HeliOS

Kernel 0.4.0

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 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)

Version

0.4.0

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
- 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 xDeviceIsAvailable (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_)

  Syscall to inquire as to whether a message queue is empty.
- xReturn xQueueIsQueueFull (const xQueue queue_, xBase ∗res_)

  Syscall to inquire as to whether a message queue is full.
- xReturn xQueueMessagesWaiting (const xQueue queue_, xBase ∗res_)

  Syscall to inquire as to whether a message queue has one or more messages waiting.

- 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 **xTaskChangeWDPeriod** (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 **xTaskGetWDPeriod** (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)

Version

0.4.0

Date

2022-09-06

Copyright

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

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

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))) {}).

4.2.2.2 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

uid↩ _	The unique identifier ("UID") of the device driver 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))) {}).

4.2.2.3 xDeviceIsAvailable()

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

The xDeviceIsAvailable() 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

uid↩ _	The unique identifier ("UID") of the device driver to be operated on.
res↩ _	The result of the inquiry; here, taken to mean the availability of the device.

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))) {}).

4.2.2.4 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

uid↩ _	The unique identifier ("UID") of the device driver to be operated on.
size↩ _	The number of bytes read from the device and contained in the data buffer.
data↩ _	The data buffer containing the data read from the device which must be freed by xMemFree().

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))) {}).

4.2.2.5 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

*device_self_↩ register_*	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

uid↩ _	The unique identifier ("UID") of the device driver to be operated on.
data↩ _	The word of data read from the device which must be fred by xMemFree().

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

*uid_*	The unique identifier ("UID") of the device driver to be operated on.
*data_*	A word of data to be written to the device. The word of data must have been allocated by xMemAlloc().

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))) {}).

4.2.2.8 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

*uid_*	The unique identifier ("UID") of the device driver to be operated on.
*size_*	The size of the data buffer, in bytes.
*data_*	The data buffer containing the data to be written to the device. The data buffer must have been allocated by xMemAlloc().

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))) {}).

4.2.2.9 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

addr↩ _	The address of the allocated memory. For example, if heap memory for a structure called mystruct (MyStruct ∗) needs to be allocated, the call to xMemAlloc() would be written as follows if(OK(xMemAlloc((volatile xAddr ∗) &mystruct, sizeof(MyStruct)))) {}.
size↩ _	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(xMem↩ GetUsed(&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

addr↩ _	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(xMem↩ GetUsed(&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

*stats_*	The memory region statistics. The memory region statistics must be freed by xMemFree().

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))) {}).

4.2.2.12 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

*stats_*	The memory region statistics. The memory region statistics must be freed by xMemFree().

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))) {}).

4.2.2.13 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

*addr_*	The address of the heap memory for which the size (i.e., amount) allocated, in bytes, is being sought.
*size_*	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(xMemGetUsed(&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

*size_*	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(xMemGetUsed(&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

*queue_*	The message queue to be operated on.
*limit_*	The message limit for the queue. When this value is reached, the message queue is considered to be full. The minimume 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(xMemGetUsed(&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

*queue_*	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(xMemGetUsed(&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

*queue_*	The message queue to be operated on.
*res_*	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(xMemGetUsed(&size))) {}).

4.2.2.18 xQueueIsQueueEmpty()

```
xReturn xQueueIsQueueEmpty (
    const xQueue queue_,
    xBase * res_ )
```

The xQueueIsQueueEmpty() syscall is used to inquire as to whether a message queue is empty. A message queue is considered empty if the length (i.e., number of messages) of a queue is zero.

Parameters

*queue_*	The message queue to be operated on.
*res_*	The result of the inquiry; taken here to mean "true" if the queue is empty, "false" if it contains one or more messages.

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))) {}).

4.2.2.19 xQueueIsQueueFull()

```
xReturn xQueueIsQueueFull (
    const xQueue queue_,
    xBase * res_ )
```

The xQueueIsQueueFull() syscall is used to inquire as to whether a message queue is full. A message queue is considered full if the length (i.e., number of messages) of a queue has reached its message limit which is configured using the CONFIG_QUEUE_MINIMUM_LIMIT (default is 5) setting.

Parameters

*queue_*	The message queue to be operated on.
*res_*	The result of the inquiry; taken here to mean "true" if the queue is full, "false" if it contains less than "limit" messages.

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))) {}).

4.2.2.20 xQueueMessagesWaiting()

```
xReturn xQueueMessagesWaiting (
    const xQueue queue_,
    xBase * res_ )
```

The xQueueMessagesWaiting() syscall is used to inquire as to whether a message queue has one or more messages waiting.

Parameters

*queue_*	The message queue to be operated on.
*res_*	The result of the inquiry; taken here to mean "true" if there is one or more messages waiting.

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))) {}).

Index