HeliOS

Kernel 0.3.5

HeliOS Developer's Guide

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

2 File Index

2.1 File List

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

3 Data Structure Documentation

3.1 MemoryRegionStats_s Struct Reference

Data structure for memory region statistics.

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 by xMemGetHeapStats() and xMemGetKernelStats() to obtain statistics about either memory region. The MemoryRegionStats_t type should be declared as xMemoryRegion↩ Stats.

See also

xMemoryRegionStats

xMemGetHeapStats()

xMemGetKernelStats()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

See also

xMemFree()

3.1.2 Field Documentation

3.1.2.1 availableSpaceInBytes `Word_t MemoryRegionStats_s::availableSpaceInBytes`

The 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::minimumEverFreeBytes↩ Remaining`

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

3.1.2.4 numberOfFreeBlocks `Word_t MemoryRegionStats_s::numberOfFreeBlocks`

The 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 queue message.

Data Fields

- Base_t messageBytes
- Char_t messageValue [CONFIG_MESSAGE_VALUE_BYTES]

3.2.1 Detailed Description

The QueueMessage_t stucture is used to store a queue message and is returned by xQueueReceive() and xQueuePeek(). The QueueMessage_t stucture should be declared as xQueueMessage.

See also

xQueueMessage

xQueueReceive()

xQueuePeek()

CONFIG_MESSAGE_VALUE_BYTES

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The message value is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the message value.

3.2.2 Field Documentation

3.2.2.1 messageBytes `Base_t QueueMessage_s::messageBytes`

The number of bytes contained in the message value which cannot exceed CONFIG_MESSAGE_VALUE_BYTES.

3.2.2.2 messageValue `Char_t QueueMessage_s::messageValue[CONFIG_MESSAGE_VALUE_BYTES]`

The ASCII queue message value - this is *NOT* a null terminated character array.

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

- HeliOS.h

3.3 SystemInfo_s Struct Reference

Data structure for information about the HeliOS system.

Data Fields

- Char_t 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 is used to store information about the HeliOS system and is returned by xSystemGetSystemInfo(). The SystemInfo_t structure should be declared as xSystemInfo.

See also

xSystemInfo

xSystemGetSystemInfo()

OS_PRODUCT_NAME_SIZE

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The product name is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the product name.

3.3.2 Field Documentation

3.3.2.1 majorVersion Base_t SystemInfo_s::majorVersion

The SemVer major version number of HeliOS.

3.3.2.2 minorVersion Base_t SystemInfo_s::minorVersion

The SemVer minor version number of HeliOS.

3.3.2.3 numberOfTasks Base_t SystemInfo_s::numberOfTasks

The number of tasks regardless of their state.

3.3.2.4 patchVersion Base_t SystemInfo_s::patchVersion

The SemVer patch version number of HeliOS.

3.3.2.5 productName Char_t SystemInfo_s::productName[OS_PRODUCT_NAME_SIZE]

The ASCII product name of the operating system (always "HeliOS").

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_t 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 ASCII name and state. The TaskInfo_t structure is returned by xTaskGetTaskInfo() and xTaskGetAllTaskInfo(). If only runtime statistics are needed, then TaskRunTimeStats_t should be used because of its smaller memory footprint. The TaskInfo_t should be declared as xTaskInfo

See also

xTaskInfo

xTaskGetTaskInfo()

xTaskGetAllTaskInfo()

CONFIG_TASK_NAME_BYTES

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The task name is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the task name.

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 duration in ticks of the task's last runtime.

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

The ASCII name of the task which is used by xTaskGetHandleByName() to return the task handle - this is *NOT* a null terminated char array.

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 duration in ticks of the task's total runtime.

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

- HeliOS.h

3.5 TaskNotification_s Struct Reference

Data structure for a direct to task notification.

Data Fields

- Base_t notificationBytes
- Char_t notificationValue [CONFIG_NOTIFICATION_VALUE_BYTES]

3.5.1 Detailed Description

The TaskNotification_t data structure is used by xTaskNotifyGive() and xTaskNotifyTake() to send and receive direct to task notifications. Direct to task notifications are part of the event-driven multitasking model. A direct to task notifcation may be received by event-driven and co-operative tasks alike. However, the benefit of direct to task notifications may only be realized by tasks scheduled as event-driven. In order to wait for a direct to task notification, the task must be in a "waiting" state which is set by xTaskWait(). The TaskNotification_t type should be declared as xTaskNotification.

See also

xTaskNotification

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

xTaskNotifyGive()

xTaskNotifyTake()

xTaskWait()

Attention

The notification value is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the notification value.

3.5.2 Field Documentation

3.5.2.1 notificationBytes `Base_t TaskNotification_s::notificationBytes`

The length in bytes of the notification value which cannot exceed CONFIG_NOTIFICATION_VALUE_BYTES.

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

The notification value whose length is specified by the notification bytes member.

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 data structure is used by xTaskGetTaskRunTimeStats() and xTaskGetAllRuntimeStats() to obtain runtime statistics about a task. The TaskRunTimeStats_t type should be declared as xTaskRunTimeStats.

See also

xTaskRunTimeStats

xTaskGetTaskRunTimeStats()

xTaskGetAllRunTimeStats()

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

3.6.2 Field Documentation

3.6.2.1 id `Base_t TaskRunTimeStats_s::id`

The ID of the task referenced by the task handle.

