



# Helios

Kernel 0.4.0

Helios Developer's Guide

|  |           |
|--|-----------|
| <b>1 Data Structure Index</b>                      | <b>1</b>  |
| <b>1 Data Structure Index</b>                      | <b>1</b>  |
| 1.1 Data Structures . . . . .                      | 1         |
| <b>2 File Index</b>                                | <b>1</b>  |
| 2.1 File List . . . . .                            | 1         |
| <b>3 Data Structure Documentation</b>              | <b>2</b>  |
| 3.1 MemoryRegionStats_s Struct Reference . . . . . | 2         |
| 3.2 QueueMessage_s Struct Reference . . . . .      | 2         |
| 3.3 SystemInfo_s Struct Reference . . . . .        | 2         |
| 3.4 TaskInfo_s Struct Reference . . . . .          | 3         |
| 3.5 TaskNotification_s Struct Reference . . . . .  | 3         |
| 3.6 TaskRunTimeStats_s Struct Reference . . . . .  | 3         |
| <b>4 File Documentation</b>                        | <b>3</b>  |
| 4.1 config.h File Reference . . . . .              | 3         |
| 4.1.1 Detailed Description . . . . .               | 4         |
| 4.1.2 Macro Definition Documentation . . . . .     | 5         |
| 4.2 HeliOS.h File Reference . . . . .              | 7         |
| 4.2.1 Detailed Description . . . . .               | 11        |
| 4.2.2 Function Documentation . . . . .             | 11        |
| <b>Index</b>                                       | <b>23</b> |

# 1 Data Structure Index

## 1.1 Data Structures

Here are the data structures with brief descriptions:

|                                     |          |
|-------------------------------------|----------|
| <a href="#">MemoryRegionStats_s</a> | <b>2</b> |
| <a href="#">QueueMessage_s</a>      | <b>2</b> |
| <a href="#">SystemInfo_s</a>        | <b>2</b> |
| <a href="#">TaskInfo_s</a>          | <b>3</b> |
| <a href="#">TaskNotification_s</a>  | <b>3</b> |
| <a href="#">TaskRunTimeStats_s</a>  | <b>3</b> |

## 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

3

[HeliOS.h](#)

Header file for end-user application code

7

## 3 Data Structure Documentation

### 3.1 MemoryRegionStats\_s Struct Reference

#### Data Fields

- Word\_t **largestFreeEntryInBytes**
- Word\_t **smallestFreeEntryInBytes**
- Word\_t **numberOfFreeBlocks**
- Word\_t **availableSpaceInBytes**
- Word\_t **successfulAllocations**
- Word\_t **successfulFrees**
- Word\_t **minimumEverFreeBytesRemaining**

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

- [HeliOS.h](#)

### 3.2 QueueMessage\_s Struct Reference

#### Data Fields

- Base\_t **messageBytes**
- Byte\_t **messageValue** [[CONFIG\\_MESSAGE\\_VALUE\\_BYTES](#)]

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

- [HeliOS.h](#)

### 3.3 SystemInfo\_s Struct Reference

#### Data Fields

- Byte\_t **productName** [[OS\\_PRODUCT\\_NAME\\_SIZE](#)]
- Base\_t **majorVersion**
- Base\_t **minorVersion**
- Base\_t **patchVersion**
- Base\_t **numberOfTasks**

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

- [HeliOS.h](#)

## 3.4 TaskInfo\_s Struct Reference

### Data Fields

- Base\_t **id**
- Byte\_t **name** [[CONFIG\\_TASK\\_NAME\\_BYTES](#)]
- TaskState\_t **state**
- Ticks\_t **lastRunTime**
- Ticks\_t **totalRunTime**

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

- [HeliOS.h](#)

## 3.5 TaskNotification\_s Struct Reference

### Data Fields

- Base\_t **notificationBytes**
- Byte\_t **notificationValue** [[CONFIG\\_NOTIFICATION\\_VALUE\\_BYTES](#)]

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

- [HeliOS.h](#)

## 3.6 TaskRunTimeStats\_s Struct Reference

### Data Fields

- Base\_t **id**
- Ticks\_t **lastRunTime**
- Ticks\_t **totalRunTime**

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 */`  
*Define the length of a device driver name.*

### 4.1.1 Detailed Description

#### Author

Manny Peterson ( [mannymsp@gmail.com](mailto:mannymsp@gmail.com) )

#### Version

0.4.0

#### Date

2022-01-31

#### Copyright

HeliOS Embedded Operating System Copyright (C) 2020-2023 Manny Peterson [mannymsp@gmail.com](mailto: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\_DEVICE\_NAME\_BYTES `#define CONFIG_DEVICE_NAME_BYTES 0x8u /* 8 */`

Setting CONFIG\_DEVICE\_NAME\_BYTES will define the length of a device driver name. The name of device drivers should be exactly this length. There really isn't a reason to change this and doing so may break existing device drivers. The default length is 8 bytes.

