HeliOS

Kernel 0.5.0

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 Dir_s Struct Reference

Data structure for directory handle.

Data Fields

- struct Volume_s * **volume**
- xWord **currentCluster**
- xHalfWord **entryIndex**
- xBase **isOpen**

3.1.1 Detailed Description

See also

xDir
xDirOpen()
xDirClose()

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

- HeliOS.h

3.2 DirEntry_s Struct Reference

Data structure for directory entry information.

Data Fields

- xByte **name** [256]
- xWord **size**
- xWord **firstCluster**
- xBase **isDirectory**
- xBase **isReadOnly**
- xBase **isHidden**
- xBase **isSystem**

3.2.1 Detailed Description

See also

xDirEntry
xDirRead()
xFileGetInfo()

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

- HeliOS.h

3.3 File_s Struct Reference

Data structure for file handle.

Data Fields

- struct Volume_s * **volume**
- xWord **firstCluster**
- xWord **currentCluster**
- xWord **fileSize**
- xWord **position**
- xByte **mode**
- xBase **isOpen**
- xBase **isDirty**

3.3.1 Detailed Description

See also

xFile
xFileOpen()
xFileClose()

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

- HeliOS.h

3.4 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

3.4.1 Detailed Description

The MemoryRegionStats_t data structure is used by xMemGetHeapStats() and xMemGetKernelStats() to obtain statistics about either memory region.

See also

xMemoryRegionStats

xMemGetHeapStats()

xMemGetKernelStats()

xMemFree()

3.4.2 Field Documentation

3.4.2.1 availableSpaceInBytes `Word_t MemoryRegionStats_s::availableSpaceInBytes`

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

3.4.2.2 largestFreeEntryInBytes `Word_t MemoryRegionStats_s::largestFreeEntryInBytes`

The largest free entry in bytes.

3.4.2.3 minimumEverFreeBytesRemaining `Word_t MemoryRegionStats_s::minimumEverFreeBytesRemaining`

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

3.4.2.4 numberOfFreeBlocks `Word_t MemoryRegionStats_s::numberOfFreeBlocks`

The number of free blocks. See CONFIG_MEMORY_REGION_BLOCK_SIZE for block size in bytes.

3.4.2.5 smallestFreeEntryInBytes `Word_t MemoryRegionStats_s::smallestFreeEntryInBytes`

The smallest free entry in bytes.

3.4.2.6 successfulAllocations `Word_t MemoryRegionStats_s::successfulAllocations`

Number of successful memory allocations.

3.4.2.7 successfulFrees `Word_t MemoryRegionStats_s::successfulFrees`

Number of successful memory "frees".

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

- HeliOS.h

3.5 QueueMessage_s Struct Reference

Data structure for a queue message.

Data Fields

- Base_t messageBytes
- Byte_t messageValue [0x8u]

3.5.1 Detailed Description

The QueueMessage_t stucture is used to store a queue message and is returned by xQueueReceive() and xQueuePeek().

See also

xQueueMessage

xQueueReceive()

xQueuePeek()

CONFIG_MESSAGE_VALUE_BYTES

xMemFree()

3.5.2 Field Documentation

3.5.2.1 messageBytes `Base_t QueueMessage_s::messageBytes`

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

3.5.2.2 messageValue `Byte_t QueueMessage_s::messageValue[0x8u]`

The queue message value.

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

- HeliOS.h

3.6 SystemInfo_s Struct Reference

Data structure for information about the HeliOS system.

Data Fields

- Byte_t productName [0x6u]
- Base_t majorVersion
- Base_t minorVersion
- Base_t patchVersion
- Base_t numberOfTasks
- Base_t littleEndian

3.6.1 Detailed Description

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

See also

xSystemInfo
xSystemGetSystemInfo()
OS_PRODUCT_NAME_SIZE
xMemFree()

3.6.2 Field Documentation

3.6.2.1 littleEndian `Base_t SystemInfo_s::littleEndian`

True if the system byte order is little endian.

3.6.2.2 majorVersion `Base_t SystemInfo_s::majorVersion`

The SemVer major version number of HeliOS.

3.6.2.3 minorVersion Base_t SystemInfo_s::minorVersion

The SemVer minor version number of HeliOS.

3.6.2.4 numberOfTasks Base_t SystemInfo_s::numberOfTasks

The number of tasks regardless of their state.

3.6.2.5 patchVersion Base_t SystemInfo_s::patchVersion

The SemVer patch version number of HeliOS.

