



Helios
Kernel 0.3.5

Helios Developer's Guide

1 Data Structure Index	1
1.1 Data Structures	1
2 File Index	2
2.1 File List	2
3 Data Structure Documentation	2
3.1 MemoryRegionStats_s Struct Reference	2
3.1.1 Detailed Description	3
3.1.2 Field Documentation	3
3.2 QueueMessage_s Struct Reference	4
3.2.1 Detailed Description	4
3.2.2 Field Documentation	4
3.3 SystemInfo_s Struct Reference	4
3.3.1 Detailed Description	5
3.3.2 Field Documentation	5
3.4 TaskInfo_s Struct Reference	6
3.4.1 Detailed Description	6
3.4.2 Field Documentation	6
3.5 TaskNotification_s Struct Reference	7
3.5.1 Detailed Description	7
3.5.2 Field Documentation	8
3.6 TaskRunTimeStats_s Struct Reference	8
3.6.1 Detailed Description	8
3.6.2 Field Documentation	8
4 File Documentation	9
4.1 config.h File Reference	9
4.1.1 Detailed Description	10
4.1.2 Macro Definition Documentation	10
4.2 HeliOS.h File Reference	13
4.2.1 Detailed Description	18
4.2.2 Macro Definition Documentation	18
4.2.3 Typedef Documentation	19
4.2.4 Enumeration Type Documentation	29
4.2.5 Function Documentation	30
Index	59

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

MemoryRegionStats_s	
Data structure for statistics on a memory region	2
QueueMessage_s	
Data structure for a message queue message	4
SystemInfo_s	
Data structure for system informaiton	4
TaskInfo_s	
Data structure for information about a task	6
TaskNotification_s	
Data structure for direct to task notifications	7
TaskRunTimeStats_s	
Data structure for task runtime statistics	8

2 File Index

2.1 File List

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

config.h	
Kernel header file for user definable settings	9
HeliOS.h	
Header file for end-user application code	13

3 Data Structure Documentation

3.1 MemoryRegionStats_s Struct Reference

Data structure for statistics on a memory region.

Data Fields

- [Word_t largestFreeEntryInBytes](#)
- [Word_t smallestFreeEntryInBytes](#)
- [Word_t numberOfFreeBlocks](#)
- [Word_t availableSpaceInBytes](#)
- [Word_t successfulAllocations](#)
- [Word_t successfulFrees](#)
- [Word_t minimumEverFreeBytesRemaining](#)

3.1.1 Detailed Description

The MemoryRegionStats_t data structure is used to store statistics about a HeliOS memory region. Statistics can be obtained for the heap and kernel memory regions.

See also

[xMemGetHeapStats\(\)](#)
[xMemGetKernelStats\(\)](#)
[xMemoryRegionStats](#)

3.1.2 Field Documentation

3.1.2.1 availableSpaceInBytes [Word_t](#) MemoryRegionStats_s::availableSpaceInBytes

Amount of free memory in bytes (i.e., numberOfFreeBlocks * CONFIG_MEMORY_REGION_BLOCK_SIZE).

3.1.2.2 largestFreeEntryInBytes [Word_t](#) MemoryRegionStats_s::largestFreeEntryInBytes

The largest free entry in bytes.

3.1.2.3 minimumEverFreeBytesRemaining [Word_t](#) MemoryRegionStats_s::minimumEverFreeBytesRemaining

Lowest water level since system initialization of free bytes of memory.

3.1.2.4 numberOfFreeBlocks [Word_t](#) MemoryRegionStats_s::numberOfFreeBlocks

Number of free blocks - see CONFIG_MEMORY_REGION_BLOCK_SIZE for block size in bytes.

3.1.2.5 smallestFreeEntryInBytes [Word_t](#) MemoryRegionStats_s::smallestFreeEntryInBytes

The smallest free entry in bytes.

3.1.2.6 successfulAllocations [Word_t](#) MemoryRegionStats_s::successfulAllocations

Number of successful memory allocations.

3.1.2.7 successfulFrees [Word_t](#) MemoryRegionStats_s::successfulFrees

Number of successful memory "frees".

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

- [HeliOS.h](#)

3.2 QueueMessage_s Struct Reference

Data structure for a message queue message.

Data Fields

- [Base_t](#) `messageBytes`
- `char` `messageValue` [[CONFIG_MESSAGE_VALUE_BYTES](#)]

3.2.1 Detailed Description

The `QueueMessage_t` data structure contains the message queue message returned by [xQueuePeek\(\)](#) and [xQueueReceive\(\)](#). The `QueueMessage_t` type should be declared as `xQueueMessage`.

See also

[xQueueMessage](#)
[xQueuePeek\(\)](#)
[xQueueReceive\(\)](#)
[xMemFree\(\)](#)
[CONFIG_MESSAGE_VALUE_BYTES](#)

Warning

The memory allocated for an instance of `xQueueMessage` must be freed using [xMemFree\(\)](#).

3.2.2 Field Documentation

3.2.2.1 `messageBytes` [Base_t](#) `QueueMessage_s::messageBytes`

The number of bytes in the `messageValue` member that makes up the message value. This cannot exceed `CONFIG_MESSAGE_VALUE_BYTES`.

3.2.2.2 `messageValue` `char` `QueueMessage_s::messageValue` [[CONFIG_MESSAGE_VALUE_BYTES](#)]

the `char` array that contains the actual message value. This is NOT a null terminated string.

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

- [HeliOS.h](#)

3.3 SystemInfo_s Struct Reference

Data structure for system information.

Data Fields

- char [productName](#) [OS_PRODUCT_NAME_SIZE]
- [Base_t](#) [majorVersion](#)
- [Base_t](#) [minorVersion](#)
- [Base_t](#) [patchVersion](#)
- [Base_t](#) [numberOfTasks](#)

3.3.1 Detailed Description

The [SystemInfo_t](#) data structure contains information about the HeliOS system and is returned by [xSystemGetSystemInfo\(\)](#). The [SystemInfo_t](#) type should be declared as [xSystemInfo](#).

See also

[xSystemInfo](#)
[xSystemGetSystemInfo\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of [xSystemInfo](#) must be freed using [xMemFree\(\)](#).

3.3.2 Field Documentation

3.3.2.1 **majorVersion** [Base_t](#) [SystemInfo_s::majorVersion](#)

The major version number of HeliOS and is Symantec Versioning Specification (SemVer) compliant.

3.3.2.2 **minorVersion** [Base_t](#) [SystemInfo_s::minorVersion](#)

The minor version number of HeliOS and is Symantec Versioning Specification (SemVer) compliant.

3.3.2.3 **numberOfTasks** [Base_t](#) [SystemInfo_s::numberOfTasks](#)

The number of tasks presently in a suspended, running or waiting state.

3.3.2.4 **patchVersion** [Base_t](#) [SystemInfo_s::patchVersion](#)

The patch version number of HeliOS and is Symantec Versioning Specification (SemVer) compliant.

3.3.2.5 **productName** `char SystemInfo_s::productName[OS_PRODUCT_NAME_SIZE]`

The name of the operating system or product. Its length is defined by `OS_PRODUCT_NAME_SIZE`. This is NOT a null terminated string.

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

- [HeliOS.h](#)

3.4 **TaskInfo_s Struct Reference**

Data structure for information about a task.

Data Fields

- [Base_t id](#)
- `char name` [[CONFIG_TASK_NAME_BYTES](#)]
- [TaskState_t state](#)
- [Ticks_t lastRunTime](#)
- [Ticks_t totalRunTime](#)

3.4.1 Detailed Description

The `TaskInfo_t` structure is similar to `xTaskRuntimeStats_t` in that it contains runtime statistics for a task. However, `TaskInfo_t` also contains additional details about a task such as its identifier, ASCII name and state. The `TaskInfo_t` structure is returned by [xTaskGetTaskInfo\(\)](#). If only runtime statistics are needed, `TaskRunTimeStats_t` should be used because of its lower memory footprint. The `TaskInfo_t` type should be declared as `xTaskInfo`.

See also

[xTaskInfo](#)
[xTaskGetTaskInfo\(\)](#)
[xMemFree\(\)](#)
[CONFIG_TASK_NAME_BYTES](#)

Warning

The memory allocated for an instance of `xTaskInfo` must be freed using [xMemFree\(\)](#).

3.4.2 Field Documentation

3.4.2.1 **id** `Base_t TaskInfo_s::id`

The task identifier which is used by [xTaskGetHandleById\(\)](#) to return the task handle.

3.4.2.2 lastRunTime `Ticks_t TaskInfo_s::lastRunTime`

The runtime duration in ticks the last time the task was executed by the scheduler.

3.4.2.3 name `char TaskInfo_s::name[CONFIG_TASK_NAME_BYTES]`

The name of the task which is used by `xTaskGetHandleByName()` to return the task handle. This is NOT a null terminated string.

3.4.2.4 state `TaskState_t TaskInfo_s::state`

The state the task is in which is one of four states specified in the `TaskState_t` enumerated data type.

3.4.2.5 totalRunTime `Ticks_t TaskInfo_s::totalRunTime`

The total runtime duration in ticks the task has been executed by the scheduler.

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

- [HeliOS.h](#)

3.5 TaskNotification_s Struct Reference

Data structure for direct to task notifications.

Data Fields

- `Base_t notificationBytes`
- `char notificationValue[CONFIG_NOTIFICATION_VALUE_BYTES]`

3.5.1 Detailed Description

The `TaskNotification_t` data structure contains the direct to task notification returned by `xTaskNotifyTake()`. The `TaskNotification_t` type should be declared as `xTaskNotification`.

See also

[xTaskNotification](#)
[xTaskNotifyTake\(\)](#)
[xMemFree\(\)](#)
[CONFIG_NOTIFICATION_VALUE_BYTES](#)

Warning

The memory allocated for an instance of `xTaskNotification` must be freed using `xMemFree()`.

3.5.2 Field Documentation

3.5.2.1 notificationBytes `Base_t TaskNotification_s::notificationBytes`

The number of bytes in the notificationValue member that makes up the notification value. This cannot exceed CONFIG_NOTIFICATION_VALUE_BYTES.

3.5.2.2 notificationValue `char TaskNotification_s::notificationValue[CONFIG_NOTIFICATION_VALUE_BYTES]`

The char array that contains the actual notification value. This is NOT a null terminated string.

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

- [HeliOS.h](#)

3.6 TaskRunTimeStats_s Struct Reference

Data structure for task runtime statistics.

Data Fields

- [Base_t id](#)
- [Ticks_t lastRunTime](#)
- [Ticks_t totalRunTime](#)

3.6.1 Detailed Description

The TaskRunTimeStats_t structure contains task runtime statistics and is returned by [xTaskGetAllRunTimeStats\(\)](#) and [xTaskGetTaskRunTimeStats\(\)](#). The TaskRunTimeStats_t type should be declared as xTaskRunTimeStats.

See also

[xTaskRunTimeStats](#)
[xTaskGetTaskRunTimeStats\(\)](#)
[xTaskGetAllRunTimeStats\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of xTaskRunTimeStats must be freed using [xMemFree\(\)](#).

3.6.2 Field Documentation

3.6.2.1 id `Base_t TaskRunTimeStats_s::id`

The task identifier which is used by `xTaskGetHandleByld()` to return the task handle.

3.6.2.2 lastRunTime `Ticks_t TaskRunTimeStats_s::lastRunTime`

The runtime duration in ticks the last time the task was executed by the scheduler.

3.6.2.3 totalRunTime `Ticks_t TaskRunTimeStats_s::totalRunTime`

The total runtime duration in ticks the task has been executed by the scheduler.

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

- [HeliOS.h](#)

4 File Documentation

4.1 config.h File Reference

Kernel header file for user definable settings.

Macros

- `#define CONFIG_MESSAGE_VALUE_BYTES 0x8u /* 8 */`
Define to enable the Arduino API C++ interface.
- `#define CONFIG_NOTIFICATION_VALUE_BYTES 0x8u /* 8 */`
Define the size in bytes of the direct to task notification value.
- `#define CONFIG_TASK_NAME_BYTES 0x8u /* 8 */`
Define the size in bytes of the ASCII task name.
- `#define CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS 0x18u /* 24 */`
Define the number of memory blocks available in all memory regions.
- `#define CONFIG_MEMORY_REGION_BLOCK_SIZE 0x20u /* 32 */`
Define the memory block size in bytes for all memory regions.
- `#define CONFIG_QUEUE_MINIMUM_LIMIT 0x5u /* 5 */`
Define the minimum value for a message queue limit.
- `#define CONFIG_STREAM_BUFFER_BYTES 0x20u /* 32 */`
Define the length of the stream buffer.
- `#define CONFIG_TASK_WD_TIMER_ENABLE`
Enable task watchdog timers.
- `#define CONFIG_DEVICE_NAME_BYTES 0x8u /* 8 */`

4.1.1 Detailed Description

Author

Manny Peterson (mannymsp@gmail.com)

Version

0.3.5

Date

2022-01-31

Copyright

Helios Embedded Operating System Copyright (C) 2020-2022 Manny Peterson mannymsp@gmail.com

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

4.1.2 Macro Definition Documentation

4.1.2.1 CONFIG_MEMORY_REGION_BLOCK_SIZE `#define CONFIG_MEMORY_REGION_BLOCK_SIZE 0x20u`
`/* 32 */`

Setting CONFIG_MEMORY_REGION_BLOCK_SIZE allows the end-user to define the size of a memory region block in bytes. The memory region block size should be set to achieve the best possible utilization of the available memory. The CONFIG_MEMORY_REGION_BLOCK_SIZE setting effects both the heap and kernel memory regions. The default value is 32 bytes. The literal must be appended with a "u" to maintain MISRA C:2012 compliance.