#### 4.1.2.2 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](#)

#### 4.1.2.3 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.

#### 4.1.2.4 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`

**4.1.2.5 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`

#### 4.1.2.6 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 [xQueuelQueueFull\(\)](#) 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

[xQueuelQueueFull\(\)](#)  
[xQueueSend\(\)](#)  
[xQueueCreate\(\)](#)

#### 4.1.2.7 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.

#### 4.1.2.8 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](#)

#### 4.1.2.9 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 [TaskNotification\\_s](#)
- struct [TaskRunTimeStats\\_s](#)
- struct [MemoryRegionStats\\_s](#)
- struct [TaskInfo\\_s](#)
- struct [QueueMessage\\_s](#)
- struct [SystemInfo\\_s](#)



## Typedefs

- typedef enum TaskState\_e **TaskState\_t**
- typedef TaskState\_t **xTaskState**
- typedef enum SchedulerState\_e **SchedulerState\_t**
- typedef SchedulerState\_t **xSchedulerState**
- typedef enum Return\_e **Return\_t**
- typedef Return\_t **xReturn**
- typedef enum TimerState\_e **TimerState\_t**
- typedef TimerState\_t **xTimerState**
- typedef enum DeviceState\_e **DeviceState\_t**
- typedef DeviceState\_t **xDeviceState**
- typedef enum DeviceMode\_e **DeviceMode\_t**
- typedef DeviceMode\_t **xDeviceMode**
- typedef VOID\_TYPE **TaskParm\_t**
- typedef TaskParm\_t \* **xTaskParm**
- typedef UINT8\_TYPE **Base\_t**
- typedef Base\_t **xBase**
- typedef UINT8\_TYPE **Byte\_t**
- typedef Byte\_t **xByte**
- typedef VOID\_TYPE **Addr\_t**
- typedef Addr\_t \* **xAddr**
- typedef SIZE\_TYPE **Size\_t**
- typedef Size\_t **xSize**
- typedef UINT16\_TYPE **HalfWord\_t**
- typedef HalfWord\_t **xHalfWord**
- typedef UINT32\_TYPE **Word\_t**
- typedef Word\_t **xWord**
- typedef UINT32\_TYPE **Ticks\_t**
- typedef Ticks\_t **xTicks**
- typedef VOID\_TYPE **Task\_t**
- typedef Task\_t \* **xTask**
- typedef VOID\_TYPE **Timer\_t**
- typedef Timer\_t \* **xTimer**
- typedef VOID\_TYPE **Queue\_t**
- typedef Queue\_t \* **xQueue**
- typedef VOID\_TYPE **StreamBuffer\_t**
- typedef StreamBuffer\_t \* **xStreamBuffer**
- typedef struct [TaskNotification\\_s](#) **TaskNotification\_t**
- typedef [TaskNotification\\_t](#) \* **xTaskNotification**
- typedef struct [TaskRunTimeStats\\_s](#) **TaskRunTimeStats\_t**
- typedef [TaskRunTimeStats\\_t](#) \* **xTaskRunTimeStats**
- typedef struct [MemoryRegionStats\\_s](#) **MemoryRegionStats\_t**
- typedef [MemoryRegionStats\\_t](#) \* **xMemoryRegionStats**
- typedef struct [TaskInfo\\_s](#) **TaskInfo\_t**
- typedef [TaskInfo\\_t](#) **xTaskInfo**
- typedef struct [QueueMessage\\_s](#) **QueueMessage\_t**
- typedef [QueueMessage\\_t](#) \* **xQueueMessage**
- typedef struct [SystemInfo\\_s](#) **SystemInfo\_t**
- typedef [SystemInfo\\_t](#) \* **xSystemInfo**

## Enumerations

- enum **TaskState\_e** { **TaskStateSuspended** , **TaskStateRunning** , **TaskStateWaiting** }
- enum **SchedulerState\_e** { **SchedulerStateSuspended** , **SchedulerStateRunning** }
- enum **Return\_e** { **ReturnOK** , **ReturnError** }
- enum **TimerState\_e** { **TimerStateSuspended** , **TimerStateRunning** }
- enum **DeviceState\_e** { **DeviceStateSuspended** , **DeviceStateRunning** }
- enum **DeviceMode\_e** { **DeviceModeReadOnly** , **DeviceModeWriteOnly** , **DeviceModeReadWrite** }