3.6.2.2 lastRunTime `Ticks_t TaskRunTimeStats_s::lastRunTime`

The duration in ticks of the task's last runtime.

3.6.2.3 totalRunTime `Ticks_t TaskRunTimeStats_s::totalRunTime`

The duration in ticks of the task's total runtime.

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)

Version

0.3.5

Date

2022-01-31

Copyright

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 AVRs, 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 xQueueIsQueueFull() 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

xQueueIsQueueFull()

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

  Data structure for a direct to task notification.
- struct TaskRunTimeStats_s

  Data structure for task runtime statistics.
- struct MemoryRegionStats_s

  Data structure for memory region statistics.
- struct TaskInfo_s

  Data structure for information about a task.
- struct QueueMessage_s

  Data structure for a queue message.
- struct SystemInfo_s

  Data structure for information about the HeliOS system.

Typedefs

- typedef TaskState_t xTaskState

  Enumerated data type for task states.
- typedef SchedulerState_t xSchedulerState

  Enumerated data type for the scheduler state.
- typedef VOID_TYPE TaskParm_t

  Data type for the task paramater.
- typedef TaskParm_t * xTaskParm

  Data type for the task paramater.
- typedef UINT8_TYPE Base_t

  Data type for the base type.
- typedef Base_t xBase

  Data type for the base type.
- typedef UINT8_TYPE Byte_t

  Data type for an 8-bit wide byte.
- typedef Byte_t xByte

  Data type for an 8-bit wide byte.
- typedef VOID_TYPE Addr_t

  Data type for a pointer to an address.
- typedef Addr_t * xAddr

  Data type for a pointer to an address.
- typedef SIZE_TYPE Size_t

  Data type for the storage requirements of an object in memory.
- typedef Size_t xSize

  Data type for the storage requirements of an object in memory.
- typedef UINT16_TYPE HalfWord_t

  Data type for a 16-bit half word.
- typedef HalfWord_t xHalfWord

  Data type for a 16-bit half word.
- typedef UINT32_TYPE Word_t

  Data type for a 32-bit word.
- typedef Word_t xWord

  Data type for a 32-bit word.
- typedef UINT32_TYPE Ticks_t

Data type for system ticks.

- typedef Ticks_t xTicks

  Data type for system ticks.
- typedef UCHAR_TYPE Char_t

  Data type for a character.
- typedef Char_t xChar

  Data type for a character.
- typedef VOID_TYPE Device_t

  Data type for a device handle.
- typedef Device_t * xDevice

  Data type for a device handle.
- typedef VOID_TYPE Task_t

  Data type for a task handle.
- typedef Task_t * xTask

  Data type for a task handle.
- typedef VOID_TYPE StreamBuffer_t

  Data type for a stream buffer handle.
- typedef StreamBuffer_t * xStreamBuffer

  Data type for a stream buffer handle.
- typedef VOID_TYPE Queue_t

  Data type for a queue handle.
- typedef Queue_t * xQueue

  Data type for a queue handle.
- typedef VOID_TYPE Timer_t

  Data type for a timer handle.
- typedef Timer_t * xTimer

  Data type for a timer handle.
- typedef struct TaskNotification_s TaskNotification_t

  Data structure for a direct to task notification.
- typedef TaskNotification_t * xTaskNotification

  Data structure for a direct to task notification.
- typedef struct TaskRunTimeStats_s TaskRunTimeStats_t

  Data structure for task runtime statistics.
- typedef TaskRunTimeStats_t * xTaskRunTimeStats

  Data structure for task runtime statistics.
- typedef struct MemoryRegionStats_s MemoryRegionStats_t

  Data structure for memory region statistics.
- typedef MemoryRegionStats_t * xMemoryRegionStats

  Data structure for memory region statistics.
- typedef struct TaskInfo_s TaskInfo_t

  Data structure for information about a task.
- typedef TaskInfo_t * xTaskInfo

  Data structure for information about a task.
- typedef struct QueueMessage_s QueueMessage_t

  Data structure for a queue message.
- typedef QueueMessage_t * xQueueMessage

  Data structure for a queue message.
- typedef struct SystemInfo_s SystemInfo_t

  Data structure for information about the HeliOS system.
- typedef SystemInfo_t * xSystemInfo

  Data structure for information about the HeliOS system.

Enumerations

- enum TaskState_t { TaskStateError , TaskStateSuspended , TaskStateRunning , TaskStateWaiting }

  Enumerated data type for task states.
- enum SchedulerState_t { SchedulerStateError , SchedulerStateSuspended , SchedulerStateRunning }

  Enumerated data type for scheduler state.