See also

[xMemAlloc\(\)](#)

[xMemFree\(\)](#)

[CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS](#)

4.1.2.2 CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS `#define CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS 0x18u /* 24 */`

The heap memory region is used by tasks. Whereas the kernel memory region is used solely by the kernel for kernel objects. The CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS setting allows the end-user to define the size, in blocks, of all memory regions thus effecting both the heap and kernel memory regions. The size of a memory block is defined by the CONFIG_MEMORY_REGION_BLOCK_SIZE setting. The size of all memory regions needs to be adjusted to fit the memory requirements of the end-user's application. By default the CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS is defined on a per platform and/or tool-chain basis therefor it is not defined here by default. The literal must be appended with a "u" to maintain MISRA C:2012 compliance.

4.1.2.3 CONFIG_MESSAGE_VALUE_BYTES `#define CONFIG_MESSAGE_VALUE_BYTES 0x8u /* 8 */`

Because HeliOS kernel is written in C, the Arduino API cannot be called directly from the kernel. For example, assertions are unable to be written to the serial bus in applications using the Arduino platform/tool-chain. The CONFIG_ENABLE_ARDUINO_CPP_INTERFACE builds the included arduino.cpp file to allow the kernel to call the Arduino API through wrapper functions such as **ArduinoAssert()**. The arduino.cpp file can be found in the /extras directory. It must be copied into the /src directory to be built.

Note

On some MCU's like the 8-bit AVR's, it is necessary to undefine the DISABLE_INTERRUPTS() macro because interrupts must be enabled to write to the serial bus.

Define to enable system assertions.

The CONFIG_ENABLE_SYSTEM_ASSERT setting allows the end-user to enable system assertions in HeliOS. Once enabled, the end-user must define CONFIG_SYSTEM_ASSERT_BEHAVIOR for there to be an effect. By default the CONFIG_ENABLE_SYSTEM_ASSERT setting is not defined.

See also

CONFIG_SYSTEM_ASSERT_BEHAVIOR

Define the system assertion behavior.

The CONFIG_SYSTEM_ASSERT_BEHAVIOR setting allows the end-user to specify the behavior (code) of the assertion which is called when CONFIG_ENABLE_SYSTEM_ASSERT is defined. Typically some sort of output is generated over a serial or other interface. By default the CONFIG_SYSTEM_ASSERT_BEHAVIOR is not defined.

Note

In order to use the **ArduinoAssert()** functionality, the CONFIG_ENABLE_ARDUINO_CPP_INTERFACE setting must be enabled.

See also

CONFIG_ENABLE_SYSTEM_ASSERT

CONFIG_ENABLE_ARDUINO_CPP_INTERFACE

```
#define CONFIG_SYSTEM_ASSERT_BEHAVIOR(f, l) __ArduinoAssert__( f , l )
```

Define the size in bytes of the message queue message value.

Setting the CONFIG_MESSAGE_VALUE_BYTES allows the end-user to define the size of the message queue message value. The larger the size of the message value, the greater impact there will be on system performance. The default size is 8 bytes. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

[xQueueMessage](#)

4.1.2.4 CONFIG_NOTIFICATION_VALUE_BYTES `#define CONFIG_NOTIFICATION_VALUE_BYTES 0x8u /* 8 */`

Setting the CONFIG_NOTIFICATION_VALUE_BYTES allows the end-user to define the size of the direct to task notification value. The larger the size of the notification value, the greater impact there will be on system performance. The default size is 8 bytes. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

[xTaskNotification](#)

4.1.2.5 CONFIG_QUEUE_MINIMUM_LIMIT `#define CONFIG_QUEUE_MINIMUM_LIMIT 0x5u /* 5 */`

Setting the CONFIG_QUEUE_MINIMUM_LIMIT allows the end-user to define the MINIMUM length limit a message queue can be created with [xQueueCreate\(\)](#). When a message queue length equals its limit, the message queue will be considered full and return true when [xQueuesQueueFull\(\)](#) is called. A full queue will also not accept messages from [xQueueSend\(\)](#). The default value is 5. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

[xQueuesQueueFull\(\)](#)

[xQueueSend\(\)](#)

[xQueueCreate\(\)](#)

4.1.2.6 CONFIG_STREAM_BUFFER_BYTES `#define CONFIG_STREAM_BUFFER_BYTES 0x20u /* 32 */`

Setting CONFIG_STREAM_BUFFER_BYTES will define the length of stream buffers created by [xStreamCreate\(\)](#). When the length of the stream buffer reaches this value, it is considered full and can no longer be written to by calling [xStreamSend\(\)](#). The default value is 32. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

4.1.2.7 CONFIG_TASK_NAME_BYTES `#define CONFIG_TASK_NAME_BYTES 0x8u /* 8 */`

Setting the CONFIG_TASK_NAME_BYTES allows the end-user to define the size of the ASCII task name. The larger the size of the task name, the greater impact there will be on system performance. The default size is 8 bytes. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

[xTaskInfo](#)

4.1.2.8 CONFIG_TASK_WD_TIMER_ENABLE `#define CONFIG_TASK_WD_TIMER_ENABLE`

Defining CONFIG_TASK_WD_TIMER_ENABLE will enable the task watchdog timer feature. The default is enabled.

4.2 HeliOS.h File Reference

Header file for end-user application code.

Data Structures

- struct [TaskRunTimeStats_s](#)
Data structure for task runtime statistics.
- struct [TaskInfo_s](#)
Data structure for information about a task.
- struct [TaskNotification_s](#)
Data structure for direct to task notifications.
- struct [QueueMessage_s](#)
Data structure for a message queue message.
- struct [SystemInfo_s](#)
Data structure for system information.
- struct [MemoryRegionStats_s](#)
Data structure for statistics on a memory region.

Macros

- #define [DEREF_TASKPARAM](#)(t, p) *((t *)p)
A C macro to simplify casting and dereferencing a task parameter.

Typedefs

- typedef uint8_t [Base_t](#)
Type definition for the base data type.
- typedef uint32_t [Word_t](#)
Type definition for the word data type.
- typedef uint8_t [Byte_t](#)
Type definition for the byte data type.
- typedef uint16_t [HWord_t](#)
Type definition for the half-word data type.
- typedef uint32_t [Ticks_t](#)
The type definition for time expressed in ticks.
- typedef size_t [Size_t](#)
The type definition for storing the size of some object in memory.
- typedef [Size_t](#) [xSize](#)
The type definition for storing the size of some object in memory.
- typedef struct [TaskRunTimeStats_s](#) [TaskRunTimeStats_t](#)
Data structure for task runtime statistics.
- typedef struct [TaskInfo_s](#) [TaskInfo_t](#)
Data structure for information about a task.
- typedef struct [TaskNotification_s](#) [TaskNotification_t](#)
Data structure for direct to task notifications.
- typedef struct [QueueMessage_s](#) [QueueMessage_t](#)
Data structure for a message queue message.
- typedef struct [SystemInfo_s](#) [SystemInfo_t](#)

- Data structure for system information.*

 - typedef struct [MemoryRegionStats_s](#) [MemoryRegionStats_t](#)

Data structure for statistics on a memory region.
- typedef void [Task_t](#)

Stub type definition for the task type.
- typedef void [StreamBuffer_t](#)

Stub type definition for a stream buffer type.
- typedef void [TaskParm_t](#)

Type definition for the task parameter.
- typedef void [Queue_t](#)

Stub type definition for the message queue type.
- typedef void [Timer_t](#)

Stub type definition for the timer type.
- typedef void [Addr_t](#)

Type definition for the memory address data type.
- typedef [Addr_t](#) * [xAddr](#)

Type definition for the memory address data type.
- typedef [Base_t](#) [xBase](#)

Type definition for the base data type.
- typedef [Word_t](#) [xWord](#)

Type definition for the word data type.
- typedef [HWord_t](#) [xHWord](#)

Type definition for the half-word data type.
- typedef [Timer_t](#) * [xTimer](#)

Stub type definition for the timer type.
- typedef [Queue_t](#) * [xQueue](#)

Stub type definition for the message queue type.
- typedef [QueueMessage_t](#) * [xQueueMessage](#)

Data structure for a message queue message.
- typedef [TaskNotification_t](#) * [xTaskNotification](#)

Data structure for direct to task notifications.
- typedef [TaskInfo_t](#) * [xTaskInfo](#)

Data structure for information about a task.
- typedef [TaskRunTimeStats_t](#) * [xTaskRunTimeStats](#)

Data structure for task runtime statistics.
- typedef [MemoryRegionStats_t](#) * [xMemoryRegionStats](#)

Data structure for statistics on a memory region.
- typedef [Task_t](#) * [xTask](#)

Stub type definition for the task type.
- typedef [StreamBuffer_t](#) * [xStreamBuffer](#)

Stub type definition for a stream buffer type.
- typedef [TaskParm_t](#) * [xTaskParm](#)

Type definition for the task parameter.
- typedef [Ticks_t](#) [xTicks](#)

The type definition for time expressed in ticks.
- typedef [Byte_t](#) [xByte](#)

Type definition for the byte data type.
- typedef [TaskState_t](#) [xTaskState](#)

Enumerated type for task states.
- typedef [SchedulerState_t](#) [xSchedulerState](#)

Enumerated type for scheduler states.
- typedef [SystemInfo_t](#) * [xSystemInfo](#)

Data structure for system information.

Enumerations

- enum `TaskState_t` { `TaskStateError` , `TaskStateSuspended` , `TaskStateRunning` , `TaskStateWaiting` }
Enumerated type for task states.
- enum `SchedulerState_t` { `SchedulerStateError` , `SchedulerStateSuspended` , `SchedulerStateRunning` }
Enumerated type for scheduler states.

Functions

- void `xSystemInit` (void)
System call to initialize the system.
- void `__SystemAssert__` (const char *file_, int line_)
System call to handle assertions.
- `xAddr` `xMemAlloc` (const `xSize` size_)
System call to allocate memory from the heap.
- void `xMemFree` (const `xAddr` addr_)
System call to free memory allocated from the heap.
- `xSize` `xMemGetUsed` (void)
System call to return the amount of allocated heap memory.
- `xSize` `xMemGetSize` (const `xAddr` addr_)
System call to return the amount of heap memory allocated for a given address.
- `xMemoryRegionStats` `xMemGetHeapStats` (void)
System call to obtain statistics on the heap.
- `xMemoryRegionStats` `xMemGetKernelStats` (void)
System call to obtain statistics on the kernel memory region.
- `xQueue` `xQueueCreate` (`xBase` limit_)
System call to create a new message queue.
- void `xQueueDelete` (`xQueue` queue_)
System call to delete a message queue.
- `xBase` `xQueueGetLength` (`xQueue` queue_)
System call to get the length of the message queue.
- `xBase` `xQueueIsQueueEmpty` (`xQueue` queue_)
System call to check if the message queue is empty.
- `xBase` `xQueueIsQueueFull` (`xQueue` queue_)
System call to check if the message queue is full.
- `xBase` `xQueueMessagesWaiting` (`xQueue` queue_)
System call to check if there are message queue messages waiting.
- `xBase` `xQueueSend` (`xQueue` queue_, `xBase` messageBytes_, const char *messageValue_)
System call to send a message using a message queue.
- `xQueueMessage` `xQueuePeek` (`xQueue` queue_)
System call to peek at the next message in a message queue.
- void `xQueueDropMessage` (`xQueue` queue_)
System call to drop the next message in a message queue.
- `xQueueMessage` `xQueueReceive` (`xQueue` queue_)
System call to receive the next message in the message queue.
- void `xQueueLockQueue` (`xQueue` queue_)
System call to LOCK the message queue.
- void `xQueueUnLockQueue` (`xQueue` queue_)
System call to UNLOCK the message queue.
- void `xTaskStartScheduler` (void)

- System call to pass control to the HeliOS scheduler.*

 - void `xTaskResumeAll` (void)
- System call to set scheduler state to running.*

 - void `xTaskSuspendAll` (void)
- System call to set the scheduler state to suspended.*