## Functions

- xReturn **xDeviceRegisterDevice** (xReturn(\*device\_self\_register\_>())  
*Syscall to register a device driver with the kernel.*
- xReturn **xDevicesAvailable** (const xHalfWord uid\_, xBase \*res\_)  
*Syscall to query the device driver about the availability of a device.*
- xReturn **xDeviceSimpleWrite** (const xHalfWord uid\_, xWord \*data\_)  
*Syscall to write a word of data to the device.*
- xReturn **xDeviceWrite** (const xHalfWord uid\_, xSize \*size\_, xAddr data\_)  
*Syscall to write multiple bytes of data to a device.*
- xReturn **xDeviceSimpleRead** (const xHalfWord uid\_, xWord \*data\_)  
*Syscall to read a word of data from the device.*
- xReturn **xDeviceRead** (const xHalfWord uid\_, xSize \*size\_, xAddr \*data\_)  
*Syscall to read multiple bytes from a device.*
- xReturn **xDeviceInitDevice** (const xHalfWord uid\_)  
*Syscall to initialize a device.*
- xReturn **xDeviceConfigDevice** (const xHalfWord uid\_, xSize \*size\_, xAddr config\_)  
*Syscall to configure a device.*
- xReturn **xMemAlloc** (volatile xAddr \*addr\_, const xSize size\_)  
*Syscall to request memory from the heap.*
- xReturn **xMemFree** (const volatile xAddr addr\_)  
*Syscall to free heap memory allocated by xMemAlloc()*
- xReturn **xMemGetUsed** (xSize \*size\_)  
*Syscall to obtain the amount of in-use heap memory.*
- xReturn **xMemGetSize** (const volatile xAddr addr\_, xSize \*size\_)  
*Syscall to obtain the amount of heap memory allocated at a specific address.*
- xReturn **xMemGetHeapStats** (xMemoryRegionStats \*stats\_)  
*Syscall to get memory statistics on the heap memory region.*
- xReturn **xMemGetKernelStats** (xMemoryRegionStats \*stats\_)  
*Syscall to get memory statistics on the kernel memory region.*
- xReturn **xQueueCreate** (xQueue \*queue\_, const xBase limit\_)  
*Syscall to create a message queue.*
- xReturn **xQueueDelete** (xQueue queue\_)  
*Syscall to delete a message queue.*
- xReturn **xQueueGetLength** (const xQueue queue\_, xBase \*res\_)  
*Syscall to get the length of a message queue.*
- xReturn **xQueueIsQueueEmpty** (const xQueue queue\_, xBase \*res\_)
- xReturn **xQueueIsQueueFull** (const xQueue queue\_, xBase \*res\_)
- xReturn **xQueueMessagesWaiting** (const xQueue queue\_, xBase \*res\_)
- xReturn **xQueueSend** (xQueue queue\_, const xBase bytes\_, const xByte \*value\_)
- xReturn **xQueuePeek** (const xQueue queue\_, xQueueMessage \*message\_)
- xReturn **xQueueDropMessage** (xQueue queue\_)