3.6.2.6 productName Byte_t SystemInfo_s::productName[0x6u]

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

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

- HeliOS.h

3.7 TaskInfo_s Struct Reference

Data structure for information about a task.

Data Fields

- Base_t id
- Byte_t name [0x8u]
- TaskState_t state
- Ticks_t lastRunTime
- Ticks_t totalRunTime

3.7.1 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 name and state. The TaskInfo_t structure is returned by xTaskGetTaskInfo() and xTaskGetAllTaskInfo(). If only runtime statistics are needed, then TaskRun↩ TimeStats_t should be used because of its smaller memory footprint.

See also

xTaskInfo

xTaskGetTaskInfo()

xTaskGetAllTaskInfo()

CONFIG_TASK_NAME_BYTES

xMemFree()

3.7.2 Field Documentation

3.7.2.1 id `Base_t TaskInfo_s::id`

The ID of the task.

3.7.2.2 lastRunTime `Ticks_t TaskInfo_s::lastRunTime`

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

3.7.2.3 name `Byte_t TaskInfo_s::name[0x8u]`

The name of the task which must be exactly CONFIG_TASK_NAME_BYTES bytes in length. Shorter task names must be padded.

3.7.2.4 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.

3.7.2.5 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

3.8 TaskNotification_s Struct Reference

Data structure for a direct to task notification.

Data Fields

- Base_t notificationBytes
- Byte_t notificationValue [0x8u]

3.8.1 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 notification 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().

See also

xTaskNotification

xMemFree()

xTaskNotifyGive()

xTaskNotifyTake()

xTaskWait()

3.8.2 Field Documentation

3.8.2.1 notificationBytes `Base_t TaskNotification_s::notificationBytes`

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

3.8.2.2 notificationValue `Byte_t TaskNotification_s::notificationValue[0x8u]`

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

3.9 TaskRunTimeStats_s Struct Reference

Data structure for task runtime statistics.

Data Fields

- Base_t id
- Ticks_t lastRunTime
- Ticks_t totalRunTime

3.9.1 Detailed Description

The TaskRunTimeStats_t data structure is used by xTaskGetTaskRunTimeStats() and xTaskGetAllRuntimeStats() to obtain runtime statistics about a task.

See also

xTaskRunTimeStats

xTaskGetTaskRunTimeStats()

xTaskGetAllRunTimeStats()

xMemFree()

3.9.2 Field Documentation

3.9.2.1 id `Base_t TaskRunTimeStats_s::id`

The ID of the task.

3.9.2.2 lastRunTime `Ticks_t TaskRunTimeStats_s::lastRunTime`

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

3.9.2.3 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

3.10 Volume_s Struct Reference

Data structure for FAT32 volume metadata.

Data Fields

- xHalfWord **blockDeviceUID**
- xWord **fatStartSector**
- xWord **dataStartSector**
- xWord **rootDirCluster**
- xByte **sectorsPerCluster**
- xHalfWord **bytesPerSector**
- xHalfWord **reservedSectors**
- xByte **numFATs**
- xWord **sectorsPerFAT**
- xBase **mounted**

3.10.1 Detailed Description

See also

xVolume
xFSMount()
xFSUnmount()

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

- HeliOS.h

3.11 VolumeInfo_s Struct Reference

Data structure for volume information.

Data Fields

- xWord **totalClusters**
- xWord **freeClusters**
- xWord **totalBytes**
- xWord **freeBytes**
- xHalfWord **bytesPerSector**
- xByte **sectorsPerCluster**
- xWord **bytesPerCluster**

3.11.1 Detailed Description

See also

xVolumeInfo
xFSGetVolumeInfo()

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

- HeliOS.h

4 File Documentation

4.1 config.h File Reference

Kernel source for build configuration.

Macros

- #define CONFIG_ENABLE_ARDUINO_CPP_INTERFACE

  Define to enable the Arduino API C++ interface.
- #define CONFIG_ENABLE_SYSTEM_ASSERT

  Define to enable system assertions.
- #define CONFIG_SYSTEM_ASSERT_BEHAVIOR(f, l) __ArduinoAssert__(f, l)

  Define the system assertion behavior.
- #define CONFIG_MESSAGE_VALUE_BYTES 0x8u /* 8 */

  Define the size in bytes of the message queue message value.
- #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 task name.
- #define CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS 0x10u /* 16 */

  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 manny@heliosproj.org

Version

0.5.0

Date

2023-03-19

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_ENABLE_ARDUINO_CPP_INTERFACE `#define CONFIG_ENABLE_ARDUINO_CPP_INTERFACE`

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.