 - `xSystemInfo` `xSystemGetSystemInfo` (void)

The `xSystemGetSystemInfo()` system call will return the type `xSystemInfo` containing information about the system including the OS (product) name, its version and how many tasks are currently in the running, suspended or waiting states.
- `xTask` `xTaskCreate` (const char *name_, void(*callback_)(xTask, xTaskParm), xTaskParm taskParameter_)

System call to create a new task.
- void `xTaskDelete` (xTask task_)

System call to delete a task.
- `xTask` `xTaskGetHandleByName` (const char *name_)

System call to get a task's handle by its ASCII name.
- `xTask` `xTaskGetHandleById` (xBase id_)

System call to get a task's handle by its task identifier.
- `xTaskRunTimeStats` `xTaskGetAllRunTimeStats` (xBase *tasks_)

System call to return task runtime statistics for all tasks.
- `xTaskRunTimeStats` `xTaskGetTaskRunTimeStats` (xTask task_)

System call to return task runtime statistics for the specified task.
- `xBase` `xTaskGetNumberOfTasks` (void)

System call to return the number of tasks regardless of their state.
- `xTaskInfo` `xTaskGetTaskInfo` (xTask task_)

System call to return the details of a task.
- `xTaskInfo` `xTaskGetAllTaskInfo` (xBase *tasks_)

System call to return the details of all tasks.
- `xTaskState` `xTaskGetTaskState` (xTask task_)

System call to return the state of a task.
- char * `xTaskGetName` (xTask task_)

System call to return the ASCII name of a task.
- `xBase` `xTaskGetId` (xTask task_)

System call to return the task identifier for a task.
- void `xTaskNotifyStateClear` (xTask task_)

System call to clear a waiting direct to task notification.
- `xBase` `xTaskNotificationIsWaiting` (xTask task_)

System call to check if a direct to task notification is waiting.
- `Base_t` `xTaskNotifyGive` (xTask task_, xBase notificationBytes_, const char *notificationValue_)

System call to give another task a direct to task notification.
- `xTaskNotification` `xTaskNotifyTake` (xTask task_)

System call to take a direct to task notification from another task.
- void `xTaskResume` (xTask task_)

System call to resume a task.
- void `xTaskSuspend` (xTask task_)

System call to suspend a task.
- void `xTaskWait` (xTask task_)

System call to place a task in a waiting state.
- void `xTaskChangePeriod` (xTask task_, xTicks timerPeriod_)

System call to set the task timer period.
- `xTicks` `xTaskGetPeriod` (xTask task_)

System call to get the task timer period.

- void `xTaskResetTimer` (`xTask` task_)
System call to reset the task timer.
- `xSchedulerState` `xTaskGetSchedulerState` (void)
System call to get the state of the scheduler.
- `xTimer` `xTimerCreate` (`xTicks` timerPeriod_)
System call to create a new timer.
- void `xTimerDelete` (`xTimer` timer_)
System call will delete a timer.
- void `xTimerChangePeriod` (`xTimer` timer_, `xTicks` timerPeriod_)
System call to change the period of a timer.
- `xTicks` `xTimerGetPeriod` (`xTimer` timer_)
System call to get the period of a timer.
- `xBase` `xTimerIsTimerActive` (`xTimer` timer_)
System call to check if a timer is active.
- `xBase` `xTimerHasTimerExpired` (`xTimer` timer_)
System call to check if a timer has expired.
- void `xTimerReset` (`xTimer` timer_)
System call to reset a timer.
- void `xTimerStart` (`xTimer` timer_)
System call to start a timer.
- void `xTimerStop` (`xTimer` timer_)
The `xTimerStop()` system call will place the timer in the stopped state. Neither `xTimerStart()` nor `xTimerStop()` will reset the timer. Timers can only be reset with `xTimerReset()`.
- void `xSystemHalt` (void)
The `xSystemHalt()` system call will halt HeliOS.
- void `xTaskChangeWdPeriod` (`xTask` *task_, `xTicks` wdTimerPeriod_)
The `xTaskChangeWdPeriod()` will change the period on the task watchdog timer.
- `xTicks` `xTaskGetWdPeriod` (`xTask` *task_)
The `xTaskGetWdPeriod()` return the current task watchdog timer.
- `xStreamBuffer` `xStreamCreate` (void)
The `xStreamCreate()` system call will create a new stream buffer.
- void `xStreamDelete` (`xStreamBuffer` stream_)
The `xStreamDelete()` system call will delete a stream buffer.
- `xBase` `xStreamSend` (`xStreamBuffer` stream_, `xByte` byte_)
The `xStreamSend()` system call will write one byte to the stream buffer.
- `xByte` * `xStreamReceive` (`xStreamBuffer` stream_, `HWord_t` *bytes_)
The `xStreamReceive()` system call will return the contents of the stream buffer.
- `xHWord` `xStreamBytesAvailable` (`xStreamBuffer` stream_)
The `xStreamBytesAvailable()` system call returns the length of the stream buffer.
- void `xStreamReset` (`xStreamBuffer` stream_)
The `xStreamReset()` system call will reset a stream buffer.
- `xBase` `xStreamIsEmpty` (`xStreamBuffer` stream_)
The `xStreamIsEmpty()` system call returns true if the stream buffer is empty.
- `xBase` `xStreamIsFull` (`xStreamBuffer` steam_)
The `xStreamIsFull()` system call returns true if the stream buffer is full.
- `xBase` `xDeviceRegisterDevice` (`xBase`(*device_self_register_())
- `xBase` `xDeviceWrite` (`xHWord` uid_, `xHWord` *bytes_, `xByte` *data_)
- `xBase` `xDeviceRead` (`xHWord` uid_, `xHWord` *bytes_, `xByte` *data_)

4.2.1 Detailed Description

Author

Manny Peterson (mannymsp@gmail.com)

Version

0.3.5

Date

2022-01-31

Copyright

Helios Embedded Operating System Copyright (C) 2020-2022 Manny Peterson mannymsp@gmail.com

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

4.2.2 Macro Definition Documentation

4.2.2.1 Deref_TaskParm `#define Deref_TaskParm(`

```
    t,  
    p ) *((t *)p)
```

When a task parameter is passed to a task, it is passed as a pointer of type void. To use the parameter, it must first be cast to the correct type and dereferenced. The following is an example of how the `Deref_TaskParm()` C macro simplifies that process.

```
void myTask_main(xTask task_, xTaskParm parm_) {  
    int i;  
    i = Deref_TaskParm(int, parm_);  
    i++;  
    Deref_TaskParm(int, parm_) = i;  
    return;  
}
```

Parameters

<i>t</i>	The data type to cast the task parameter to (e.g., int).
<i>p</i>	The task pointer, typically named parm_.

4.2.3 Typedef Documentation

4.2.3.1 Addr_t typedef void Addr_t

The xAddr type is used to store a memory address and is used to pass memory addresses back and forth between system calls and the end-user application. It is not necessary to use the xAddr type within the end-user application as long as the type is not used to interact with the HeliOS kernel through system calls.

4.2.3.2 Base_t typedef uint8_t Base_t

A simple data type is often needed as an argument for a system call or a return type. The Base_t type is used in such a case where there are no other structural data requirements and is typically an unsigned 8-bit integer. The Base_t type should be declared as xBase.

See also

[xBase](#)

4.2.3.3 Byte_t typedef uint8_t Byte_t

A byte is an 8-bit data type in HeliOS.

See also

[xByte](#)

4.2.3.4 HWord_t typedef uint16_t HWord_t

A half-word is a 16-bit data type in HeliOS.

See also

[xHWord](#)

4.2.3.5 MemoryRegionStats_t typedef struct MemoryRegionStats_s MemoryRegionStats_t

The MemoryRegionStats_t data structure is used to store statistics about a HeliOS memory region. Statistics can be obtained for the heap and kernel memory regions.

See also

[xMemGetHeapStats\(\)](#)

[xMemGetKernelStats\(\)](#)

[xMemoryRegionStats](#)

4.2.3.6 Queue_t `typedef void Queue_t`

The Queue_t type is a stub type definition for the internal message queue structure and is treated as a message queue handle by most of the message queue related system calls. The members of the data structure are not accessible. The Queue_t type should be declared as xQueue.

See also

[xQueue](#)
[xQueueDelete\(\)](#)

Warning

The memory allocated for an instance of xQueue must be freed using [xQueueDelete\(\)](#).

4.2.3.7 QueueMessage_t `typedef struct QueueMessage_s QueueMessage_t`

The QueueMessage_t data structure contains the message queue message returned by [xQueuePeek\(\)](#) and [xQueueReceive\(\)](#). The QueueMessage_t type should be declared as xQueueMessage.

See also

[xQueueMessage](#)
[xQueuePeek\(\)](#)
[xQueueReceive\(\)](#)
[xMemFree\(\)](#)
[CONFIG_MESSAGE_VALUE_BYTES](#)

Warning

The memory allocated for an instance of xQueueMessage must be freed using [xMemFree\(\)](#).

4.2.3.8 Size_t `typedef size_t Size_t`

The Size_t type is used to store the size of an object in memory and is always represented in bytes. Size_t should always be declared as xSize.

See also

[xSize](#)

4.2.3.9 **StreamBuffer_t** `typedef void StreamBuffer_t`

The `StreamBuffer_t` type is a stub definition for the internal stream buffer data structure and is treated as a handle by most of the stream system calls. The members of this data structure are not accessible. The `StreamBuffer_t` type should be declared as `xStreamBuffer`.

See also

[xStreamBuffer](#)
[xStreamDelete\(\)](#)

Warning

The memory allocated for an instance of the `xStreamBuffer` must be freed by [xStreamDelete\(\)](#)

4.2.3.10 **SystemInfo_t** `typedef struct SystemInfo_s SystemInfo_t`

The `SystemInfo_t` data structure contains information about the HeliOS system and is returned by [xSystemGetSystemInfo\(\)](#). The `SystemInfo_t` type should be declared as `xSystemInfo`.

See also

[xSystemInfo](#)
[xSystemGetSystemInfo\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of `xSystemInfo` must be freed using [xMemFree\(\)](#).

4.2.3.11 **Task_t** `typedef void Task_t`

The `Task_t` type is a stub type definition for the internal task data structure and is treated as a task handle by most of the task related system calls. The members of the data structure are not accessible. The `Task_t` type should be declared as `xTask`.

See also

[xTask](#)
[xTaskDelete\(\)](#)

Warning

The memory allocated for an instance of `xTask` must be freed by [xTaskDelete\(\)](#)

4.2.3.12 TaskInfo_t `typedef struct TaskInfo_s TaskInfo_t`

The TaskInfo_t structure is similar to xTaskRuntimeStats_t in that it contains runtime statistics for a task. However, TaskInfo_t also contains additional details about a task such as its identifier, ASCII name and state. The TaskInfo_t structure is returned by [xTaskGetTaskInfo\(\)](#). If only runtime statistics are needed, TaskRunTimeStats_t should be used because of its lower memory footprint. The TaskInfo_t type should be declared as xTaskInfo.

See also

[xTaskInfo](#)
[xTaskGetTaskInfo\(\)](#)
[xMemFree\(\)](#)
[CONFIG_TASK_NAME_BYTES](#)

Warning

The memory allocated for an instance of xTaskInfo must be freed using [xMemFree\(\)](#).

4.2.3.13 TaskNotification_t `typedef struct TaskNotification_s TaskNotification_t`

The TaskNotification_t data structure contains the direct to task notification returned by [xTaskNotifyTake\(\)](#). The TaskNotification_t type should be declared as xTaskNotification.

See also

[xTaskNotification](#)
[xTaskNotifyTake\(\)](#)
[xMemFree\(\)](#)
[CONFIG_NOTIFICATION_VALUE_BYTES](#)

Warning

The memory allocated for an instance of xTaskNotification must be freed using [xMemFree\(\)](#).

4.2.3.14 TaskParm_t `typedef void TaskParm_t`

The TaskParm_t type is used to pass a parameter to a task at the time of creation using [xTaskCreate\(\)](#). A task parameter is a pointer of type void and can point to any number of intrinsic types, arrays and/or user defined structures which can be passed to a task. It is up to the end-user to manage, allocate and free the memory related to these objects using [xMemAlloc\(\)](#) and [xMemFree\(\)](#). The TaskParm_t should be declared as xTaskParm.

See also

[xTaskParm](#)
[xMemAlloc\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of xTaskParm must be freed using [xMemFree\(\)](#).

4.2.3.15 TaskRunTimeStats_t `typedef struct TaskRunTimeStats_s TaskRunTimeStats_t`

The TaskRunTimeStats_t structure contains task runtime statistics and is returned by [xTaskGetAllRunTimeStats\(\)](#) and [xTaskGetTaskRunTimeStats\(\)](#). The TaskRunTimeStats_t type should be declared as xTaskRunTimeStats.