- xReturn **xQueueReceive** (xQueue queue\_, [xQueueMessage](#) \*message\_)
- xReturn **xQueueLockQueue** (xQueue queue\_)
- xReturn **xQueueUnLockQueue** (xQueue queue\_)
- xReturn **xStreamCreate** (xStreamBuffer \*stream\_)
- xReturn **xStreamDelete** (const xStreamBuffer stream\_)
- xReturn **xStreamSend** (xStreamBuffer stream\_, const xByte byte\_)
- xReturn **xStreamReceive** (const xStreamBuffer stream\_, xHalfWord \*bytes\_, xByte \*\*data\_)
- xReturn **xStreamBytesAvailable** (const xStreamBuffer stream\_, xHalfWord \*bytes\_)
- xReturn **xStreamReset** (const xStreamBuffer stream\_)
- xReturn **xStreamIsEmpty** (const xStreamBuffer stream\_, xBase \*res\_)
- xReturn **xStreamIsFull** (const xStreamBuffer stream\_, xBase \*res\_)
- xReturn **xSystemAssert** (const char \*file\_, const int line\_)
- xReturn **xSystemInit** (void)
- xReturn **xSystemHalt** (void)
- xReturn **xSystemGetSystemInfo** ([xSystemInfo](#) \*info\_)
- xReturn **xTaskCreate** (xTask \*task\_, const xByte \*name\_, void(\*callback\_)(xTask task\_, xTaskParm parm\_), xTaskParm taskParameter\_)
- xReturn **xTaskDelete** (const xTask task\_)
- xReturn **xTaskGetHandleByName** (xTask \*task\_, const xByte \*name\_)
- xReturn **xTaskGetHandleById** (xTask \*task\_, const xBase id\_)
- xReturn **xTaskGetAllRunTimeStats** ([xTaskRunTimeStats](#) \*stats\_, xBase \*tasks\_)
- xReturn **xTaskGetTaskRunTimeStats** (const xTask task\_, [xTaskRunTimeStats](#) \*stats\_)
- xReturn **xTaskGetNumberOfTasks** (xBase \*tasks\_)
- xReturn **xTaskGetTaskInfo** (const xTask task\_, [xTaskInfo](#) \*info\_)
- xReturn **xTaskGetAllTaskInfo** ([xTaskInfo](#) \*info\_, xBase \*tasks\_)
- xReturn **xTaskGetTaskState** (const xTask task\_, xTaskState \*state\_)
- xReturn **xTaskGetName** (const xTask task\_, xByte \*\*name\_)
- xReturn **xTaskGetId** (const xTask task\_, xBase \*id\_)
- xReturn **xTaskNotifyStateClear** (xTask task\_)
- xReturn **xTaskNotificationIsWaiting** (const xTask task\_, xBase \*res\_)
- xReturn **xTaskNotifyGive** (xTask task\_, const xBase bytes\_, const xByte \*value\_)
- xReturn **xTaskNotifyTake** (xTask task\_, [xTaskNotification](#) \*notification\_)
- xReturn **xTaskResume** (xTask task\_)
- xReturn **xTaskSuspend** (xTask task\_)
- xReturn **xTaskWait** (xTask task\_)
- xReturn **xTaskChangePeriod** (xTask task\_, const xTicks period\_)
- xReturn **xTaskChangeWDPPeriod** (xTask task\_, const xTicks period\_)
- xReturn **xTaskGetPeriod** (const xTask task\_, xTicks \*period\_)
- xReturn **xTaskResetTimer** (xTask task\_)
- xReturn **xTaskStartScheduler** (void)
- xReturn **xTaskResumeAll** (void)
- xReturn **xTaskSuspendAll** (void)
- xReturn **xTaskGetSchedulerState** (xSchedulerState \*state\_)
- xReturn **xTaskGetWDPPeriod** (const xTask task\_, xTicks \*period\_)
- xReturn **xTimerCreate** (xTimer \*timer\_, const xTicks period\_)
- xReturn **xTimerDelete** (const xTimer timer\_)
- xReturn **xTimerChangePeriod** (xTimer timer\_, const xTicks period\_)
- xReturn **xTimerGetPeriod** (const xTimer timer\_, xTicks \*period\_)
- xReturn **xTimerIsTimerActive** (const xTimer timer\_, xBase \*res\_)
- xReturn **xTimerHasTimerExpired** (const xTimer timer\_, xBase \*res\_)
- xReturn **xTimerReset** (xTimer timer\_)
- xReturn **xTimerStart** (xTimer timer\_)
- xReturn **xTimerStop** (xTimer timer\_)

### 4.2.1 Detailed Description

#### Author

Manny Peterson ( [mannymsp@gmail.com](mailto:mannymsp@gmail.com) )

#### Version

0.4.0

#### Date

2022-09-06

#### Copyright

HeliOS Embedded Operating System Copyright (C) 2020-2023 Manny Peterson [mannymsp@gmail.com](mailto: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 Function Documentation

**4.2.2.1 xDeviceConfigDevice()** `xReturn xDeviceConfigDevice (`  
`const xHalfWord uid_,`  
`xSize * size_,`  
`xAddr config_ )`

The `xDeviceConfigDevice()` will call the device driver's `DEVICENAME_config()` function to configure the device. The syscall is bi-directional (i.e., it will write configuration data to the device and read the same from the device before returning). The purpose of the bi-directional functionality is to allow the device's configuration to be set and queried using one syscall. The structure of the configuration data is left to the device driver's author. What is required is that the configuration data memory is allocated using `xMemAlloc()` and that the "size\_" parameter is set to the size (i.e., amount) of the configuration data (e.g., `sizeof(MyDeviceDriverConfig)`) in bytes.