Functions

- xBase xDeviceRegisterDevice (xBase(∗device_self_register_)())

  System call to register a device driver.
- xBase xDeviceIsAvailable (const xHalfWord uid_)

  System call to check if a device is available.
- xBase xDeviceSimpleWrite (const xHalfWord uid_, xWord ∗data_)

  System call to write fixed length data to a device.
- xBase xDeviceWrite (const xHalfWord uid_, xSize ∗size_, xAddr data_)

  System call to write variable length data to a device.
- xBase xDeviceSimpleRead (const xHalfWord uid_, xWord ∗data_)

  System call to read fixed length data from a device.
- xBase xDeviceRead (const xHalfWord uid_, xSize ∗size_, xAddr data_)

  System call to read variable length data from a device.
- xBase xDeviceInitDevice (const xHalfWord uid_)

  System call to initialize a device driver and its device.
- xBase xDeviceConfigDevice (const xHalfWord uid_, xSize ∗size_, xAddr config_)

  System call to configure a device driver and its device.
- xAddr xMemAlloc (const xSize size_)

  System call to allocate memory from the heap.
- void xMemFree (const volatile 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 volatile xAddr addr_)

  System call to return the amount of heap memory allcoated 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 (const xBase limit_)

  System call to create a new message queue.
- void xQueueDelete (xQueue queue_)

  System call to delete a message queue.
- xBase xQueueGetLength (const xQueue queue_)

  System call to get the length of the message queue.
- xBase xQueueIsQueueEmpty (const xQueue queue_)

  System call to check if the message queue is empty.
- xBase xQueueIsQueueFull (const xQueue queue_)

  System call to check if the message queue is full.
- xBase xQueueMessagesWaiting (const xQueue queue_)

  System call to check if there are message queue messages waiting.
- xBase xQueueSend (xQueue queue_, const xBase messageBytes_, const xChar ∗messageValue_)

System call to send a message using a message queue.

- xQueueMessage xQueuePeek (const 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.
- xStreamBuffer xStreamCreate (void)

  The xStreamCreate() system call will create a new stream buffer.
- void xStreamDelete (const xStreamBuffer stream_)

  The xStreamDelete() system call will delete a stream buffer.
- xBase xStreamSend (xStreamBuffer stream_, const xByte byte_)

  The xStreamSend() system call will write one byte to the stream buffer.
- xByte * xStreamReceive (const xStreamBuffer stream_, xHalfWord *bytes_)

  The xStreamReceive() system call will return the contents of the stream buffer.
- xHalfWord xStreamBytesAvailable (const xStreamBuffer stream_)

  The xStreamBytesAvailable() system call returns the length of the stream buffer.
- void xStreamReset (const xStreamBuffer stream_)

  The xStreamReset() system call will reset a stream buffer.
- xBase xStreamIsEmpty (const xStreamBuffer stream_)

  The xStreamIsEmpty() system call returns true if the stream buffer is empty.
- xBase xStreamIsFull (const xStreamBuffer stream_)

  The xStreamIsFull() system call returns true if the stream buffer is full.
- void xSystemInit (void)

  System call to initialize the system.
- void xSystemHalt (void)

  The xSystemHalt() system call will halt HeliOS.
- xSystemInfo xSystemGetSystemInfo (void)

  The xSystemGetSystemInfo() system call will return information about the running system.
- xTask xTaskCreate (const xChar *name_, void(*callback_)(xTask task_, xTaskParm parm_), xTaskParm taskParameter_)

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

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

  System call to get a task's handle by its ASCII name.
- xTask xTaskGetHandleById (const 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 (const 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 (const 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 (const xTask task_)

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

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

  System call to check if a direct to task notification is waiting.
- xBase xTaskNotifyGive (xTask task_, const xBase notificationBytes_, const xChar *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_, const xTicks timerPeriod_)

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

  System call to get the task timer period.
- void xTaskResetTimer (xTask task_)

  System call to reset the task timer.
- 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.
- xSchedulerState xTaskGetSchedulerState (void)

  System call to get the state of the scheduler.
- void xTaskChangeWDPeriod (xTask task_, const xTicks wdTimerPeriod_)

  The xTaskChangeWDPeriod() will change the period on the task watchdog timer.
- xTicks xTaskGetWDPeriod (const xTask task_)

  The xTaskGetWDPeriod() return the current task watchdog timer.
- xTimer xTimerCreate (const xTicks timerPeriod_)

  System call to create a new timer.
- void xTimerDelete (const xTimer timer_)

  System call will delete a timer.
- void xTimerChangePeriod (xTimer timer_, const xTicks timerPeriod_)

  System call to change the period of a timer.
- xTicks xTimerGetPeriod (const xTimer timer_)

  System call to get the period of a timer.
- xBase xTimerIsTimerActive (const xTimer timer_)

  System call to check if a timer is active.
- xBase xTimerHasTimerExpired (const 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_)

  System call to stop a timer.
- void **__SystemAssert__** (const char *file_, int line_)

4.2.1 Detailed Description

Author

Manny Peterson (mannymsp@gmail.com)

Version

0.3.5

Date

2022-09-06

Copyright

HeliOS Embedded Operating System Copyright (C) 2020-2023 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 Typedef Documentation

4.2.2.1 Addr_t `typedef VOID_TYPE Addr_t`

The Addr_t type is a pointer of type void and is used to pass addresses between the end-user application and system calls. It is not necessary to use the Addr_t ype within the end-user application as long as the type is not used to interact with the kernel through system calls. Addr_t should be declared as xAddr.

See also

xAddr

4.2.2.2 Base_t `typedef UINT8_TYPE Base_t`

The Base_t type is a simple data type often used as an argument or return type for system calls when the value is known not to exceed its 8-bit width and no data structure requirements exist. There are no guarantees the Base_t will always be 8-bits wide. If an 8-bit data type is needed that is guaranteed to remain 8-bits wide, the Byte_t data type should be used. Base_t should be declared as xBase.