4.1.2.3 CONFIG_ENABLE_SYSTEM_ASSERT `#define CONFIG_ENABLE_SYSTEM_ASSERT`

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

4.1.2.4 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.

See also

xMemAlloc()

xMemFree()

CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS

4.1.2.5 CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS `#define CONFIG_MEMORY_REGION_SIZE_IN_↩ BLOCKS 0x10u /* 16 */`

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. The default value is 16 blocks.

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

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.

See also

xQueueMessage

4.1.2.7 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.

See also

xTaskNotification

4.1.2.8 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.

See also

xQueueIsQueueFull()

xQueueSend()

xQueueCreate()

4.1.2.9 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.

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

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 )
```

4.1.2.11 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 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.

See also

xTaskInfo

4.1.2.12 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

Public API header for HeliOS embedded operating system applications.

Data Structures

- struct Volume_s

  Data structure for FAT32 volume metadata.
- struct File_s

  Data structure for file handle.
- struct Dir_s

  Data structure for directory handle.
- struct DirEntry_s

  Data structure for directory entry information.
- struct VolumeInfo_s

  Data structure for volume information.
- 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.

Macros

- #define **DEREF_TASKPARM**(type_, ptr_) (∗((type_ ∗) (ptr_)))

Typedefs

- typedef enum TaskState_e TaskState_t

  Enumerated type defining the possible states of a task.
- typedef TaskState_t xTaskState

  Public API type alias for task states.
- typedef enum SchedulerState_e SchedulerState_t

  Enumerated type for scheduler state.
- typedef SchedulerState_t xSchedulerState

  Enumerated type for scheduler state.
- typedef enum Return_e Return_t

  Enumerated type for syscall return type.
- typedef Return_t xReturn

  Enumerated type for syscall return type.
- typedef void TaskParm_t

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

  Data type for the task paramater.
- typedef uint8_t Base_t

  Data type for the base type.
- typedef Base_t xBase

Data type for the base type.

- typedef uint8_t Byte_t

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

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

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

  Data type for a pointer to a memory address.
- typedef size_t 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_t HalfWord_t

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

  Data type for a 16-bit half word.
- typedef uint32_t Word_t

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

  Data type for a 32-bit word.
- typedef uint32_t Ticks_t

  Data type for system ticks.
- typedef Ticks_t xTicks

  Data type for system ticks.
- typedef void Task_t

  Data type for a task.
- typedef Task_t * xTask

  Data type for a task.
- typedef void Timer_t

  Data type for a timer.
- typedef Timer_t * xTimer

  Data type for a timer.
- typedef void Queue_t

  Data type for a queue.
- typedef Queue_t * xQueue

  Data type for a queue.
- typedef void StreamBuffer_t

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

  Data type for a stream buffer.
- typedef struct Volume_s Volume_t

  Data structure for FAT32 volume metadata.
- typedef Volume_t * xVolume

  Data type for a FAT32 volume.
- typedef struct File_s File_t

  Data structure for file handle.
- typedef File_t * xFile

  Data type for a file handle.
- typedef struct Dir_s Dir_t

  Data structure for directory handle.

- typedef Dir_t * xDir

  Data type for a directory handle.
- typedef struct DirEntry_s DirEntry_t

  Data structure for directory entry information.
- typedef DirEntry_t * xDirEntry

  Data type for a directory entry.
- typedef struct VolumeInfo_s VolumeInfo_t

  Data structure for volume information.
- typedef VolumeInfo_t * xVolumeInfo

  Data type for volume information.
- 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_e { TaskStateSuspended , TaskStateRunning , TaskStateWaiting }