See also

[xTaskRunTimeStats](#)
[xTaskGetTaskRunTimeStats\(\)](#)
[xTaskGetAllRunTimeStats\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of xTaskRunTimeStats must be freed using [xMemFree\(\)](#).

4.2.3.16 Ticks_t `typedef uint32_t Ticks_t`

The xTicks type is used by several of the task and timer related system calls to express time. The unit of measure for time is always ticks.

See also

[xTicks](#)

4.2.3.17 Timer_t `typedef void Timer_t`

The Timer_t type is a stub type definition for the internal timer data structure and is treated as a timer handle by most of the timer related system calls. The members of the data structure are not accessible. The Timer_t type should be declared as xTimer.

See also

[xTimer](#)
[xTimerDelete\(\)](#)

Warning

The memory allocated for an instance of xTimer must be freed using [xTimerDelete\(\)](#).

4.2.3.18 Word_t `typedef uint32_t Word_t`

A word is a 32-bit data type in HeliOS.

See also

[xWord](#)

4.2.3.19 xAddr `typedef Addr_t* xAddr`

The xAddr type is used to store a memory address and is used to pass memory addresses back and forth between system calls and the end-user application. It is not necessary to use the xAddr type within the end-user application as long as the type is not used to interact with the HeliOS kernel through system calls.

4.2.3.20 xBase `typedef Base_t xBase`

A simple data type is often needed as an argument for a system call or a return type. The xBase type is used in such a case where there are no other structural data requirements is typically an unsigned 8-bit integer.

See also

[Base_t](#)

4.2.3.21 xByte `typedef Byte_t xByte`

A byte is an 8-bit data type in HeliOS.

See also

[Byte_t](#)

4.2.3.22 xHWord `typedef HWord_t xHWord`

A half-word is a 16-bit data type in HeliOS.

See also

[HWord_t](#)

4.2.3.23 xMemoryRegionStats `typedef MemoryRegionStats_t* xMemoryRegionStats`

The xMemoryRegionStats data structure is used to store statistics about a HeliOS memory region. Statistics can be obtained for the heap and kernel memory regions.

See also

[xMemGetHeapStats\(\)](#)
[xMemGetKernelStats\(\)](#)
[MemoryRegionStats_t](#)

4.2.3.24 xQueue `typedef Queue_t* xQueue`

The xQueue type is a stub type definition for the internal message queue structure and is treated as a message queue handle by most of the message queue related system calls. The members of the data structure are not accessible.

See also

[Queue_t](#)
[xQueueDelete\(\)](#)

Warning

The memory allocated for an instance of xQueue must be freed using [xQueueDelete\(\)](#).

4.2.3.25 xQueueMessage `typedef QueueMessage_t* xQueueMessage`

The xQueueMessage data structure contains the message queue message returned by [xQueuePeek\(\)](#) and [xQueueReceive\(\)](#). See QueueMessage_t for information about the data structure's members.

See also

[QueueMessage_t](#)
[xQueuePeek\(\)](#)
[xQueueReceive\(\)](#)
[xMemFree\(\)](#)
[CONFIG_MESSAGE_VALUE_BYTES](#)

Warning

The memory allocated for an instance of xQueueMessage must be freed using [xMemFree\(\)](#).

4.2.3.26 xSchedulerState `typedef SchedulerState_t xSchedulerState`

The scheduler can be in one of four possible states defined in the SchedulerState_t enumerated type. The state of the scheduler is changed by calling `xTaskSuspendAll()` and `xTaskResumeAll()`. The state can be obtained by calling `xTaskGetSchedulerState()`.

See also

`xSchedulerState`
`xTaskSuspendAll()`
`xTaskResumeAll()`
`xTaskGetSchedulerState()`

4.2.3.27 xSize `typedef Size_t xSize`

The xSize type is used to store the size of an object in memory and is always represented in bytes.

4.2.3.28 xStreamBuffer `typedef StreamBuffer_t* xStreamBuffer`

The xStreamBuffer type is a stub definition for the internal stream buffer data structure and is treated as a handle by most of the stream system calls. The members of this data structure are not accessible.

See also

`StreamBuffer_t`
`xStreamDelete()`

Warning

The memory allocated for an instance of the xStreamBuffer must be freed by `xStreamDelete()`

4.2.3.29 xSystemInfo `typedef SystemInfo_t* xSystemInfo`

The xSystemInfo data structure contains information about the HeliOS system and is returned by `xSystemGetSystemInfo()`. See xSystemInfo_t for information about the data structure's members.

See also

`SystemInfo_t`
`xSystemGetSystemInfo()`
`xMemFree()`

Warning

The memory allocated for an instance of xSystemInfo must be freed using `xMemFree()`.

4.2.3.30 xTask `typedef Task_t* xTask`

The xTask type is a stub type definition for the internal task data structure and is treated as a task handle by most of the task related system calls. The members of the data structure are not accessible.

See also

[Task_t](#)
[xTaskCreate\(\)](#)
[xTaskDelete\(\)](#)

Warning

The memory allocated for an instance of xTask must be freed by [xTaskDelete\(\)](#)

4.2.3.31 xTaskInfo `typedef TaskInfo_t* xTaskInfo`

The xTaskInfo structure is similar to xTaskRunTimeStats in that it contains runtime statistics for a task. However, xTaskInfo also contains additional details about a task such as its identifier, ASCII name and state. The xTaskInfo structure is returned by [xTaskGetTaskInfo\(\)](#). If only runtime statistics are needed, xTaskRunTimeStats should be used because of its lower memory footprint. See TaskInfo_t for information about the data structure's members.

See also

[TaskInfo_t](#)
[xTaskGetTaskInfo\(\)](#)
[xMemFree\(\)](#)
[CONFIG_TASK_NAME_BYTES](#)

Warning

The memory allocated for an instance of xTaskInfo must be freed using [xMemFree\(\)](#).

4.2.3.32 xTaskNotification `typedef TaskNotification_t* xTaskNotification`

The xTaskNotification data structure contains the direct to task notification returned by [xTaskNotifyTake\(\)](#). See TaskNotification_t for information about the data structure's members.

See also

[TaskNotification_t](#)
[xTaskNotifyTake\(\)](#)
[xMemFree\(\)](#)
[CONFIG_NOTIFICATION_VALUE_BYTES](#)

Warning

The memory allocated for an instance of xTaskNotification must be freed using [xMemFree\(\)](#).

4.2.3.33 xTaskParm `typedef TaskParm_t* xTaskParm`

The xTaskParm type is used to pass a parameter to a task at the time of creation using [xTaskCreate\(\)](#). A task parameter is a pointer of type void and can point to any number of intrinsic types, arrays and/or user defined structures which can be passed to a task. It is up to the end-user to manage allocate and free the memory related to these objects using [xMemAlloc\(\)](#) and [xMemFree\(\)](#).

See also

[TaskParm_t](#)
[xMemAlloc\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of xTaskParm must be freed using [xMemFree\(\)](#).

4.2.3.34 xTaskRunTimeStats `typedef TaskRunTimeStats_t* xTaskRunTimeStats`

The xTaskRunTimeStats structure contains task runtime statistics and is returned by [xTaskGetAllRunTimeStats\(\)](#) and [xTaskGetTaskRunTimeStats\(\)](#). See TaskRunTimeStats_t for information about the data structure's members.

See also

[TaskRunTimeStats_t](#)
[xTaskGetTaskRunTimeStats\(\)](#)
[xTaskGetAllRunTimeStats\(\)](#)
[xMemFree\(\)](#)

Warning

The memory allocated for an instance of xTaskRunTimeStats must be freed using [xMemFree\(\)](#).

4.2.3.35 xTaskState `typedef TaskState_t xTaskState`

A task can be in one of the four possible states defined in the TaskState_t enumerated type. The state of a task is changed by calling [xTaskResume\(\)](#), [xTaskSuspend\(\)](#) or [xTaskWait\(\)](#).

See also

[TaskState_t](#)
[xTaskResume\(\)](#)
[xTaskSuspend\(\)](#)
[xTaskWait\(\)](#)

4.2.3.36 xTicks typedef `Ticks_t xTicks`

The xTicks type is used by several of the task and timer related system calls to express time. The unit of measure for time is always ticks.

See also

[Ticks_t](#)

4.2.3.37 xTimer typedef `Timer_t* xTimer`

The xTimer type is a stub type definition for the internal timer data structure and is treated as a timer handle by most of the timer related system calls. The members of the data structure are not accessible.

See also

[Timer_t](#)

[xTimerDelete\(\)](#)

Warning

The memory allocated for an instance of xTimer must be freed using [xTimerDelete\(\)](#).

4.2.3.38 xWord typedef `Word_t xWord`

A word is a 32-bit data type in HeliOS.

See also

[Word_t](#)

4.2.4 Enumeration Type Documentation**4.2.4.1 SchedulerState_t** enum `SchedulerState_t`

The scheduler can be in one of four possible states defined in the SchedulerState_t enumerated type. The state of the scheduler is changed by calling [xTaskSuspendAll\(\)](#) and [xTaskResumeAll\(\)](#). The state can be obtained by calling [xTaskGetSchedulerState\(\)](#).

See also

[xSchedulerState](#)

[xTaskSuspendAll\(\)](#)

[xTaskResumeAll\(\)](#)

Enumerator

SchedulerStateError	Not used.
SchedulerStateSuspended	State the scheduler is in after xTaskSuspendAll() is called.
SchedulerStateRunning	State the scheduler is in after xTaskResumeAll() is called.

4.2.4.2 TaskState_t `enum TaskState_t`

A task can be in one of the four possible states defined in the `TaskState_t` enumerated type. The state of a task is changed by calling [xTaskResume\(\)](#), [xTaskSuspend\(\)](#) or [xTaskWait\(\)](#). The `TaskState_t` enumerated type should be declared as `xTaskState`.

See also

[xTaskState](#)
[xTaskResume\(\)](#)
[xTaskSuspend\(\)](#)
[xTaskWait\(\)](#)

Enumerator

TaskStateError	Returned by xTaskGetTaskState() when task cannot be found.
TaskStateSuspended	State a task is in when it is first created by xTaskCreate() or suspended by xTaskSuspend() .
TaskStateRunning	State a task is in after xTaskResume() is called.
TaskStateWaiting	State a task is in after xTaskWait() is called.

4.2.5 Function Documentation

4.2.5.1 `__SystemAssert__()` `void __SystemAssert__ (` `const char * file_,` `int line_)`

The **SystemAssert()** system call handles assertions. The **SystemAssert()** system call should not be called directly. Instead, the `SYSASSERT()` macro should be used. The system assertion functionality will only work when the `CONFIG_ENABLE_SYSTEM_ASSERT` and `CONFIG_SYSTEM_ASSERT_BEHAVIOR` settings are defined.

See also

`SYSASSERT`
`CONFIG_ENABLE_SYSTEM_ASSERT`
`CONFIG_SYSTEM_ASSERT_BEHAVIOR`

Parameters

<i>file</i> ↔ —	This is automatically defined by the compiler's definition of <i>FILE</i>
<i>line</i> ↔ —	This is automatically defined by the compiler's definition of <i>LINE</i>

4.2.5.2 xDeviceRead() `xBase xDeviceRead (`

```

    xHWord uid_,
    xHWord * bytes_,
    xByte * data_ )

```

Parameters

<i>uid_</i>	
<i>bytes</i> ↔ —	
<i>data</i> ↔ —	

Returns

xBase

4.2.5.3 xDeviceRegisterDevice() `xBase xDeviceRegisterDevice (`

```

    xBase(*)() device_self_register_ )

```

Parameters

<i>device_self_</i> ↔ <i>register_</i>	
---	--

Returns

xBase

4.2.5.4 xDeviceWrite() `xBase xDeviceWrite (`

```

    xHWord uid_,
    xHWord * bytes_,
    xByte * data_ )

```


Parameters

<i>uid_</i>	
<i>bytes</i> ↔	
—	
<i>data</i> ↔	
—	

Returns

xBase

4.2.5.5 xMemAlloc() `xAddr xMemAlloc (`
`const xSize size_)`

The `xMemAlloc()` system call allocates memory from the heap for HeliOS system calls and end-user tasks. The size of the heap, in bytes, is dependent on the `CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS` and `CONFIG_↔
 MEMORY_REGION_BLOCK_SIZE` settings. `xMemAlloc()` functions similarly to `calloc()` in that it clears the memory it allocates.

See also

`CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS`
`CONFIG_MEMORY_REGION_BLOCK_SIZE`
`xMemFree()`

Parameters

<i>size</i> ↔	The amount (size) of the memory to be allocated from the heap in bytes.
—	

Returns

xAddr If successful, `xMemAlloc()` returns the address of the newly allocated memory. If unsuccessful, the system call will return null.