See also

xBase

Byte_t

4.2.2.3 Byte_t `typedef UINT8_TYPE Byte_t`

The Byte_t type is an 8-bit wide data type and is guaranteed to always be 8-bits wide. Byte_t should be declared as xByte.

See also

xByte

4.2.2.4 Char_t `typedef UCHAR_TYPE Char_t`

The Char_t data type is used to store an 8-bit char and is typically used for char arrays for ASCII names (e.g., task name). Char_t should be declared as xChar.

See also

xChar

4.2.2.5 Device_t `typedef VOID_TYPE Device_t`

The Device_t data type is used as a device handle. The device handle is created when xDeviceRegisterDevice() is called. For more information about devices and device drivers, see xDeviceRegisterDevice() for more information. Device_t should be declared as xDevice.

See also

xDevice

xDeviceRegisterDevice()

4.2.2.6 HalfWord_t typedef UINT16_TYPE HalfWord_t

The HalfWord_t type is a 16-bit wide data type and is guaranteed to always be 16-bits wide. HalfWord_t should be declared as xHalfWord.

See also

xHalfWord

4.2.2.7 MemoryRegionStats_t typedef struct MemoryRegionStats_s MemoryRegionStats_t

The MemoryRegionStats_t data structure is used by xMemGetHeapStats() and xMemGetKernelStats() to obtain statistics about either memory region. The MemoryRegionStats_t type should be declared as xMemoryRegion↩ Stats.

See also

xMemoryRegionStats

xMemGetHeapStats()

xMemGetKernelStats()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.8 Queue_t typedef VOID_TYPE Queue_t

The Queue_t data type is used as a queue handle. The queue handle is created when xQueueCreate() is called. For more information about queues, see xQueueCreate(). Queue_t should be declared as xQueue.

See also

xQueue

xQueueCreate()

Attention

The memory referenced by the queue handle must be freed by calling xQueueDelete().

See also

xQueueDelete()

4.2.2.9 QueueMessage_t `typedef struct QueueMessage_s QueueMessage_t`

The QueueMessage_t stucture is used to store a queue message and is returned by xQueueReceive() and xQueuePeek(). The QueueMessage_t stucture should be declared as xQueueMessage.

See also

xQueueMessage

xQueueReceive()

xQueuePeek()

CONFIG_MESSAGE_VALUE_BYTES

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The message value is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the message value.

4.2.2.10 Size_t `typedef SIZE_TYPE Size_t`

The Size_t type is used for the storage requirements of an object in memory and is always represented in bytes. Size_t should be declared as xSize.

See also

xSize

4.2.2.11 StreamBuffer_t `typedef VOID_TYPE StreamBuffer_t`

The StreamBuffer_t data type is used as a stream buffer handle. The stream buffer handle is created when xStreamCreate() is called. For more information about stream buffers, see xStreamCreate(). Stream_t should be declared as xStream.

See also

xStream

xStreamCreate()

Attention

The memory referenced by the stream buffer handle must be freed by calling xStreamDelete().

See also

xStreamDelete()

4.2.2.12 SystemInfo_t `typedef struct SystemInfo_s SystemInfo_t`

The SystemInfo_t data structure is used to store information about the HeliOS system and is returned by xSystemGetSystemInfo(). The SystemInfo_t structure should be declared as xSystemInfo.

See also

xSystemInfo

xSystemGetSystemInfo()

OS_PRODUCT_NAME_SIZE

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The product name is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the product name.

4.2.2.13 Task_t `typedef VOID_TYPE Task_t`

The Task_t data type is used as a task handle. The task handle is created when xTaskCreate() is called. For more information about tasks, see xTaskCreate(). Task_t should be declared as xTask.

See also

xTask

xTaskCreate()

Attention

The memory referenced by the task handle must be freed by calling xTaskDelete().

See also

xTaskDelete()

4.2.2.14 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 ASCII name and state. The TaskInfo_t structure is returned by xTaskGetTaskInfo() and xTaskGetAllTaskInfo(). If only runtime statistics are needed, then TaskRunTimeStats_t should be used because of its smaller memory footprint. The TaskInfo_t should be declared as xTaskInfo

See also

xTaskInfo

xTaskGetTaskInfo()

xTaskGetAllTaskInfo()

CONFIG_TASK_NAME_BYTES

xMemFree()

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

The task name is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the task name.

4.2.2.15 TaskNotification_t `typedef struct TaskNotification_s TaskNotification_t`

The TaskNotification_t data structure is used by xTaskNotifyGive() and xTaskNotifyTake() to send and receive direct to task notifications. Direct to task notifications are part of the event-driven multitasking model. A direct to task notifcation may be received by event-driven and co-operative tasks alike. However, the benefit of direct to task notifications may only be realized by tasks scheduled as event-driven. In order to wait for a direct to task notification, the task must be in a "waiting" state which is set by xTaskWait(). The TaskNotification_t type should be declared as xTaskNotification.

See also

xTaskNotification

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

xTaskNotifyGive()

xTaskNotifyTake()

xTaskWait()

Attention

The notification value is *NOT* null terminated and thus Standard C Library string functions such as strcmp(), strcpy() and strlen(), which expect a null terminated char array, must not be used to manipulate the notification value.