#### Parameters

|                       |  |
|-----------------------|--|
| <code>uid_</code>     | The unique identifier ("UID") of the device driver to be operated on.                                  |
| <code>size_↔</code>   | The size (i.e., amount) of configuration data to be written and read to and from the device, in bytes. |
| <code>config_↔</code> | The configuration data. The configuration data must have been allocated by <code>xMemAlloc()</code> .  |

## Returns

**xReturn** On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return ReturnError. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be ReturnOK or ReturnError. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.2 xDeviceInitDevice()** `xReturn xDeviceInitDevice (`  
`const xHalfWord uid_ )`

The `xDeviceInitDevice()` syscall will call the device driver's `DRIVERNAME_init()` function to bootstrap the device. For example, setting memory mapped registers to starting values or setting the device driver's state and mode. This syscall is optional and is dependent on the specifics of the device driver's implementation by its author.

## Parameters

|                   |   |
|-------------------|---|
| <code>uid↔</code> | The unique identifier ("UID") of the device driver to be operated on. |
| <code>—</code>    |   |

## Returns

**xReturn** On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return ReturnError. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be ReturnOK or ReturnError. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.3 xDevicesAvailable()** `xReturn xDeviceIsAvailable (`  
`const xHalfWord uid_,`  
`xBase * res_ )`

The `xDevicesAvailable()` syscall queries the device driver about the availability of a device. Generally "available" means the that the device is available for read and/or write operations though the meaning is implementation specific and left up to the device driver's author.

## Parameters

|                   |  |
|-------------------|--|
| <code>uid↔</code> | The unique identifier ("UID") of the device driver to be operated on.          |
| <code>—</code>    |  |
| <code>res↔</code> | The result of the inquiry; here, taken to mean the availability of the device. |
| <code>—</code>    |  |

## Returns

**xReturn** On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.4 xDeviceRead()** `xReturn xDeviceRead (`  
`const xHalfWord uid_,`  
`xSize * size_,`  
`xAddr * data_ )`

The `xDeviceRead()` syscall will read multiple bytes of data from a device into a data buffer. The data buffer must be freed by `xMemFree()`. Whether the data is read from the device is dependent on the device driver mode, state and implementation of these features by the device driver's author.

## Parameters

|                          |   |
|--------------------------|---|
| <code>uid_↔</code><br>—  | The unique identifier ("UID") of the device driver to be operated on.                                     |
| <code>size_↔</code><br>— | The number of bytes read from the device and contained in the data buffer.                                |
| <code>data_↔</code><br>— | The data buffer containing the data read from the device which must be freed by <code>xMemFree()</code> . |

## Returns

**xReturn** On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.5 xDeviceRegisterDevice()** `xReturn xDeviceRegisterDevice (`  
`xReturn(*)() device_self_register_ )`

The `xDeviceRegisterDevice()` syscall is a component of the HeliOS device driver model which registers a device driver with the HeliOS kernel. This syscall must be made before a device driver can be called by `xDeviceRead()`, `xDeviceWrite()`, etc. Once a device is registered, it cannot be un-registered - it can only be placed in a suspended state which is done by calling `xDeviceConfigDevice()`. However, as with most aspects of the HeliOS device driver model, it is important to note that the implementation of and support for device state and mode is up to the device driver's author.

**Note**

A device driver's unique identifier ("UID") must be a globally unique identifier. No two device drivers in the same application can share the same UID. This is best achieved by ensuring the device driver author selects a UID for his device driver that is not in use by other device drivers. A device driver template and device drivers can be found in /drivers.

**Parameters**

|                                    |   |
|------------------------------------|---|
| <i>device_self_↔<br/>register_</i> | The device driver's self registration function, DRIVERNAME_self_register(). |
|------------------------------------|---|

**Returns**

xReturn On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if xTaskGetId() was unable to locate the task by the task object (i.e., xTask) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), xTaskGetId() would return ReturnError. All HeliOS syscalls return the xReturn (a.k.a., Return\_t) type which can either be ReturnOK or ReturnError. The C macros OK() and ERROR() can be used as a more concise way of checking the return value of a syscall (e.g., if(OK(xMem↔GetUsed(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

**4.2.2.6 xDeviceSimpleRead()** xReturn xDeviceSimpleRead (  
     const xHalfWord uid\_,  
     xWord \* data\_ )

The [xDeviceSimpleRead\(\)](#) syscall will read a word of data from a device. The word of data must be freed by [xMemFree\(\)](#). Whether the data is read from the device is dependent on the device driver mode, state and implementation of these features by the device driver's author.