  Enumerated type defining the possible states of a task.
- enum SchedulerState_e { SchedulerStateSuspended , SchedulerStateRunning }

  Enumerated type for scheduler state.
- enum Return_e { ReturnOK , ReturnError }

  Enumerated type for syscall return type.

Functions

- xReturn xDeviceRegisterDevice (xReturn(∗device_self_register_)())

  Register a device driver with HeliOS.
- xReturn xDeviceIsAvailable (const xHalfWord uid_, xBase ∗res_)

  Syscall to query the device driver about the availability of a device.
- xReturn xDeviceSimpleWrite (const xHalfWord uid_, xByte data_)

  Syscall to write a byte of data to the device.
- xReturn xDeviceWrite (const xHalfWord uid_, xSize ∗size_, xAddr data_)

  Write data to a device.
- xReturn xDeviceSimpleRead (const xHalfWord uid_, xByte ∗data_)

  Syscall to read a byte of data from the device.
- xReturn xDeviceRead (const xHalfWord uid_, xSize ∗size_, xAddr ∗data_)

  Read data 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_)

  Allocate memory from the user heap.
- xReturn xMemFree (const volatile xAddr addr_)

  Free memory previously allocated from the user heap.
- xReturn xMemFreeAll (void)

  Syscall to free all 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_)

  Create a message queue for inter-task communication.
- xReturn xQueueDelete (xQueue queue_)

  Delete a message queue and free its resources.
- xReturn xQueueGetLength (const xQueue queue_, xBase ∗res_)

  Query the maximum message capacity of a queue.
- xReturn xQueueIsQueueEmpty (const xQueue queue_, xBase ∗res_)

  Check if a message queue has no messages waiting.
- xReturn xQueueIsQueueFull (const xQueue queue_, xBase ∗res_)

  Check if a message queue is at maximum capacity.
- xReturn xQueueMessagesWaiting (const xQueue queue_, xBase ∗res_)

  Query the number of messages currently in a queue.
- xReturn xQueueSend (xQueue queue_, const xBase bytes_, const xByte ∗value_)

  Send a message to a queue (producer operation)
- xReturn xQueuePeek (const xQueue queue_, xQueueMessage ∗message_)

  Examine the next queue message without removing it.
- xReturn xQueueDropMessage (xQueue queue_)

  Remove the next message from queue without retrieving it.
- xReturn xQueueReceive (xQueue queue_, xQueueMessage ∗message_)

Receive and remove a message from a queue (consumer operation)

- xReturn xQueueLockQueue (xQueue queue_)

  Lock a queue to prevent new messages from being sent.
- xReturn xQueueUnLockQueue (xQueue queue_)

  Unlock a queue to resume accepting new messages.
- xReturn xStreamCreate (xStreamBuffer *stream_)

  Create a byte-oriented stream buffer for inter-task communication.
- xReturn xStreamDelete (const xStreamBuffer stream_)

  Delete a stream buffer and free its resources.
- xReturn xStreamSend (xStreamBuffer stream_, const xByte byte_)

  Send a single byte to a stream buffer (producer operation)
- xReturn xStreamReceive (const xStreamBuffer stream_, xHalfWord *bytes_, xByte **data_)

  Receive all waiting bytes from a stream buffer (consumer operation)
- xReturn xStreamBytesAvailable (const xStreamBuffer stream_, xHalfWord *bytes_)

  Query the number of bytes waiting in a stream buffer.
- xReturn xStreamReset (const xStreamBuffer stream_)

  Clear all bytes from a stream buffer.
- xReturn xStreamIsEmpty (const xStreamBuffer stream_, xBase *res_)

  Check if a stream buffer contains no waiting bytes.
- xReturn xStreamIsFull (const xStreamBuffer stream_, xBase *res_)

  Check if a stream buffer is at full capacity.
- xReturn xSystemAssert (const char *file_, const int line_)

  Syscall to to raise a system assert.
- xReturn xSystemInit (void)

  Syscall to bootstrap HeliOS.
- xReturn xSystemHalt (void)

  Syscall to halt HeliOS.
- xReturn xSystemGetSystemInfo (xSystemInfo *info_)

  Syscall to inquire about the system.
- xReturn xTaskCreate (xTask *task_, const xByte *name_, void(*callback_)(xTask task_, xTaskParm parm_), xTaskParm taskParameter_)

  Create a new task for cooperative multitasking.
- xReturn xTaskDelete (const xTask task_)

  Delete a task and free its resources.
- xReturn xTaskGetHandleByName (xTask *task_, const xByte *name_)

  Syscall to get the task handle by name.
- xReturn xTaskGetHandleById (xTask *task_, const xBase id_)

  Syscall to get the task handle by task id.