4.2.2.16 TaskParm_t `typedef VOID_TYPE TaskParm_t`

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

See also

xTaskParm

xTaskCreate()

xMemAlloc()

xMemFree()

4.2.2.17 TaskRunTimeStats_t `typedef struct TaskRunTimeStats_s TaskRunTimeStats_t`

The TaskRunTimeStats_t data structure is used by xTaskGetTaskRunTimeStats() and xTaskGetAllRuntimeStats() to obtain runtime statistics about a task. The TaskRunTimeStats_t type should be declared as xTaskRunTimeStats.

See also

xTaskRunTimeStats

xTaskGetTaskRunTimeStats()

xTaskGetAllRunTimeStats()

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.18 Ticks_t `typedef UINT32_TYPE Ticks_t`

The Ticks_t type is used to store ticks from the system clock. Ticks is not bound to any one unit of measure for time though most systems are configured for millisecond resolution, milliseconds is not guaranteed and is dependent on the system clock frequency and prescaler. Ticks_t should be declared as xTicks.

See also

xTicks

4.2.2.19 Timer_t `typedef VOID_TYPE Timer_t`

The Timer_t data type is used as a timer handle. The timer handle is created when xTimerCreate() is called. For more information about timers, see xTimerCreate(). Timer_t should be declared as xTimer.

See also

xTimer

xTimerCreate()

Attention

The memory referenced by the timer handle must be freed by calling xTimerDelete().

See also

xTimerDelete()

4.2.2.20 Word_t `typedef UINT32_TYPE Word_t`

The Word_t type is a 32-bit wide data type and is guaranteed to always be 32-bits wide. Word_t should be declared as xWord.

See also

xWord

4.2.2.21 xAddr `typedef Addr_t* xAddr`

See also

Addr_t

4.2.2.22 xBase `typedef Base_t xBase`

See also

Base_t

4.2.2.23 xByte `typedef Byte_t xByte`

See also

Byte_t

4.2.2.24 xChar `typedef Char_t xChar`

See also

Data_t

4.2.2.25 xDevice `typedef Device_t* xDevice`

See also

Device_t

4.2.2.26 xHalfWord `typedef HalfWord_t xHalfWord`

See also

HalfWord_t

4.2.2.27 xMemoryRegionStats `typedef MemoryRegionStats_t* xMemoryRegionStats`

See also

MemoryRegionStats_t

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.28 xQueue `typedef Queue_t* xQueue`

See also

Queue_t

Attention

The memory referenced by the queue handle must be freed by calling xQueueDelete().

See also

xQueueDelete()

4.2.2.29 xQueueMessage `typedef QueueMessage_t* xQueueMessage`

See also

QueueMessage_t

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.30 xSchedulerState `typedef SchedulerState_t xSchedulerState`

See also

SchedulerState_t

4.2.2.31 xSize `typedef Size_t xSize`

See also

Size_t

4.2.2.32 xStreamBuffer `typedef StreamBuffer_t* xStreamBuffer`

See also

StreamBuffer_t

Attention

The memory referenced by the stream buffer handle must be freed by calling xStreamDelete().

See also

xStreamDelete()

4.2.2.33 xSystemInfo `typedef SystemInfo_t* xSystemInfo`

See also

SystemInfo_t

Attention

The memory allocated for the data structure must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.34 xTask `typedef Task_t* xTask`

See also

Task_t

Attention

The memory referenced by the task handle must be freed by calling xTaskDelete().

See also

xTaskDelete()

4.2.2.35 xTaskInfo `typedef TaskInfo_t* xTaskInfo`

See also

TaskInfo_t

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.36 xTaskNotification `typedef TaskNotification_t* xTaskNotification`

See also

TaskNotification_t

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.37 xTaskParm `typedef TaskParm_t* xTaskParm`

See also

TaskParm_t

4.2.2.38 xTaskRunTimeStats `typedef TaskRunTimeStats_t* xTaskRunTimeStats`

See also

TaskRunTimeStats_t

Attention

The memory allocated for the data struture must be freed by calling xMemFree().

See also

xMemFree()

4.2.2.39 xTaskState `typedef TaskState_t xTaskState`

See also

TaskState_t

4.2.2.40 xTicks `typedef Ticks_t xTicks`

See also

Ticks_t

4.2.2.41 xTimer `typedef Timer_t* xTimer`

See also

Timer_t

Attention

The memory referenced by the timer handle must be freed by calling xTimerDelete().

See also

xTimerDelete()

4.2.2.42 xWord `typedef Word_t xWord`

See also

Word_t

4.2.3 Enumeration Type Documentation

4.2.3.1 SchedulerState_t `enum SchedulerState_t`

The scheduler can be in one of three possible states as defined by the SchedulerState_t enumerated data type. The state the scheduler is in is changed by calling xTaskSuspendAll() and xTaskResumeAll(). The state the scheduler is in can be obtained by calling xTaskGetSchedulerState(). SchedulerState_t should be declared (i.e., used) as xSchedulerState.