**Parameters**

|                     |   |
|---------------------|---|
| <i>uid_↔<br/>—</i>  | The unique identifier ("UID") of the device driver to be operated on.                     |
| <i>data_↔<br/>—</i> | The word of data read from the device which must be freed by <a href="#">xMemFree()</a> . |

**Returns**

xReturn On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if xTaskGetId() was unable to locate the task by the task object (i.e., xTask) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), xTaskGetId() would return ReturnError. All HeliOS syscalls return the xReturn (a.k.a., Return\_t) type which can either be ReturnOK or ReturnError. The C macros OK() and ERROR() can be used as a more concise way of checking the return value of a syscall (e.g., if(OK(xMem↔GetUsed(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

**4.2.2.7 xDeviceSimpleWrite()** `xReturn xDeviceSimpleWrite (`  
`const xHalfWord uid_,`  
`xWord * data_ )`

The `xDeviceSimpleWrite()` syscall will write a word (i.e., `xWord`) of data to a device. The word of data must have been allocated by `xMemAlloc()`. Whether the data is written to the device is dependent on the device driver mode, state and implementation of these features by the device driver's author.

#### Parameters

|                          |   |
|--------------------------|---|
| <code>uid</code> ↔<br>—  | The unique identifier ("UID") of the device driver to be operated on.   |
| <code>data</code> ↔<br>— | A word of data to be written to the device. The word of data must have been allocated by <code>xMemAlloc()</code> . |

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.8 xDeviceWrite()** `xReturn xDeviceWrite (`  
`const xHalfWord uid_,`  
`xSize * size_,`  
`xAddr data_ )`

The `xDeviceWrite()` syscall will write multiple bytes of data contained in a data buffer to a device. The data buffer must have been allocated by `xMemAlloc()`. Whether the data is written to the device is dependent on the device driver mode, state and implementation of these features by the device driver's author.

#### Parameters

|                          |   |
|--------------------------|---|
| <code>uid</code> ↔<br>—  | The unique identifier ("UID") of the device driver to be operated on.   |
| <code>size</code> ↔<br>— | The size of the data buffer, in bytes.  |
| <code>data</code> ↔<br>— | The data buffer containing the data to be written to the device. The data buffer must have been allocated by <code>xMemAlloc()</code> . |

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).



**4.2.2.9 xMemAlloc()** `xReturn xMemAlloc (`  
`volatile xAddr * addr_,`  
`const xSize size_ )`

The `xMemAlloc()` syscall allocates heap memory for user's application. The amount of available heap memory is dependent on the `CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS` and `CONFIG_MEMORY_REGION_BLOCK_SIZE` settings. Similar to `libc calloc()`, `xMemAlloc()` clears (i.e., zeros out) the allocated memory it allocates. Because the address of the newly allocated heap memory is handed back through the "addr\_" argument, the argument must be cast to "volatile xAddr \*" to avoid compiler warnings.

#### Parameters

|                          |   |
|--------------------------|---|
| <code>addr_↔</code><br>— | The address of the allocated memory. For example, if heap memory for a structure called <code>mystruct</code> ( <code>MyStruct *</code> ) needs to be allocated, the call to <code>xMemAlloc()</code> would be written as follows <code>if(OK(xMemAlloc((volatile xAddr *) &amp;mystruct, sizeof(MyStruct)))) {}</code> . |
| <code>size_↔</code><br>— | The amount of heap memory, in bytes, being requested.   |

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.10 xMemFree()** `xReturn xMemFree (`  
`const volatile xAddr addr_ )`

The `xMemFree()` syscall frees (i.e., de-allocates) heap memory allocated by `xMemAlloc()`. `xMemFree()` is also used to free heap memory allocated by syscalls including `xTaskGetAllRunTimeStats()`.

#### Parameters

|                          |  |
|--------------------------|--|
| <code>addr_↔</code><br>— | The address of the allocated memory to be freed. |
|--------------------------|--|

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMemGetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.11 xMemGetHeapStats()** `xReturn xMemGetHeapStats (`  
`xMemoryRegionStats * stats_ )`

The `xMemGetHeapStats()` syscall is used to obtain detailed statistics about the heap memory region which can be used by the application to monitor memory utilization.

#### Parameters

|                      |   |
|----------------------|---|
| <code>stats_↔</code> | The memory region statistics. The memory region statistics must be freed by <code>xMemFree()</code> . |
| <code>_</code>       |   |

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HelIOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.12 xMemGetKernelStats()** `xReturn xMemGetKernelStats (`  
`xMemoryRegionStats * stats_ )`

The `xMemGetKernelStats()` syscall is used to obtain detailed statistics about the kernel memory region which can be used by the application to monitor memory utilization.