Note

HeliOS technically does not allocate memory from what is traditionally heap memory. HeliOS uses a private "heap" which is actually static memory allocated at compile time. This is done to maintain MISRA C:2012 compliance since standard library functions like `malloc()`, `calloc()` and `free()` are not permitted.

4.2.5.6 xMemFree() `void xMemFree (`
`const xAddr addr_)`

The `xMemFree()` system call will free heap memory allocated by `xMemAlloc()` and other HeliOS system calls such as `xSystemGetSystemInfo()`.

See also

[xMemAlloc\(\)](#)

Parameters

<i>addr</i> ↔	The address of the allocated heap memory to be freed.
—	

Warning

[xMemFree\(\)](#) cannot be used to free memory allocated for kernel objects. Memory allocated by [xTaskCreate\(\)](#), [xTimerCreate\(\)](#) or [xQueueCreate\(\)](#) must be freed by their respective delete system calls (i.e., [xTaskDelete\(\)](#)).

4.2.5.7 xMemGetHeapStats() [xMemoryRegionStats](#) xMemGetHeapStats (
 void)

The [xMemGetHeapStats\(\)](#) system call will return statistics about the heap so the end-user can better understand the state of the heap.

See also

[xMemoryRegionStats](#)

Returns

[xMemoryRegionStats](#) Returns the [xMemoryRegionStats](#) structure or null if unsuccessful.

Warning

The memory allocated by [xMemGetHeapStats\(\)](#) must be freed by [xMemFree\(\)](#).

4.2.5.8 xMemGetKernelStats() [xMemoryRegionStats](#) xMemGetKernelStats (
 void)

The [xMemGetKernelStats\(\)](#) system call will return statistics about the kernel memory region so the end-user can better understand the state of kernel memory.

See also

[xMemoryRegionStats](#)

Returns

[xMemoryRegionStats](#) Returns the [xMemoryRegionStats](#) structure or null if unsuccessful.

Warning

The memory allocated by [xMemGetKernelStats\(\)](#) must be freed by [xMemFree\(\)](#).

4.2.5.9 xMemGetSize() [xSize](#) xMemGetSize (
 const [xAddr](#) *addr_*)

The [xMemGetSize\(\)](#) system call returns the amount of heap memory in bytes that is currently allocated to a specific address. If the address is null or invalid, [xMemGetSize\(\)](#) will return zero bytes.

Parameters

<code>addr_↔</code> —	The address of the allocated heap memory to obtain the size of the memory, in bytes, that is allocated.
--------------------------	---

Returns

`Size_t` The amount of memory currently allocated to the specific address in bytes. If the address is invalid or null, `xMemGetSize()` will return zero.

Note

If the address `addr_` points to a structure that, for example, is 48 bytes in size based on `sizeof()`, `xMemGetSize()` will return the number of bytes allocated by the block(s) that contain the structure. Assuming the default block size of 32, a 48 byte structure would require TWO blocks so `xMemGetSize()` would return 64 - not 48. `xMemGetSize()` also checks the health of the heap and will return zero if it detects a consistency issue with the heap. Thus, `xMemGetSize()` can be used to validate addresses before the objects they reference are accessed.

4.2.5.10 `xMemGetUsed()` `xSize` `xMemGetUsed` (`void`)

The `xMemGetUsed()` system call returns the amount of heap memory, in bytes, that is currently allocated. Calls to `xMemAlloc()` increases and `xMemFree()` decreases the amount of memory in use.

Returns

`Size_t` The amount of memory currently allocated in bytes. If no heap memory is currently allocated, `xMemGetUsed()` will return zero.

Note

`xMemGetUsed()` returns the amount of heap memory that is currently allocated to end-user objects AND kernel objects. However, only end-user objects may be freed using `xMemFree()`. Kernel objects must be freed using their respective delete system call (e.g., `xTaskDelete()`).

4.2.5.11 `xQueueCreate()` `xQueue` `xQueueCreate` (`xBase` `limit_`)

The `xQueueCreate()` system call creates a message queue for inter-task communication.

See also

`xQueue`
`xQueueDelete()`
`CONFIG_QUEUE_MINIMUM_LIMIT`

Parameters

<i>limit</i> ↔ —	The message limit for the queue. When this number is reach, the queue is considered full and xQueueSend() will fail. The minimum limit for queues is dependent on the setting CONFIG_QUEUE_MINIMUM_LIMIT.
---------------------	---

Returns

[xQueue](#) A queue is returned if successful, otherwise null is returned if unsuccessful.

Warning

The message queue memory can only be freed by [xQueueDelete\(\)](#).

4.2.5.12 xQueueDelete() `void xQueueDelete (`
 [xQueue](#) *queue_* `)`

The [xQueueDelete\(\)](#) system call will delete a message queue created by [xQueueCreate\(\)](#). [xQueueDelete\(\)](#) will delete a queue regardless of how many messages the queue contains at the time [xQueueDelete\(\)](#) is called. Any messages the message queue contains will be deleted in the process of deleting the message queue.

See also

[xQueueCreate\(\)](#)

Parameters

<i>queue</i> ↔ —	The queue to be deleted.
---------------------	--------------------------

4.2.5.13 xQueueDropMessage() `void xQueueDropMessage (`
 [xQueue](#) *queue_* `)`

The [xQueueDropMessage\(\)](#) system call will drop the next message from the message queue without returning the message.

Parameters

<i>queue</i> ↔ —	The queue to drop the next message from.
---------------------	--

4.2.5.14 xQueueGetLength() `xBase` [xQueueGetLength](#) (
 [xQueue](#) *queue_* `)`

The `xQueueGetLength()` system call returns the length of the queue (the number of messages the queue currently contains).

Parameters

<code>queue</code> ↔	The queue to return the length of.
—	

Returns

`xBase` The number of messages in the queue. If unsuccessful or if the queue is empty, `xQueueGetLength()` returns zero.

4.2.5.15 `xQueueIsQueueEmpty()` `xBase` `xQueueIsQueueEmpty` (`xQueue` `queue_`)

The `xQueueIsQueueEmpty()` system call will return a true or false dependent on whether the queue is empty (message queue length is zero) or contains one or more messages.

Parameters

<code>queue</code> ↔	The queue to determine whether it is empty.
—	

Returns

`xBase` True if the queue is empty. False if the queue has one or more messages. `xQueueIsQueueEmpty()` will also return false if the queue parameter is invalid.

4.2.5.16 `xQueueIsQueueFull()` `xBase` `xQueueIsQueueFull` (`xQueue` `queue_`)

The `xQueueIsQueueFull()` system call will return a true or false dependent on whether the queue is full or contains zero messages. A queue is considered full if the number of messages in the queue is equal to the queue's length limit.

Parameters

<code>queue</code> ↔	The queue to determine whether it is full.
—	

Returns

`xBase` True if the queue is full. False if the queue has zero. `xQueueIsQueueFull()` will also return false if the queue parameter is invalid.

4.2.5.17 xQueueLockQueue() `void xQueueLockQueue (`
 `xQueue queue_)`

The `xQueueLockQueue()` system call will lock the message queue. Locking a message queue will prevent `xQueueSend()` from sending messages to the queue.

Parameters

<i>queue</i> ↔	The queue to lock.
—	

4.2.5.18 xQueueMessagesWaiting() `xBase xQueueMessagesWaiting (`
 `xQueue queue_)`

The `xQueueMessagesWaiting()` system call returns true or false dependent on whether there is at least one message waiting. The message queue does not have to be full to return true.

Parameters

<i>queue</i> ↔	The queue to determine whether one or more messages are waiting.
—	

Returns

`xBase` True if one or more messages are waiting. False if there are no messages waiting of the queue parameter is invalid.

4.2.5.19 xQueuePeek() `xQueueMessage xQueuePeek (`
 `xQueue queue_)`

The `xQueuePeek()` system call will return the next message in the specified message queue without dropping the message.

See also

[xQueueMessage](#)
[xMemFree\(\)](#)

Parameters

<i>queue</i> ↔	The queue to return the next message from.
—	

Returns

`xQueueMessage` The next message in the queue. If the queue is empty or the queue parameter is invalid, `xQueuePeek()` will return null.

Warning

The memory allocated by `xQueuePeek()` must be freed by `xMemFree()`.

4.2.5.20 xQueueReceive() `xQueueMessage` `xQueueReceive` (
 `xQueue` `queue_`)

The `xQueueReceive()` system call will return the next message in the message queue and drop it from the message queue.

See also

`xQueueMessage`
`xMemFree()`

Parameters

<code>queue_↔</code> —	The queue to return the next message from.
---------------------------	--

Returns

`xQueueMessage` The message returned from the queue. If the queue is empty of the queue parameter is invalid, `xQueueReceive()` will return null.

Warning

The memory allocated by `xQueueReceive()` must be freed by `xMemFree()`.

4.2.5.21 xQueueSend() `xBase` `xQueueSend` (
 `xQueue` `queue_`,
 `xBase` `messageBytes_`,
 const char * `messageValue_`)

The `xQueueSend()` system call will send a message using the specified message queue. The size of the message value is passed in the message bytes parameter. The maximum message value size in bytes is dependent on the `CONFIG_MESSAGE_VALUE_BYTES` setting.

See also

`CONFIG_MESSAGE_VALUE_BYTES`
`xQueuePeek()`
`xQueueReceive()`

Parameters

<i>queue_</i>	The queue to send the message to.
<i>message↔ Bytes_</i>	The number of bytes contained in the message value. The number of bytes must be greater than zero and less than or equal to the setting CONFIG_MESSAGE_VALUE_BYTES.
<i>message↔ Value_</i>	The message value. If the message value is greater than defined in CONFIG_MESSAGE_VALUE_BYTES, only the number of bytes defined in CONFIG_MESSAGE_VALUE_BYTES will be copied into the message value. The message value is NOT a null terminated string.

Returns

xBase [xQueueSend\(\)](#) returns RETURN_SUCCESS if the message was sent to the queue successfully. Otherwise RETURN_FAILURE if unsuccessful.

4.2.5.22 xQueueUnLockQueue() `void xQueueUnLockQueue (
xQueue queue_)`

The [xQueueUnLockQueue\(\)](#) system call will unlock the message queue. Unlocking a message queue will allow [xQueueSend\(\)](#) to send messages to the queue.

Parameters

<i>queue↔ _</i>	The queue to unlock.
---------------------	----------------------

4.2.5.23 xStreamBytesAvailable() `xHWord xStreamBytesAvailable (
xStreamBuffer stream_)`

The [xStreamBytesAvailable\(\)](#) system call will return the length of the stream buffer in bytes (i.e., bytes available to be received by [xStreamReceive\(\)](#)).

Parameters

<i>stream↔ _</i>	The stream to operate on.
----------------------	---------------------------

Returns

xHWord The length of the stream buffer in bytes.

4.2.5.24 xStreamCreate() `xStreamBuffer xStreamCreate (
void)`

The [xStreamCreate\(\)](#) system call will create a new stream buffer. The memory for a stream buffer is allocated from kernel memory and therefor cannot be freed by calling [xMemFree\(\)](#).

Returns

`xStreamBuffer` The newly created stream buffer.

Warning

The stream buffer created by `xStreamCreate()` must be freed by calling `xStreamDelete()`.

4.2.5.25 xStreamDelete() `void xStreamDelete (`
`xStreamBuffer stream_)`

The `xStreamDelete()` system call will delete a stream buffer and free its memory. Once a stream buffer is deleted, it can not be written to or read from.

Parameters

<code>stream↔</code>	The stream buffer to operate on.
<code>_</code>	

4.2.5.26 xStreamIsEmpty() `xBase xStreamIsEmpty (`
`xStreamBuffer stream_)`

The `xStreamIsEmpty()` system call is used to determine if the stream buffer is empty. A stream buffer is considered empty when it's length is equal to zero. If the buffer is greater than zero in length, `xStreamIsEmpty()` will return false.

Parameters

<code>stream↔</code>	The stream buffer to operate on.
<code>_</code>	

Returns

`xBase` Returns true if the stream buffer length is equal to zero in length, otherwise `xStreamIsEmpty()` will return false.

4.2.5.27 xStreamIsFull() `xBase xStreamIsFull (`
`xStreamBuffer steam_)`

The `xStreamIsFull()` system call is used to determine if the stream buffer is full. A stream buffer is considered full when it's length is equal to `CONFIG_STREAM_BUFFER_BYTES`. If the buffer is less than `CONFIG_STREAM_↔
BUFFER_BYTES` in length, `xStreamIsFull()` will return false.

Parameters

<i>stream</i> ↔	The stream buffer to operate on.
—	

Returns

`xBase` Returns true if the stream buffer is equal to `CONFIG_STREAM_BUFFER_BYTES` in length, otherwise `xStreamIsFull()` will return false.

4.2.5.28 `xStreamReceive()` `xByte*` `xStreamReceive` (
 `xStreamBuffer` *stream_*,
 `HWord_t` * *bytes_*)

The `xStreamReceive()` system call will return the contents of the stream buffer. The contents are returned as a byte array whose length is known by the `bytes_` paramater. Because the byte array is stored in the heap, it must be freed by calling `xMemFree()`.