See also

xSchedulerState

xTaskSuspendAll()

xTaskResumeAll()

xTaskGetSchedulerState()

xTaskStartScheduler()

Enumerator

SchedulerStateError	Not used - reserved for future use.
SchedulerStateSuspended	State the scheduler is in after calling xTaskSuspendAll() - xTaskStartScheduler() will stop scheduling tasks for execution and relinquish control when xTaskSuspendAll() is called.
SchedulerStateRunning	State the scheduler is in after calling xTaskResumeAll() - xTaskStartScheduler() will continue to schedule tasks for execution until xTaskSuspendAll() is called.

4.2.3.2 TaskState_t `enum TaskState_t`

A task can be in one of four possible states as defined by the TaskState_t enumerated data type. The state a task is in is changed by calling xTaskResume(), xTaskSuspend() or xTaskWait(). The HeliOS scheduler will only schedule, for execution, tasks in either the TaskStateRunning or TaskStateWaiting state. TaskState_t should be declared (i.e., used) as xTaskState.

See also

xTaskState

xTaskResume()

xTaskSuspend()

xTaskWait()

xTaskGetTaskState()

Enumerator

TaskStateError	Returned by xTaskGetTaskState() when the task cannot be found.
TaskStateSuspended	State a task is in when it is first created OR after calling xTaskSuspend() - tasks in the TaskStateSuspended state will not be scheduled for execution.
TaskStateRunning	State a task is in after calling xTaskResume() - tasks in the TaskStateRunning state will be scheduled co-operatively.
TaskStateWaiting	State a task is in after calling xTaskWait() - tasks in the TaskStateWaiting state will be scheduled as event driven.

4.2.4 Function Documentation

4.2.4.1 xDeviceConfigDevice()

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

The xDeviceConfigDevice() system call will configure the device driver and its device. Like most aspects of the HeliOS device driver model, the implementation of this feature is dependent on the author of the device driver. Some device drivers may not implement this feature of the device driver model at all. A device driver template and pre-packaged drivers can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.
*size_*	The number of bytes (i.e., size) of the device configuration data structure.
*config_*	A pointer to the size_ bytes of memory occupied by the configuration data structure - the memory *MUST* be located in the heap memory region.

Returns

xBase If configuration of the device driver was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.2 xDeviceInitDevice()

```
xBase xDeviceInitDevice (
        const xHalfWord uid_ )
```

The xDeviceInitDevice() system call will initialize the device driver and its device. Like most aspects of the HeliOS device driver model, the implementation of this feature is dependent on the author of the device driver. Some device drivers may not implement this feature of the device driver model at all. A device driver template and pre-packaged drivers can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.

Returns

xBase If the initialization of the device driver was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.3 xDeviceIsAvailable()

```
xBase xDeviceIsAvailable (
        const xHalfWord uid_ )
```

The xDeviceIsAvailable() system call checks to see if a device driver is "available", generally for a read, write or read/write operation. What "available" means is up to the device driver author and may change based on what mode the device driver is in. For example, if the device driver is in read-only mode, then xDeviceIsAvailable() may return "true" if data is ready to be read from the device. A device driver template and pre-packaged device drivers can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.

Returns

xBase If the device driver is "available", then "true" is returned. Otherwise "false" is returned.

4.2.4.4 xDeviceRead()

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

The xDeviceRead() system call will read variable length data from a device driver. Whether the data is read is dependent on the device driver mode, state and implementation of these features by the device driver author. A device driver template and pre-packaged drivers can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.
*size_*	The number of bytes read from the device.
*data_*	A pointer to the data buffer to read the data into from the device - because xDeviceRead() uses variable length data, the pointer must reference size_ bytes of memory and the memory *MUST* be located in the heap memory region.

Returns

xBase If the read operation was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.5 xDeviceRegisterDevice()

```
xBase xDeviceRegisterDevice (
        xBase(*)() device_self_register_ )
```

The xDeviceRegisterDevice() system call, as part of the HeliOS device driver model, registers a device driver with the HeliOS kernel. This system call must be made before a device driver can be called by xDeviceRead(), xDeviceWrite(), etc. If the device driver is successfully registered with the kernel, xDeviceRegisterDevice() will return RETURN_SUCCESS. 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 device driver model in HeliOS, it is important to note that the implementation of the device state and mode is up to the device driver author. A quick word about device driver unique identifiers (uid). A device driver 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 another device driver. A device driver template and pre-packaged device drivers can be found in /drivers.

Parameters

*device_self_register_*	A pointer to the self registration function for the device driver.

Returns

xBase If the device driver is successfully registered with the kernel, xDeviceRegisterDevice() returns RETURN_SUCCESS. Otherwise RETURN_FAILURE is returned..

4.2.4.6 xDeviceSimpleRead()

```
xBase xDeviceSimpleRead (
        const xHalfWord uid_,
        xWord * data_ )
```

The xDeviceSimpleRead() system call will read fixed length (one word) data from a device driver. Whether the data is read is dependent on the device driver mode, state and implementation of these features by the device driver author. A device driver template and pre-packaged device drivers can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.
*data_*	A pointer to the data buffer to read the data into from the device - because xDeviceSimpleRead() uses fixed length data, the pointer must reference a word of memory and the memory *MUST* be located in the heap memory region.

Returns

xBase If the read operation was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.7 xDeviceSimpleWrite()

```
xBase xDeviceSimpleWrite (
        const xHalfWord uid_,
        xWord * data_ )
```

The xDeviceSimpleWrite() system call will write fixed length (one word) data to a device driver. Whether the data is written is dependent on the device driver mode, state and implementation of these features by the device driver author. A device driver template and pre-packaged device drives can be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver
*data_*	A pointer to the data to be written to the device - because xDeviceSimpleWrite() uses fixed length data, the pointer must reference a word of memory and the memory *MUST* be located in the heap memory region.