#### Parameters

|                      |   |
|----------------------|---|
| <code>stats_↔</code> | The memory region statistics. The memory region statistics must be freed by <code>xMemFree()</code> . |
| <code>_</code>       |   |

#### Returns

`xReturn` On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HelIOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.13 xMemGetSize()** `xReturn xMemGetSize (`  
`const volatile xAddr addr_,`  
`xSize * size_ )`

The `xMemGetSize()` syscall can be used to obtain the amount, in bytes, of heap memory allocated at a specific address. The address must be the address obtained from `xMemAlloc()`.

## Parameters

|                    |  |
|--------------------|--|
| <i>addr</i> ↔<br>— | The address of the heap memory for which the size (i.e., amount) allocated, in bytes, is being sought. |
| <i>size</i> ↔<br>— | The size (i.e., amount), in bytes, of heap memory allocated to the address.                            |

## Returns

**xReturn** On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.14 xMemGetUsed()** `xReturn xMemGetUsed (`  
`xSize * size_ )`

The `xMemGetUsed()` syscall will update the "size\_" argument with the amount, in bytes, of in-use heap memory. If more memory statistics are needed, `xMemGetHeapStats()` provides a more complete picture of the heap memory region.

## Parameters

|                    |   |
|--------------------|---|
| <i>size</i> ↔<br>— | The size (i.e., amount), in bytes, of in-use heap memory. |
|--------------------|---|

## Returns

**xReturn** On success, the syscall returns `ReturnOK`. On failure, the syscall returns `ReturnError`. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if `xTaskGetId()` was unable to locate the task by the task object (i.e., `xTask`) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), `xTaskGetId()` would return `ReturnError`. All HeliOS syscalls return the `xReturn` (a.k.a., `Return_t`) type which can either be `ReturnOK` or `ReturnError`. The C macros `OK()` and `ERROR()` can be used as a more concise way of checking the return value of a syscall (e.g., `if(OK(xMem↔GetUsed(&size))) {}` or `if(ERROR(xMemGetUsed(&size))) {}`).

**4.2.2.15 xQueueCreate()** `xReturn xQueueCreate (`  
`xQueue * queue_,`  
`const xBase limit_ )`

The `xQueueCreate()` syscall will create a new message queue for inter-task communication.

**Parameters**

|                     |   |
|---------------------|---|
| <i>queue</i> ↔<br>— | The message queue to be operated on.  |
| <i>limit</i> —      | The message limit for the queue. When this value is reached, the message queue is considered to be full. The minimum message limit is configured using the CONFIG_QUEUE_MINIMUM_LIMIT (default is 5) setting. |

**Returns**

xReturn On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if xTaskGetId() was unable to locate the task by the task object (i.e., xTask) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), xTaskGetId() would return ReturnError. All HeliOS syscalls return the xReturn (a.k.a., Return\_t) type which can either be ReturnOK or ReturnError. The C macros OK() and ERROR() can be used as a more concise way of checking the return value of a syscall (e.g., if(OK(xMem↔GetUsed(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

**4.2.2.16 xQueueDelete()** xReturn xQueueDelete (  
xQueue queue\_ )

The [xQueueDelete\(\)](#) syscall will delete a message queue used for inter-task communication.

**Parameters**

|                     |                                      |
|---------------------|--------------------------------------|
| <i>queue</i> ↔<br>— | The message queue to be operated on. |
|---------------------|--------------------------------------|

**Returns**

xReturn On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if xTaskGetId() was unable to locate the task by the task object (i.e., xTask) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), xTaskGetId() would return ReturnError. All HeliOS syscalls return the xReturn (a.k.a., Return\_t) type which can either be ReturnOK or ReturnError. The C macros OK() and ERROR() can be used as a more concise way of checking the return value of a syscall (e.g., if(OK(xMem↔GetUsed(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

**4.2.2.17 xQueueGetLength()** xReturn xQueueGetLength (  
const xQueue queue\_,  
xBase \* res\_ )

The [xQueueGetLength\(\)](#) syscall is used to inquire about the length (i.e., the number of messages) of a message queue.