Parameters

<i>stream</i> ↔	The stream to operate on.
—	
<i>bytes</i> ↔	The number of bytes returned (i.e., length of the byte array) by <code>xStreamReceive()</code> .
—	

Returns

`xByte*` The byte array containing the contents of the stream buffer.

Warning

The byte array returned by `xStreamReceive()` must be freed by calling `xMemFree()`.

4.2.5.29 `xStreamReset()` `void` `xStreamReset` (
 `xStreamBuffer` *stream_*)

The `xStreamReset()` system call will clear the contents of the stream buffer and reset its length to zero.

Parameters

<i>stream</i> ↔	The stream buffer to operate on.
—	

4.2.5.30 xStreamSend() `xBase` xStreamSend (
 `xStreamBuffer` *stream_*,
 `xByte` *byte_*)

The `xStreamSend()` system call will write one byte to the stream buffer. If the stream buffer's length is equal to `CONFIG_STREAM_BUFFER_BYTES` (i.e., full) then the byte will not be written to the stream buffer and `xStreamSend()` will return `RETURN_FAILURE`.

Parameters

<i>stream_</i> ↔	
<i>byte_</i>	

Returns

`xBase`

4.2.5.31 xSystemGetSystemInfo() `xSystemInfo` xSystemGetSystemInfo (
 `void`)

Returns

`xSystemInfo` The system info is returned if successful, otherwise null is returned if unsuccessful.

See also

`xSystemInfo`

`xMemFree()`

Warning

The memory allocated by the `xSystemGetSystemInfo()` must be freed with `xMemFree()`

4.2.5.32 xSystemHalt() `void` xSystemHalt (
 `void`)

The `xSystemHalt()` system call will halt HeliOS. Once `xSystemHalt()` is called, the system must be reset.

4.2.5.33 xSystemInit() `void` xSystemInit (
 `void`)

The `xSystemInit()` system call initializes the required interrupt handlers and memory and must be called prior to calling any other system call.

4.2.5.34 xTaskChangePeriod() `void xTaskChangePeriod (`
`xTask task_,`
`xTicks timerPeriod_)`

The `xTaskChangePeriod()` system call will change the period (ticks) on the task timer for the specified task. The timer period must be greater than zero. To have any effect, the task must be in the waiting state set by calling `xTaskWait()` on the task. Once the timer period is set and the task is in the waiting state, the task will be executed every `timerPeriod_` ticks. Changing the period to zero will prevent the task from being executed even if it is in the waiting state unless it were to receive a direct to task notification.

See also

[xTaskWait\(\)](#)
[xTaskGetPeriod\(\)](#)
[xTaskResetTimer\(\)](#)

Parameters

<i>task_</i>	The task to change the timer period for.
<i>timer↔ Period_</i>	The timer period in ticks.

4.2.5.35 xTaskChangeWDPeriod() `void xTaskChangeWDPeriod (`
`xTask * task_,`
`xTicks wdTimerPeriod_)`

The `xTaskChangeWDPeriod()` system call will change the task watchdog timer period. The period, measured in ticks, must be greater than zero to have any effect. If the tasks last runtime exceeds the task watchdog timer period, the task will automatically be placed in a suspended state.

See also

[xTaskGetWDPeriod\(\)](#)

Parameters

<i>task_</i>	The task to change the task watchdog timer for.
<i>wdTimer↔ Period_</i>	The task watchdog timer period which is measured in ticks. If zero, the task watchdog timer will not have any effect.

4.2.5.36 xTaskCreate() `xTask xTaskCreate (`
`const char * name_,`
`void(*) (xTask, xTaskParm) callback_,`
`xTaskParm taskParameter_)`

The `xTaskCreate()` system call will create a new task. The task will be created with its state set to suspended. The `xTaskCreate()` and `xTaskDelete()` system calls cannot be called within a task. They MUST be called outside of the scope of the HeliOS scheduler.

Parameters

<i>name_</i>	The ASCII name of the task which can be used by xTaskGetHandleByName() to obtain the task handle. The length of the name is depended on the CONFIG_TASK_NAME_BYTES. The task name is NOT a null terminated char string.
<i>callback_</i>	The address of the task main function. This is the function that will be invoked by the scheduler when a task is scheduled for execution.
<i>task_</i> <i>Parameter_</i>	A pointer to any type or structure that the end-user wants to pass into the task as a parameter. The task parameter is not required and may simply be set to null.

Returns

xTask A handle to the newly created task.

See also

[xTask](#)
[xTaskParm](#)
[xTaskDelete\(\)](#)
[xTaskState](#)
[CONFIG_TASK_NAME_BYTES](#)

Warning

[xTaskCreate\(\)](#) MUST be called outside the scope of the HeliOS scheduler (i.e., not from a task's main). The task memory can only be freed by [xTaskDelete\(\)](#).

4.2.5.37 xTaskDelete() `void xTaskDelete (`
`xTask task_)`

The [xTaskDelete\(\)](#) system call will delete a task. The [xTaskCreate\(\)](#) and [xTaskDelete\(\)](#) system calls cannot be called within a task. They MUST be called outside of the scope of the HeliOS scheduler.

Parameters

<i>task_</i> —	The handle of the task to be deleted.
-------------------	---------------------------------------

Warning

[xTaskDelete\(\)](#) MUST be called outside the scope of the HeliOS scheduler (i.e., not from a task's main).

4.2.5.38 xTaskGetAllRunTimeStats() `xTaskRunTimeStats xTaskGetAllRunTimeStats (`
`xBase * tasks_)`

The [xTaskGetAllRunTimeStats\(\)](#) system call will return the runtime statistics for all of the tasks regardless of their state. The [xTaskGetAllRunTimeStats\(\)](#) system call returns the `xTaskRunTimeStats` type. An `xBase` variable must be passed by reference to [xTaskGetAllRunTimeStats\(\)](#) which will be updated by [xTaskGetAllRunTimeStats\(\)](#) to contain the number of tasks so the end-user can iterate through the tasks. The `xTaskRunTimeStats` memory must be freed by [xMemFree\(\)](#) after it is no longer needed.

See also

[xTaskRunTimeStats](#)
[xMemFree\(\)](#)

Parameters

<code>tasks_↔</code> —	A variable of type <code>xBase</code> passed by reference which will contain the number of tasks upon return. If no tasks currently exist, this variable will not be modified.
---------------------------	--

Returns

`xTaskRunTimeStats` The runtime stats returned by [xTaskGetAllRunTimeStats\(\)](#). If there are currently no tasks then this will be null. This memory must be freed by [xMemFree\(\)](#).

Warning

The memory allocated by [xTaskGetAllRunTimeStats\(\)](#) must be freed by [xMemFree\(\)](#).

4.2.5.39 xTaskGetAllTaskInfo() `xTaskInfo` xTaskGetAllTaskInfo (`xBase * tasks_`)

The [xTaskGetAllTaskInfo\(\)](#) system call returns the `xTaskInfo` structure containing the details of ALL tasks including their identifier, name, state and runtime statistics.

See also

[xTaskInfo](#)

Parameters

<code>tasks_↔</code> —	A variable of type <code>xBase</code> passed by reference which will contain the number of tasks upon return. If no tasks currently exist, this variable will not be modified.
---------------------------	--

Returns

`xTaskInfo` The `xTaskInfo` structure containing the tasks details. [xTaskGetAllTaskInfo\(\)](#) returns null if there no tasks or if a consistency issue is detected.

Warning

The memory allocated by [xTaskGetAllTaskInfo\(\)](#) must be freed by [xMemFree\(\)](#).

4.2.5.40 xTaskGetHandleById() `xTask` xTaskGetHandleById (
 `xBase` *id_*)

The `xTaskGetHandleById()` system call will return the task handle of the task specified by identifier identifier.

See also

`xBase`

Parameters

<i>id_</i> ↔ _↔	The identifier of the task to return the handle of.
--------------------	---

Returns

`xTask` The task handle. `xTaskGetHandleById()` returns null if the the task identifier cannot be found.

4.2.5.41 xTaskGetHandleByName() `xTask` xTaskGetHandleByName (
 `const char *` *name_*)

The `xTaskGetHandleByName()` system call will return the task handle of the task specified by its ASCII name. The length of the task name is dependent on the `CONFIG_TASK_NAME_BYTES` setting. The name is compared byte-for-byte so the name is case sensitive.

See also

`CONFIG_TASK_NAME_BYTES`

Parameters

<i>name_</i> ↔ _	The ASCII name of the task to return the handle of. The task name is NOT a null terminated string.
---------------------	--

Returns

`xTask` The task handle. `xTaskGetHandleByName()` returns null if the name cannot be found.

4.2.5.42 xTaskGetId() `xBase` xTaskGetId (
 `xTask` *task_*)

The `xTaskGetId()` system call returns the task identifier for the task.

Parameters

<i>task</i> ↔	The task to return the identifier of.
—	

Returns

xBase The identifier of the task. If the task cannot be found, [xTaskGetId\(\)](#) returns zero (all tasks identifiers are 1 or greater).

4.2.5.43 xTaskGetName() `char* xTaskGetName (`
 [xTask](#) *task_* `)`

The [xTaskGetName\(\)](#) system call returns the ASCII name of the task. The size of the task is dependent on the setting CONFIG_TASK_NAME_BYTES. The task name is NOT a null terminated char string. The memory allocated for the char array must be freed by [xMemFree\(\)](#) when no longer needed.

See also

[CONFIG_TASK_NAME_BYTES](#)
[xMemFree\(\)](#)

Parameters

<i>task</i> ↔	The task to return the name of.
—	

Returns

char* A pointer to the char array containing the ASCII name of the task. The task name is NOT a null terminated char string. [xTaskGetName\(\)](#) will return null if the task cannot be found.

Warning

The memory allocated by [xTaskGetName\(\)](#) must be free by [xMemFree\(\)](#).

4.2.5.44 xTaskGetNumberOfTasks() `xBase xTaskGetNumberOfTasks (`
 void `)`

The [xTaskGetNumberOfTasks\(\)](#) system call returns the current number of tasks regardless of their state.

Returns

xBase The number of tasks.

4.2.5.45 xTaskGetPeriod() `xTicks xTaskGetPeriod (`
`xTask task_)`

The `xTaskGetPeriod()` will return the period for the timer for the specified task. See `xTaskChangePeriod()` for more information on how the task timer works.

See also

`xTaskWait()`
`xTaskChangePeriod()`
`xTaskResetTimer()`

Parameters

<code>task_↔</code>	The task to return the timer period for.
—	

Returns

`xTicks` The timer period in ticks. `xTaskGetPeriod()` will return zero if the timer period is zero or if the task could not be found.

4.2.5.46 xTaskGetSchedulerState() `xSchedulerState xTaskGetSchedulerState (`
`void)`

The `xTaskGetSchedulerState()` system call will return the state of the scheduler. The state of the scheduler can only be changed using `xTaskSuspendAll()` and `xTaskResumeAll()`.

See also

`xSchedulerState`
`xTaskSuspendAll()`
`xTaskResumeAll()`

Returns

`xSchedulerState` The state of the scheduler.

4.2.5.47 xTaskGetTaskInfo() `xTaskInfo xTaskGetTaskInfo (`
`xTask task_)`

The `xTaskGetTaskInfo()` system call returns the `xTaskInfo` structure containing the details of the task including its identifier, name, state and runtime statistics.

See also

`xTaskInfo`

Parameters

<i>task</i> ↔	The task to return the details of.
—	

Returns

`xTaskInfo` The `xTaskInfo` structure containing the task details. `xTaskGetTaskInfo()` returns null if the task cannot be found.

Warning

The memory allocated by `xTaskGetTaskInfo()` must be freed by `xMemFree()`.

4.2.5.48 `xTaskGetTaskRunTimeStats()` `xTaskRunTimeStats` `xTaskGetTaskRunTimeStats (`
`xTask task_)`

The `xTaskGetTaskRunTimeStats()` system call returns the task runtime statistics for one task. The `xTaskGetTaskRunTimeStats()` system call returns the `xTaskRunTimeStats` type. The memory must be freed by calling `xMemFree()` after it is no longer needed.

See also

`xTaskRunTimeStats`
`xMemFree()`

Parameters

<i>task</i> ↔	The task to get the runtime statistics for.
—	

Returns

`xTaskRunTimeStats` The runtime stats returned by `xTaskGetTaskRunTimeStats()`. `xTaskGetTaskRunTimeStats()` will return null if the task cannot be found.

Warning

The memory allocated by `xTaskGetTaskRunTimeStats()` must be freed by `xMemFree()`.

4.2.5.49 `xTaskGetTaskState()` `xTaskState` `xTaskGetTaskState (`
`xTask task_)`

The `xTaskGetTaskState()` system call will return the state of the task.