Returns

xBase If the write operation was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.8 xDeviceWrite()

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

The xDeviceWrite() system call will write variable length data to a device driver. Whether the data is written is dependent on the device driver mode, state and implementation of these features by the device driver author. A device driver template and pre-packaged drivers an be found in /drivers.

Parameters

*uid_*	The unique identifier of the device driver.
*size_*	The length (i.e., size) of the data to be written to the device.
*data_*	A pointer to the data to be written to the device - because xDeviceWrite uses variable length data, the pointer must reference size_ bytes of memory and the memory *MUST* be located in the heap memory region.

Returns

xBase If the write operation was successful, RETURN_SUCCESS is returned. Otherwise RETURN_FAILURE is returned.

4.2.4.9 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

*size_*	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.

4.2.4.10 xMemFree() `void xMemFree ( const volatile 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

*addr_*	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 (e.g., xTaskDelete()).

4.2.4.11 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().

4.2.4.12 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().

4.2.4.13 xMemGetSize() xSize xMemGetSize (const volatile 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

*addr_*	The address of the allocated heap memory to obtain the size of the memory, in bytes, that is allocated.

Returns

xSize 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 base 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.

4.2.4.14 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

xSize 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()).

4.2.4.15 xQueueCreate()

```
xQueue xQueueCreate (
        const xBase limit_ )
```

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

See also

xQueue

xQueueDelete()

CONFIG_QUEUE_MINIMUM_LIMIT

Parameters

*limit_*	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().

4.2.4.16 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

queue↩ _	The queue to be deleted.

4.2.4.17 xQueueDropMessage() `void xQueueDropMessage (`

`xQueue queue_ )`

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

Parameters

queue↩ _	The queue to drop the next message from.

4.2.4.18 xQueueGetLength() `xBase xQueueGetLength (`

`const xQueue queue_ )`

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

Parameters

queue↩ _	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.

4.2.4.19 xQueueIsQueueEmpty() `xBase xQueueIsQueueEmpty (`

`const xQueue queue_ )`

The xQueueIsEmpty() 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

queue↩ _	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.

4.2.4.20 xQueueIsQueueFull() `xBase xQueueIsQueueFull ( const xQueue queue_ )`

The xQueueIsFull() 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

*queue_*	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.

4.2.4.21 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

*queue_*	The queue to lock.

4.2.4.22 xQueueMessagesWaiting() `xBase xQueueMessagesWaiting ( const xQueue queue_ )`

The xQueueMessageWaiting() 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

*queue_*	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.

4.2.4.23 xQueuePeek() `xQueueMessage xQueuePeek ( const 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

*queue_*	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().

4.2.4.24 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

*queue_*	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().

4.2.4.25 xQueueSend()

```
xBase xQueueSend (
        xQueue queue_,
        const xBase messageBytes_,
        const xChar * 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

*queue_*	The queue to send the message to.
*messageBytes_*	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.
*messageValue_*	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.

4.2.4.26 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

*queue_*	The queue to unlock.

4.2.4.27 xStreamBytesAvailable() `xHalfWord xStreamBytesAvailable ( const 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

*stream_*	The stream to operate on.

Returns

xHalfWord The length of the stream buffer in bytes.

4.2.4.28 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().

4.2.4.29 xStreamDelete() `void xStreamDelete ( const 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

*stream_*	The stream buffer to operate on.

4.2.4.30 xStreamIsEmpty() `xBase xStreamIsEmpty (`

```
const 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

*stream_*	The stream buffer to operate on.

Returns

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

4.2.4.31 xStreamIsFull() `xBase xStreamIsFull (`

```
const xStreamBuffer stream_ )
```

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

*stream_*	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.

4.2.4.32 xStreamReceive() `xByte* xStreamReceive (`

```
const xStreamBuffer stream_,
xHalfWord * 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

*stream_*	The stream to operate on.
*bytes_*	The number of bytes returned (i.e., length of the byte array) by xStreamReceive().

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().

4.2.4.33 xStreamReset()

```
void xStreamReset (
    const xStreamBuffer stream_ )
```

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

Parameters

*stream_*	The stream buffer to operate on.

4.2.4.34 xStreamSend()

```
xBase xStreamSend (
    xStreamBuffer stream_,
    const 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

*stream_*	The stream buffer to operate on.
*byte_*	The byte to be sent to the stream buffer.

Returns

xBase Returns RETURN_SUCCESS if the byte was successfully written to the stream buffer. Otherwise, returns RETURN_FAILURE.

4.2.4.35 xSystemGetSystemInfo()

```
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.

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().

4.2.4.36 xSystemHalt()

```
void xSystemHalt (
        void  )
```

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

4.2.4.37 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.

4.2.4.38 xTaskChangePeriod()

```
void xTaskChangePeriod (
        xTask task_,
        const 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

*task_*	The task to change the timer period for.
*timer↩ Period_*	The timer period in ticks.