- xReturn xTaskGetAllRunTimeStats (xTaskRunTimeStats *stats_, xBase *tasks_)

  Syscall to get obtain the runtime statistics of all tasks.
- xReturn xTaskGetTaskRunTimeStats (const xTask task_, xTaskRunTimeStats *stats_)

  Syscall to get the runtime statistics for a single task.
- xReturn xTaskGetNumberOfTasks (xBase *tasks_)

  Syscall to get the number of tasks.
- xReturn xTaskGetTaskInfo (const xTask task_, xTaskInfo *info_)

  Syscall to get info about a task.
- xReturn xTaskGetAllTaskInfo (xTaskInfo *info_, xBase *tasks_)

  Syscall to get info about all tasks.
- xReturn xTaskGetTaskState (const xTask task_, xTaskState *state_)

  Syscall to get the state of a task.
- xReturn xTaskGetName (const xTask task_, xByte **name_)

Syscall to get the name of a task.

- xReturn xTaskGetId (const xTask task_, xBase *id_)

  Syscall to get the task id of a task.
- xReturn xTaskNotifyStateClear (xTask task_)

  Syscall to clear a waiting direct-to-task notification.
- xReturn xTaskNotificationIsWaiting (const xTask task_, xBase *res_)

  Syscall to inquire as to whether a direct-to-task notification is waiting.
- xReturn xTaskNotifyGive (xTask task_, const xBase bytes_, const xByte *value_)

  Syscall to give (i.e., send) a task a direct-to-task notification.
- xReturn xTaskNotifyTake (xTask task_, xTaskNotification *notification_)

  Syscall to take (i.e. receive) a waiting direct-to-task notification.
- xReturn xTaskResume (xTask task_)

  Activate a task for scheduler execution.
- xReturn xTaskSuspend (xTask task_)

  Deactivate a task to prevent scheduler execution.
- xReturn xTaskWait (xTask task_)

  Place a task in event-driven waiting state.
- xReturn xTaskChangePeriod (xTask task_, const xTicks period_)

  Syscall to change the interval period of a task timer.
- xReturn xTaskChangeWDPeriod (xTask task_, const xTicks period_)

  Syscall to change the task watchdog timer period.
- xReturn xTaskGetPeriod (const xTask task_, xTicks *period_)

  Syscall to obtain the task timer period.
- xReturn xTaskResetTimer (xTask task_)

  Syscall to set the task timer elapsed time to nil.
- xReturn xTaskStartScheduler (void)

  Start the HeliOS cooperative task scheduler.
- xReturn xTaskResumeAll (void)

  Resume the scheduler to enable task execution.
- xReturn xTaskSuspendAll (void)

  Suspend the scheduler and return control to caller.
- xReturn xTaskGetSchedulerState (xSchedulerState *state_)

  Syscall to get the scheduler state.
- xReturn xTaskGetWDPeriod (const xTask task_, xTicks *period_)

  Syscall to get the task watchdog timer period.
- xReturn xTimerCreate (xTimer *timer_, const xTicks period_)

  Create a software timer for general-purpose timekeeping.
- xReturn xTimerDelete (const xTimer timer_)

  Delete an application timer and free its resources.
- xReturn xTimerChangePeriod (xTimer timer_, const xTicks period_)

  Change the period of an existing application timer.
- xReturn xTimerGetPeriod (const xTimer timer_, xTicks *period_)

  Query the current period of an application timer.
- xReturn xTimerIsTimerActive (const xTimer timer_, xBase *res_)

  Check if an application timer is currently running.
- xReturn xTimerHasTimerExpired (const xTimer timer_, xBase *res_)

  Check if an application timer has expired.
- xReturn xTimerReset (xTimer timer_)

  Reset an application timer's elapsed time to zero.
- xReturn xTimerStart (xTimer timer_)

  Start an application timer to begin timing.

- xReturn xTimerStop (xTimer timer_)

  Stop an application timer and halt timing.
- xReturn xFSFormat (const xHalfWord blockDeviceUID_, const xByte *volumeLabel_)

  Format a block device with a FAT32 filesystem.
- xReturn xFSMount (xVolume *volume_, const xHalfWord blockDeviceUID_)

  Mount a FAT32 filesystem from a block device.
- xReturn xFSUnmount (xVolume volume_)

  Unmount a FAT32 filesystem volume.
- xReturn xFSGetVolumeInfo (const xVolume volume_, xVolumeInfo *info_)

  Query information about a mounted FAT32 volume.
- xReturn xFileOpen (xFile *file_, xVolume volume_, const xByte *path_, const xByte mode_)

  Open a file on a mounted FAT32 volume.
- xReturn xFileClose (xFile file_)

  Close an open file.
- xReturn xFileRead (xFile file_, const xSize size_, xByte **data_)

  Read data from a file.
- xReturn xFileWrite (xFile file_, const xSize size_, const xByte *data_)