See also

`xTaskState`

Parameters

<i>task</i> ↔	The task to return the state of.
—	

Returns

xTaskState The xTaskState of the task. If the task cannot be found, [xTaskGetTaskState\(\)](#) will return null.

4.2.5.50 xTaskGetWDPeriod() [xTicks](#) xTaskGetWDPeriod (
 [xTask](#) * *task_*)

The [xTaskGetWDPeriod\(\)](#) will return the current task watchdog timer for the task.

See also

[xTaskChangeWDPeriod\(\)](#)

Parameters

<i>task</i> ↔	The task to get the task watchdog timer period for.
—	

Returns

Ticks_t The task watchdog timer period which is measured in ticks.

4.2.5.51 xTaskNotificationIsWaiting() [xBase](#) xTaskNotificationIsWaiting (
 [xTask](#) *task_*)

The [xTaskNotificationIsWaiting\(\)](#) system call will return true or false depending on whether there is a direct to task notification waiting for the task.

Parameters

<i>task</i> ↔	The task to check for a waiting task notification.
—	

Returns

xBase Returns true if there is a task notification. False if there is no notification or if the task could not be found.

4.2.5.52 xTaskNotifyGive() `Base_t xTaskNotifyGive (`
 `xTask task_,`
 `xBase notificationBytes_,`
 `const char * notificationValue_)`

The `xTaskNotifyGive()` system call will give a direct to task notification to the specified task. The task notification bytes is the number of bytes contained in the notification value. The number of notification bytes must be between one and the `CONFIG_NOTIFICATION_VALUE_BYTES` setting. The notification value must contain a pointer to a char array containing the notification value. If the task already has a waiting task notification, `xTaskNotifyGive()` will NOT overwrite the waiting task notification. `xTaskNotifyGive()` will return true if the direct to task notification was successfully given.

See also

[CONFIG_NOTIFICATION_VALUE_BYTES](#)
[xTaskNotifyTake\(\)](#)

Parameters

<i>task_</i>	The task to send the task notification to.
<i>notification↔ Bytes_</i>	The number of bytes contained in the notification value. The number must be between one and the <code>CONFIG_NOTIFICATION_VALUE_BYTES</code> setting.
<i>notification↔ Value_</i>	A char array containing the notification value. The notification value is NOT a null terminated string.

Returns

`xBase RETURN_SUCCESS` if the direct to task notification was successfully given, `RETURN_FAILURE` if not.

4.2.5.53 xTaskNotifyStateClear() `void xTaskNotifyStateClear (`
 `xTask task_)`

The `xTaskNotifyStateClear()` system call will clear a waiting direct to task notification if one exists without returning the notification.

Parameters

<i>task↔ _</i>	The task to clear the notification for.
--------------------	---

4.2.5.54 xTaskNotifyTake() `xTaskNotification xTaskNotifyTake (`
 `xTask task_)`

The `xTaskNotifyTake()` system call will return the waiting direct to task notification if there is one. The `xTaskNotifyTake()` system call will return an `xTaskNotification` structure containing the notification bytes and its value. The memory allocated by `xTaskNotifyTake()` must be freed by `xMemFree()`.

See also

[xTaskNotification](#)
[xTaskNotifyGive\(\)](#)
[xMemFree\(\)](#)
[CONFIG_NOTIFICATION_VALUE_BYTES](#)

Parameters

<i>task</i> ↔	The task to return a waiting task notification.
—	

Returns

[xTaskNotification](#) The [xTaskNotification](#) structure containing the notification bytes and value. [xTaskNotifyTake\(\)](#) will return null if no waiting task notification exists or if the task cannot be found.

Warning

The memory allocated by [xTaskNotifyTake\(\)](#) must be freed by [xMemFree\(\)](#).

4.2.5.55 xTaskResetTimer() `void xTaskResetTimer (`
`xTask task_)`

The [xTaskResetTimer\(\)](#) system call will reset the task timer. [xTaskResetTimer\(\)](#) does not change the timer period or the task state when called. See [xTaskChangePeriod\(\)](#) for more details on task timers.

See also

[xTaskWait\(\)](#)
[xTaskChangePeriod\(\)](#)
[xTaskGetPeriod\(\)](#)

Parameters

<i>task</i> ↔	The task to reset the task timer for.
—	

4.2.5.56 xTaskResume() `void xTaskResume (`
`xTask task_)`

The [xTaskResume\(\)](#) system call will resume a suspended task. Tasks are suspended on creation so either [xTaskResume\(\)](#) or [xTaskWait\(\)](#) must be called to place the task in a state that the scheduler will execute.

See also

[xTaskState](#)
[xTaskSuspend\(\)](#)
[xTaskWait\(\)](#)

Parameters

<i>task</i> ↔	The task to set its state to running.
—	

4.2.5.57 xTaskResumeAll() `void xTaskResumeAll (`
`void)`

The [xTaskResumeAll\(\)](#) system call will set the scheduler state to running so the next call to [xTaskStartScheduler\(\)](#) will resume execute of all tasks. The state of each task is not altered by [xTaskSuspendAll\(\)](#) or [xTaskResumeAll\(\)](#).

See also

[xTaskSuspendAll\(\)](#)

4.2.5.58 xTaskStartScheduler() `void xTaskStartScheduler (`
`void)`

The [xTaskStartScheduler\(\)](#) system call passes control to the HeliOS scheduler. This system call will not return until [xTaskSuspendAll\(\)](#) is called. If [xTaskSuspendAll\(\)](#) is called, [xTaskResumeAll\(\)](#) must be called before [xTaskStartScheduler\(\)](#) can be called again to continue executing tasks.

4.2.5.59 xTaskSuspend() `void xTaskSuspend (`
`xTask task_)`

The [xTaskSuspend\(\)](#) system call will suspend a task. A task that has been suspended will not be executed by the scheduler until [xTaskResume\(\)](#) or [xTaskWait\(\)](#) is called.

See also

[xTaskState](#)
[xTaskResume\(\)](#)
[xTaskWait\(\)](#)

Parameters

<i>task</i> ↔	The task to suspend.
—	

4.2.5.60 xTaskSuspendAll() `void xTaskSuspendAll (`
`void)`

The [xTaskSuspendAll\(\)](#) system call will set the scheduler state to suspended so the scheduler will stop and return. The state of each task is not altered by [xTaskSuspendAll\(\)](#) or [xTaskResumeAll\(\)](#).

See also

[xTaskResumeAll\(\)](#)

4.2.5.61 xTaskWait() `void xTaskWait (`
`xTask task_)`

The [xTaskWait\(\)](#) system call will place a task in the waiting state. A task must be in the waiting state for event driven multitasking with either direct to task notifications OR setting the period on the task timer with [xTaskChangePeriod\(\)](#). A task in the waiting state will not be executed by the scheduler until an event has occurred.

See also

[xTaskState](#)

[xTaskResume\(\)](#)

[xTaskSuspend\(\)](#)

Parameters

<i>task_</i> ↔	The task to place in the waiting state.
—	

4.2.5.62 xTimerChangePeriod() `void xTimerChangePeriod (`
`xTimer timer_ ,`
`xTicks timerPeriod_)`

The [xTimerChangePeriod\(\)](#) system call will change the period of the specified timer. The timer period is measured in ticks. If the timer period is zero, the [xTimerHasTimerExpired\(\)](#) system call will always return false.

See also

[xTimerHasTimerExpired\(\)](#)

Parameters

<i>timer_</i>	The timer to change the period for.
<i>timer_</i> ↔ <i>Period_</i>	The timer period in ticks. Timer period must be zero or greater.

4.2.5.63 xTimerCreate() `xTimer xTimerCreate (`
`xTicks timerPeriod_)`

The `xTimerCreate()` system call will create a new timer. Timers differ from task timers in that they do not create events that effect the scheduling of a task. Timers can be used by tasks to initiate various task activities based on a specified time period represented in ticks. The memory allocated by `xTimerCreate()` must be freed by `xTimerDelete()`. Unlike tasks, timers may be created and deleted within tasks.

See also

`xTimer`
`xTimerDelete()`

Parameters

<i>timer</i> ↔ <i>Period_</i>	The number of ticks before the timer expires.
----------------------------------	---

Returns

`xTimer` The newly created timer. If the timer period parameter is less than zero or `xTimerCreate()` was unable to allocate the required memory, `xTimerCreate()` will return null.

Warning

The timer memory can only be freed by `xTimerDelete()`.

4.2.5.64 xTimerDelete() `void xTimerDelete (`
`xTimer timer_)`

The `xTimerDelete()` system call will delete a timer. For more information on timers see the `xTaskTimerCreate()` system call.

See also

`xTimerCreate()`

Parameters

<i>timer</i> ↔ —	The timer to be deleted.
---------------------	--------------------------

4.2.5.65 xTimerGetPeriod() `xTicks` xTimerGetPeriod (
 `xTimer` *timer_*)

The [xTimerGetPeriod\(\)](#) system call will return the current timer period for the specified timer.

Parameters

<i>timer</i> ↔	The timer to get the timer period for.
—	

Returns

`xTicks` The timer period. If the timer cannot be found, [xTimerGetPeriod\(\)](#) will return zero.

4.2.5.66 xTimerHasTimerExpired() `xBase` xTimerHasTimerExpired (
 `xTimer` *timer_*)

The [xTimerHasTimerExpired\(\)](#) system call will return true or false dependent on whether the timer period for the specified timer has elapsed. [xTimerHasTimerExpired\(\)](#) will NOT reset the timer. Timers will not automatically reset. Timers MUST be reset with [xTimerReset\(\)](#).

See also

[xTimerReset\(\)](#)

Parameters

<i>timer</i> ↔	The timer to determine if the period has expired.
—	

Returns

`xBase` True if the timer has expired, false if the timer has not expired or could not be found.

4.2.5.67 xTimerIsTimerActive() `xBase` xTimerIsTimerActive (
 `xTimer` *timer_*)

The [xTimerIsTimerActive\(\)](#) system call will return true if the timer has been started with [xTimerStart\(\)](#).

See also

[xTimerStart\(\)](#)

Parameters

<i>timer</i> ↔	The timer to check if active.
—	

Returns

xBase True if active, false if not active or if the timer could not be found.

4.2.5.68 xTimerReset() `void xTimerReset (`
 xTimer *timer_*)

The [xTimerReset\(\)](#) system call will reset the start time of the timer to zero.

Parameters

<i>timer</i> ↔	The timer to be reset.
—	

4.2.5.69 xTimerStart() `void xTimerStart (`
 xTimer *timer_*)

The [xTimerStart\(\)](#) system call will place the timer in the running (active) state. Neither [xTimerStart\(\)](#) nor [xTimerStop\(\)](#) will reset the timer. Timers can only be reset with [xTimerReset\(\)](#).

See also

[xTimerStop\(\)](#)
[xTimerReset\(\)](#)

Parameters

<i>timer</i> ↔	The timer to be started.
—	

4.2.5.70 xTimerStop() `void xTimerStop (`
 xTimer *timer_*)

See also

[xTimerStart\(\)](#)
[xTimerReset\(\)](#)

Parameters

<i>timer</i> ↔ —	The timer to be stopped.
---------------------	--------------------------

Index

`__SystemAssert__`
HeliOS.h, [30](#)

`Addr_t`
HeliOS.h, [19](#)

`availableSpaceInBytes`
MemoryRegionStats_s, [3](#)

`Base_t`
HeliOS.h, [19](#)

`Byte_t`
HeliOS.h, [19](#)

`config.h`, [9](#)

- `CONFIG_MEMORY_REGION_BLOCK_SIZE`, [10](#)
- `CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS`, [10](#)
- `CONFIG_MESSAGE_VALUE_BYTES`, [11](#)
- `CONFIG_NOTIFICATION_VALUE_BYTES`, [11](#)
- `CONFIG_QUEUE_MINIMUM_LIMIT`, [12](#)
- `CONFIG_STREAM_BUFFER_BYTES`, [12](#)
- `CONFIG_TASK_NAME_BYTES`, [12](#)
- `CONFIG_TASK_WD_TIMER_ENABLE`, [12](#)

`CONFIG_MEMORY_REGION_BLOCK_SIZE`
config.h, [10](#)

`CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS`
config.h, [10](#)

`CONFIG_MESSAGE_VALUE_BYTES`
config.h, [11](#)

`CONFIG_NOTIFICATION_VALUE_BYTES`
config.h, [11](#)

`CONFIG_QUEUE_MINIMUM_LIMIT`
config.h, [12](#)

`CONFIG_STREAM_BUFFER_BYTES`
config.h, [12](#)

`CONFIG_TASK_NAME_BYTES`
config.h, [12](#)

`CONFIG_TASK_WD_TIMER_ENABLE`
config.h, [12](#)

`DEREF_TASKPARAM`
HeliOS.h, [18](#)

