current system time.
- static sysinterval_t chVTTimeElapsedSinceX (systime_t start)
  Returns the elapsed time since the specified start time.
- static bool chVTIsSystemTimeWithinX (systime_t start, systime_t end)
  Checks if the current system time is within the specified time window.
- static bool chVTIsSystemTimeWithin (systime_t start, systime_t end)
  Checks if the current system time is within the specified time window.
- static bool chVTGetTimersStateI (sysinterval_t *timep)
  Returns the time interval until the next timer event.
- static bool chVTIsArmedI (const virtual_timer_t *vtp)
  Returns `true` if the specified timer is armed.
- static bool chVTIsArmed (const virtual_timer_t *vtp)
  Returns `true` if the specified timer is armed.

- static void chVTResetI (virtual_timer_t *vtp)

  Disables a Virtual Timer.
- static void chVTReset (virtual_timer_t *vtp)

  Disables a Virtual Timer.
- static void chVTSetI (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)

  Enables a virtual timer.
- static void chVTSet (virtual_timer_t *vtp, sysinterval_t delay, vtfunc_t vtfunc, void *par)

  Enables a virtual timer.
- static void chVTDoTickI (void)

  Virtual timers ticker.

10.49.1 Detailed Description

Time and Virtual Timers module macros and structures.

Index