**Parameters**

|                     |  |
|---------------------|--|
| <i>queue</i> ↔<br>_ | The message queue to be operated on.   |
| <i>res</i> _        | The result of the inquiry; taken here to mean the number of messages a message queue contains. |

**Returns**

*xReturn* On success, the syscall returns ReturnOK. On failure, the syscall returns ReturnError. A failure is any condition in which the syscall was unable to achieve its intended objective. For example, if *xTaskGetId()* was unable to locate the task by the task object (i.e., *xTask*) passed to the syscall, because either the object was null or invalid (e.g., a deleted task), *xTaskGetId()* would return ReturnError. All HelIOS syscalls return the *xReturn* (a.k.a., *Return\_t*) type which can either be ReturnOK or ReturnError. The C macros OK() and ERROR() can be used as a more concise way of checking the return value of a syscall (e.g., if(OK(*xMem*↔*GetUsed(&size)))* {} or if(ERROR(*xMem*↔*GetUsed(&size)))* {}).



## Index

config.h, [3](#)  
    CONFIG\_DEVICE\_NAME\_BYTES, [5](#)  
    CONFIG\_MEMORY\_REGION\_BLOCK\_SIZE, [5](#)  
    CONFIG\_MEMORY\_REGION\_SIZE\_IN\_BLOCKS, [5](#)  
    CONFIG\_MESSAGE\_VALUE\_BYTES, [5](#)  
    CONFIG\_NOTIFICATION\_VALUE\_BYTES, [6](#)  
    CONFIG\_QUEUE\_MINIMUM\_LIMIT, [6](#)  
    CONFIG\_STREAM\_BUFFER\_BYTES, [7](#)  
    CONFIG\_TASK\_NAME\_BYTES, [7](#)  
    CONFIG\_TASK\_WD\_TIMER\_ENABLE, [7](#)  
CONFIG\_DEVICE\_NAME\_BYTES  
    config.h, [5](#)  
CONFIG\_MEMORY\_REGION\_BLOCK\_SIZE  
    config.h, [5](#)  
CONFIG\_MEMORY\_REGION\_SIZE\_IN\_BLOCKS  
    config.h, [5](#)  
CONFIG\_MESSAGE\_VALUE\_BYTES  
    config.h, [5](#)  
CONFIG\_NOTIFICATION\_VALUE\_BYTES  
    config.h, [6](#)  
CONFIG\_QUEUE\_MINIMUM\_LIMIT  
    config.h, [6](#)  
CONFIG\_STREAM\_BUFFER\_BYTES  
    config.h, [7](#)  
CONFIG\_TASK\_NAME\_BYTES  
    config.h, [7](#)  
CONFIG\_TASK\_WD\_TIMER\_ENABLE  
    config.h, [7](#)  
  
HeliOS.h, [7](#)  
    xDeviceConfigDevice, [11](#)  
    xDeviceInitDevice, [12](#)  
    xDevicesAvailable, [12](#)  
    xDeviceRead, [13](#)  
    xDeviceRegisterDevice, [13](#)  
    xDeviceSimpleRead, [14](#)  
    xDeviceSimpleWrite, [14](#)  
    xDeviceWrite, [15](#)  
    xMemAlloc, [16](#)  
    xMemFree, [16](#)  
    xMemGetHeapStats, [17](#)  
    xMemGetKernelStats, [17](#)  
    xMemGetSize, [17](#)  
    xMemGetUsed, [19](#)  
    xQueueCreate, [19](#)  
    xQueueDelete, [20](#)  
    xQueueGetLength, [20](#)  
  
MemoryRegionStats\_s, [2](#)  
  
QueueMessage\_s, [2](#)  
  
SystemInfo\_s, [2](#)  
  
TaskInfo\_s, [3](#)  
  
TaskNotification\_s, [3](#)  
TaskRunTimeStats\_s, [3](#)  
  
xDeviceConfigDevice  
    HeliOS.h, [11](#)  
xDeviceInitDevice  
    HeliOS.h, [12](#)  
xDevicesAvailable  
    HeliOS.h, [12](#)  
xDeviceRead  
    HeliOS.h, [13](#)  
xDeviceRegisterDevice  
    HeliOS.h, [13](#)  
xDeviceSimpleRead  
    HeliOS.h, [14](#)  
xDeviceSimpleWrite  
    HeliOS.h, [14](#)  
xDeviceWrite  
    HeliOS.h, [15](#)  
xMemAlloc  
    HeliOS.h, [16](#)  
xMemFree  
    HeliOS.h, [16](#)  
xMemGetHeapStats  
    HeliOS.h, [17](#)  
xMemGetKernelStats  
    HeliOS.h, [17](#)  
xMemGetSize  
    HeliOS.h, [17](#)  
xMemGetUsed  
    HeliOS.h, [19](#)  
xQueueCreate  
    HeliOS.h, [19](#)  
xQueueDelete  
    HeliOS.h, [20](#)  
xQueueGetLength  
    HeliOS.h, [20](#)