HeliOS.h, [13](#)

- `__SystemAssert__`, [30](#)
- `Addr_t`, [19](#)
- `Base_t`, [19](#)
- `Byte_t`, [19](#)
- `DEREF_TASKPARAM`, [18](#)
- `HWord_t`, [19](#)
- `MemoryRegionStats_t`, [19](#)
- `Queue_t`, [19](#)
- `QueueMessage_t`, [20](#)
- `SchedulerState_t`, [29](#)
- `SchedulerStateError`, [30](#)
- `SchedulerStateRunning`, [30](#)
- `SchedulerStateSuspended`, [30](#)
- `Size_t`, [20](#)
- `StreamBuffer_t`, [20](#)
- `SystemInfo_t`, [21](#)
- `Task_t`, [21](#)
- `TaskInfo_t`, [21](#)
- `TaskNotification_t`, [22](#)
- `TaskParm_t`, [22](#)
- `TaskRunTimeStats_t`, [22](#)
- `TaskState_t`, [30](#)
- `TaskStateError`, [30](#)
- `TaskStateRunning`, [30](#)
- `TaskStateSuspended`, [30](#)
- `TaskStateWaiting`, [30](#)
- `Ticks_t`, [23](#)
- `Timer_t`, [23](#)
- `Word_t`, [23](#)
- `xAddr`, [24](#)
- `xBase`, [24](#)
- `xByte`, [24](#)
- `xDeviceRead`, [31](#)
- `xDeviceRegisterDevice`, [31](#)
- `xDeviceWrite`, [31](#)
- `xHWord`, [24](#)
- `xMemAlloc`, [32](#)
- `xMemFree`, [32](#)
- `xMemGetHeapStats`, [33](#)
- `xMemGetKernelStats`, [33](#)
- `xMemGetSize`, [33](#)
- `xMemGetUsed`, [34](#)
- `xMemoryRegionStats`, [24](#)
- `xQueue`, [25](#)
- `xQueueCreate`, [34](#)
- `xQueueDelete`, [35](#)
- `xQueueDropMessage`, [35](#)
- `xQueueGetLength`, [35](#)
- `xQueueIsQueueEmpty`, [36](#)
- `xQueueIsQueueFull`, [36](#)
- `xQueueLockQueue`, [36](#)
- `xQueueMessage`, [25](#)
- `xQueueMessagesWaiting`, [37](#)
- `xQueuePeek`, [37](#)
- `xQueueReceive`, [38](#)
- `xQueueSend`, [38](#)
- `xQueueUnLockQueue`, [39](#)
- `xSchedulerState`, [25](#)
- `xSize`, [26](#)
- `xStreamBuffer`, [26](#)
- `xStreamBytesAvailable`, [39](#)
- `xStreamCreate`, [39](#)
- `xStreamDelete`, [40](#)
- `xStreamIsEmpty`, [40](#)
- `xStreamIsFull`, [40](#)
- `xStreamReceive`, [41](#)
- `xStreamReset`, [41](#)

- xStreamSend, 41
- xSystemGetSystemInfo, 42
- xSystemHalt, 42
- xSystemInfo, 26
- xSystemInit, 42
- xTask, 26
- xTaskChangePeriod, 42
- xTaskChangeWDPPeriod, 43
- xTaskCreate, 43
- xTaskDelete, 44
- xTaskGetAllRunTimeStats, 44
- xTaskGetAllTaskInfo, 45
- xTaskGetHandleById, 45
- xTaskGetHandleByName, 46
- xTaskGetId, 46
- xTaskGetName, 47
- xTaskGetNumberOfTasks, 47
- xTaskGetPeriod, 47
- xTaskGetSchedulerState, 48
- xTaskGetTaskInfo, 48
- xTaskGetTaskRunTimeStats, 49
- xTaskGetTaskState, 49
- xTaskGetWDPPeriod, 50
- xTaskInfo, 27
- xTaskNotification, 27
- xTaskNotificationIsWaiting, 50
- xTaskNotifyGive, 50
- xTaskNotifyStateClear, 51
- xTaskNotifyTake, 51
- xTaskParm, 27
- xTaskResetTimer, 52
- xTaskResume, 52
- xTaskResumeAll, 53
- xTaskRunTimeStats, 28
- xTaskStartScheduler, 53
- xTaskState, 28
- xTaskSuspend, 53
- xTaskSuspendAll, 54
- xTaskWait, 54
- xTicks, 28
- xTimer, 29
- xTimerChangePeriod, 54
- xTimerCreate, 55
- xTimerDelete, 55
- xTimerGetPeriod, 55
- xTimerHasTimerExpired, 56
- xTimerIsTimerActive, 56
- xTimerReset, 57
- xTimerStart, 57
- xTimerStop, 57
- xWord, 29
- HWord_t
 - HeliOS.h, 19
- id
 - TaskInfo_s, 6
 - TaskRunTimeStats_s, 8
- largestFreeEntryInBytes
 - MemoryRegionStats_s, 3
- lastRunTime
 - TaskInfo_s, 6
 - TaskRunTimeStats_s, 9
- majorVersion
 - SystemInfo_s, 5
- MemoryRegionStats_s, 2
 - availableSpaceInBytes, 3
 - largestFreeEntryInBytes, 3
 - minimumEverFreeBytesRemaining, 3
 - numberOfFreeBlocks, 3
 - smallestFreeEntryInBytes, 3
 - successfulAllocations, 3
 - successfulFrees, 3
- MemoryRegionStats_t
 - HeliOS.h, 19
- messageBytes
 - QueueMessage_s, 4
- messageValue
 - QueueMessage_s, 4
- minimumEverFreeBytesRemaining
 - MemoryRegionStats_s, 3
- minorVersion
 - SystemInfo_s, 5
- name
 - TaskInfo_s, 7
- notificationBytes
 - TaskNotification_s, 8
- notificationValue
 - TaskNotification_s, 8
- numberOfFreeBlocks
 - MemoryRegionStats_s, 3
- numberOfTasks
 - SystemInfo_s, 5
- patchVersion
 - SystemInfo_s, 5
- productName
 - SystemInfo_s, 5
- Queue_t
 - HeliOS.h, 19
- QueueMessage_s, 4
 - messageBytes, 4
 - messageValue, 4
- QueueMessage_t
 - HeliOS.h, 20
- SchedulerState_t
 - HeliOS.h, 29
- SchedulerStateError
 - HeliOS.h, 30
- SchedulerStateRunning
 - HeliOS.h, 30
- SchedulerStateSuspended
 - HeliOS.h, 30
- Size_t

- HeliOS.h, 20
- smallestFreeEntryInBytes
 - MemoryRegionStats_s, 3
- state
 - TaskInfo_s, 7
- StreamBuffer_t
 - HeliOS.h, 20
- successfulAllocations
 - MemoryRegionStats_s, 3
- successfulFrees
 - MemoryRegionStats_s, 3
- SystemInfo_s, 4
 - majorVersion, 5
 - minorVersion, 5
 - numberOfTasks, 5
 - patchVersion, 5
 - productName, 5
- SystemInfo_t
 - HeliOS.h, 21
- Task_t
 - HeliOS.h, 21
- TaskInfo_s, 6
 - id, 6
 - lastRunTime, 6
 - name, 7
 - state, 7
 - totalRunTime, 7
- TaskInfo_t
 - HeliOS.h, 21
- TaskNotification_s, 7
 - notificationBytes, 8
 - notificationValue, 8
- TaskNotification_t
 - HeliOS.h, 22
- TaskParm_t
 - HeliOS.h, 22
- TaskRunTimeStats_s, 8
 - id, 8
 - lastRunTime, 9
 - totalRunTime, 9
- TaskRunTimeStats_t
 - HeliOS.h, 22
- TaskState_t
 - HeliOS.h, 30
- TaskStateError
 - HeliOS.h, 30
- TaskStateRunning
 - HeliOS.h, 30
- TaskStateSuspended
 - HeliOS.h, 30
- TaskStateWaiting
 - HeliOS.h, 30
- Ticks_t
 - HeliOS.h, 23
- Timer_t
 - HeliOS.h, 23
- totalRunTime
 - TaskInfo_s, 7
- TaskRunTimeStats_s, 9
- Word_t
 - HeliOS.h, 23
- xAddr
 - HeliOS.h, 24
- xBase
 - HeliOS.h, 24
- xByte
 - HeliOS.h, 24
- xDeviceRead
 - HeliOS.h, 31
- xDeviceRegisterDevice
 - HeliOS.h, 31
- xDeviceWrite
 - HeliOS.h, 31
- xHWord
 - HeliOS.h, 24
- xMemAlloc
 - HeliOS.h, 32
- xMemFree
 - HeliOS.h, 32
- xMemGetHeapStats
 - HeliOS.h, 33
- xMemGetKernelStats
 - HeliOS.h, 33
- xMemGetSize
 - HeliOS.h, 33
- xMemGetUsed
 - HeliOS.h, 34
- xMemoryRegionStats
 - HeliOS.h, 24
- xQueue
 - HeliOS.h, 25
- xQueueCreate
 - HeliOS.h, 34
- xQueueDelete
 - HeliOS.h, 35
- xQueueDropMessage
 - HeliOS.h, 35
- xQueueGetLength
 - HeliOS.h, 35
- xQueueIsQueueEmpty
 - HeliOS.h, 36
- xQueueIsQueueFull
 - HeliOS.h, 36
- xQueueLockQueue
 - HeliOS.h, 36
- xQueueMessage
 - HeliOS.h, 25
- xQueueMessagesWaiting
 - HeliOS.h, 37
- xQueuePeek
 - HeliOS.h, 37
- xQueueReceive
 - HeliOS.h, 38
- xQueueSend
 - HeliOS.h, 38

xQueueUnLockQueue	xTaskGetSchedulerState
HeliOS.h, 39	HeliOS.h, 48
xSchedulerState	xTaskGetTaskInfo
HeliOS.h, 25	HeliOS.h, 48
xSize	xTaskGetTaskRunTimeStats
HeliOS.h, 26	HeliOS.h, 49
xStreamBuffer	xTaskGetTaskState
HeliOS.h, 26	HeliOS.h, 49
xStreamBytesAvailable	xTaskGetWDPeriod
HeliOS.h, 39	HeliOS.h, 50
xStreamCreate	xTaskInfo
HeliOS.h, 39	HeliOS.h, 27
xStreamDelete	xTaskNotification
HeliOS.h, 40	HeliOS.h, 27
xStreamIsEmpty	xTaskNotificationIsWaiting
HeliOS.h, 40	HeliOS.h, 50
xStreamIsFull	xTaskNotifyGive
HeliOS.h, 40	HeliOS.h, 50
xStreamReceive	xTaskNotifyStateClear
HeliOS.h, 41	HeliOS.h, 51
xStreamReset	xTaskNotifyTake
HeliOS.h, 41	HeliOS.h, 51
xStreamSend	xTaskParm
HeliOS.h, 41	HeliOS.h, 27
xSystemGetSystemInfo	xTaskResetTimer
HeliOS.h, 42	HeliOS.h, 52
xSystemHalt	xTaskResume
HeliOS.h, 42	HeliOS.h, 52
xSystemInfo	xTaskResumeAll
HeliOS.h, 26	HeliOS.h, 53
xSystemInit	xTaskRunTimeStats
HeliOS.h, 42	HeliOS.h, 28
xTask	xTaskStartScheduler
HeliOS.h, 26	HeliOS.h, 53
xTaskChangePeriod	xTaskState
HeliOS.h, 42	HeliOS.h, 28
xTaskChangeWDPeriod	xTaskSuspend
HeliOS.h, 43	HeliOS.h, 53
xTaskCreate	xTaskSuspendAll
HeliOS.h, 43	HeliOS.h, 54
xTaskDelete	xTaskWait
HeliOS.h, 44	HeliOS.h, 54
xTaskGetAllRunTimeStats	xTicks
HeliOS.h, 44	HeliOS.h, 28
xTaskGetAllTaskInfo	xTimer
HeliOS.h, 45	HeliOS.h, 29
xTaskGetHandleById	xTimerChangePeriod
HeliOS.h, 45	HeliOS.h, 54
xTaskGetHandleByName	xTimerCreate
HeliOS.h, 46	HeliOS.h, 55
xTaskGetId	xTimerDelete
HeliOS.h, 46	HeliOS.h, 55
xTaskGetName	xTimerGetPeriod
HeliOS.h, 47	HeliOS.h, 55
xTaskGetNumberOfTasks	xTimerHasTimerExpired
HeliOS.h, 47	HeliOS.h, 56
xTaskGetPeriod	xTimerIsTimerActive
HeliOS.h, 47	HeliOS.h, 56

xTimerReset
 HeliOS.h, [57](#)
xTimerStart
 HeliOS.h, [57](#)
xTimerStop
 HeliOS.h, [57](#)
xWord
 HeliOS.h, [29](#)