4.2.4.39 xTaskChangeWDPeriod() `void xTaskChangeWDPeriod (`

```
xTask task_,
const 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

*task_*	The task to change the task watchdog timer for.
*wdTimer↩ Period_*	The task watchdog timer period which is measured in ticks. If zero, the task watchdog timer will not have any effect.

4.2.4.40 xTaskCreate() `xTask xTaskCreate (`

```
const xChar * name_,
void(*)(xTask task_, xTaskParm parm_) 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

*name_*	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.
*callback_*	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.
*task↩ Parameter_*	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().

4.2.4.41 xTaskDelete() `void xTaskDelete ( const 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

*task_*	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).

4.2.4.42 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

*tasks_*	A variable of type xBase 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().

4.2.4.43 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

*tasks_*	A variable of type xBase 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().

4.2.4.44 xTaskGetHandleById()

```
xTask xTaskGetHandleById (
        const xBase id_ )
```

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

See also

xBase

Parameters

*id_*	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.

4.2.4.45 xTaskGetHandleByName()

```
xTask xTaskGetHandleByName (
        const xChar * 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

*name_*	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.

4.2.4.46 xTaskGetId()

```
xBase xTaskGetId (
        const xTask task_ )
```

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

Parameters

*task_*	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).

4.2.4.47 xTaskGetName() xChar* xTaskGetName (const 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

*task_*	The task to return the name of.

Returns

xChar* 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().

4.2.4.48 xTaskGetNumberOfTasks() xBase xTaskGetNumberOfTasks (void)

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

Returns

xBase The number of tasks.

4.2.4.49 xTaskGetPeriod() xTicks xTaskGetPeriod (const 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

*task_*	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.

4.2.4.50 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.

4.2.4.51 xTaskGetTaskInfo()

```
xTaskInfo xTaskGetTaskInfo (
        const 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

*task_*	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().

4.2.4.52 xTaskGetTaskRunTimeStats() `xTaskRunTimeStats xTaskGetTaskRunTimeStats ( const 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

*task_*	The task to get the runtime statistics for.

Returns

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

Warning

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

4.2.4.53 xTaskGetTaskState() `xTaskState xTaskGetTaskState ( const xTask task_ )`

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

See also

xTaskState

Parameters

*task_*	The task to return the state of.

Returns

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

4.2.4.54 xTaskGetWDPeriod()

```
xTicks xTaskGetWDPeriod (
    const xTask task_ )
```

The xTaskGetWDPeriod() will return the current task watchdog timer for the task.

See also

xTaskChangeWDPeriod()

Parameters

*task_*	The task to get the task watchdog timer period for.

Returns

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

4.2.4.55 xTaskNotificationIsWaiting()

```
xBase xTaskNotificationIsWaiting (
    const 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

*task_*	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.

4.2.4.56 xTaskNotifyGive()

```
xBase xTaskNotifyGive (
    xTask task_,
    const xBase notificationBytes_,
    const xChar * 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

*task_*	The task to send the task notification to.
*notification↩ Bytes_*	The number of bytes contained in the notification value. The number must be between one and the CONFIG_NOTIFICATION_VALUE_BYTES setting.
*notification↩ Value_*	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.

4.2.4.57 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

task↩ _	The task to clear the notification for.

4.2.4.58 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

*task_*	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().

4.2.4.59 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

*task_*	The task to reset the task timer for.

4.2.4.60 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

task↩ _	The task to set its state to running.

4.2.4.61 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()

4.2.4.62 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.

4.2.4.63 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

task↩ _	The task to suspend.

4.2.4.64 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()

4.2.4.65 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

*task_*	The task to place in the waiting state.

4.2.4.66 xTimerChangePeriod() `void xTimerChangePeriod (`

```
xTimer timer_,
const 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

*timer_*	The timer to change the period for.
*timerPeriod_*	The timer period in is ticks. Timer period must be zero or greater.

4.2.4.67 xTimerCreate() `xTimer xTimerCreate (`

```
        const 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

*timer↩ Period_*	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().

4.2.4.68 xTimerDelete() `void xTimerDelete (`

```
        const xTimer timer_ )
```

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

See also

xTimerCreate()

Parameters

timer↩ _	The timer to be deleted.

4.2.4.69 xTimerGetPeriod() `xTicks xTimerGetPeriod (`

```
        const xTimer timer_ )
```

The xTimerGetPeriod() system call will return the current timer period for the specified timer.

Parameters

*timer_*	The timer to get the timer period for.

Returns

xTicks The timer period. If the timer cannot be found, xTimerGetPeriod() will return zero.

4.2.4.70 xTimerHasTimerExpired()

```
xBase xTimerHasTimerExpired (
        const 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

*timer_*	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.

4.2.4.71 xTimerIsTimerActive()

```
xBase xTimerIsTimerActive (
        const xTimer timer_ )
```

The xTimerIsTimerActive() system call will return true of the timer has been started with xTimerStart().

See also

xTimerStart()

Parameters

*timer_*	The timer to check if active.

Returns

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

4.2.4.72 xTimerReset() `void xTimerReset (`

`xTimer timer_ )`

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

Parameters

*timer_*	The timer to be reset.

4.2.4.73 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

*timer_*	The timer to be started.

4.2.4.74 xTimerStop() `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().

See also

xTimerStart()

xTimerReset()

Parameters

timer↩ _	The timer to be stopped.

Index