  Write data to a file.
- xReturn xFileSeek (xFile file_, const xWord offset_, const xByte origin_)

  Change the file position for reading or writing.
- xReturn xFileTell (const xFile file_, xWord *position_)

  Get the current file position.
- xReturn xFileGetSize (const xFile file_, xWord *size_)

  Get the size of a file in bytes.
- xReturn xFileSync (xFile file_)

  Flush file writes to storage.
- xReturn xFileTruncate (xFile file_, const xWord size_)

  Resize a file to a specified size.
- xReturn xFileEOF (const xFile file_, xBase *eof_)

  Check if file position is at end-of-file.
- xReturn xFileExists (xVolume volume_, const xByte *path_, xBase *exists_)

  Check if a file exists on the volume.
- xReturn xFileUnlink (xVolume volume_, const xByte *path_)

  Delete a file from the filesystem.
- xReturn xFileRename (xVolume volume_, const xByte *oldPath_, const xByte *newPath_)

  Rename or move a file.
- xReturn xFileGetInfo (xVolume volume_, const xByte *path_, xDirEntry *entry_)

  Get detailed information about a file.
- xReturn xDirOpen (xDir *dir_, xVolume volume_, const xByte *path_)

  Open a directory for reading.
- xReturn xDirClose (xDir dir_)

  Close an open directory.
- xReturn xDirRead (xDir dir_, xDirEntry *entry_)

  Read the next entry from a directory.
- xReturn xDirRewind (xDir dir_)

  Reset directory read position to beginning.
- xReturn xDirMake (xVolume volume_, const xByte *path_)

  Create a new directory.
- xReturn xDirRemove (xVolume volume_, const xByte *path_)

  Delete an empty directory.

4.2.1 Detailed Description

Author

Manny Peterson manny@heliosproj.org

Version

0.5.0

Date

2023-03-19

This header file provides the complete public API for HeliOS, a lightweight embedded operating system designed for resource-constrained microcontrollers. It includes all type definitions, enumerations, and system call (syscall) declarations needed to develop applications on HeliOS.

HeliOS provides the following subsystems:

- Task Management: Create and manage cooperative multitasking
- Memory Management: Dynamic heap allocation with safety checks
- Device I/O: Abstract device driver interface
- Timers: Software timers for periodic and one-shot events
- Queues: Inter-task message passing with FIFO semantics
- Streams: Byte-oriented data buffers for streaming I/O
- Filesystem: FAT32 filesystem support with block device abstraction

All HeliOS system calls follow the naming convention xSubsystem∗() where x is the prefix for public APIs and Subsystem identifies the functional area (e.g., xTask, xMem, xQueue).

Copyright

4.2.2 Typedef Documentation

4.2.2.1 Addr_t `typedef void Addr_t`

The Addr_t type is a pointer of type void and is used to pass addresses between the end-user application and syscalls. It is not necessary to use the Addr_t type within the end-user application as long as the type is not used to interact with the kernel through syscalls

See also

xAddr

4.2.2.2 Base_t `typedef uint8_t Base_t`

The Base_t type is a simple data type often used as an argument or result type for syscalls 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.

See also

xBase

Byte_t

4.2.2.3 Byte_t `typedef uint8_t Byte_t`

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

See also

xByte

4.2.2.4 Dir_t `typedef struct Dir_s Dir_t`

See also

xDir

xDirOpen()

xDirClose()

4.2.2.5 DirEntry_t `typedef struct DirEntry_s DirEntry_t`

See also

xDirEntry

xDirRead()

xFileGetInfo()

4.2.2.6 File_t `typedef struct File_s File_t`

See also

xFile

xFileOpen()

xFileClose()

4.2.2.7 HalfWord_t `typedef uint16_t HalfWord_t`

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

See also

xHalfWord

4.2.2.8 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.

See also

xMemoryRegionStats

xMemGetHeapStats()

xMemGetKernelStats()

xMemFree()

4.2.2.9 Queue_t `typedef void Queue_t`

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

See also

xQueue
xQueueCreate()
xQueueDelete()

4.2.2.10 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().

See also

xQueueMessage
xQueueReceive()
xQueuePeek()
CONFIG_MESSAGE_VALUE_BYTES
xMemFree()

4.2.2.11 Return_t `typedef enum Return_e Return_t`

All HeliOS syscalls return the 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(xMemGet↩ Used(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

See also

OK()
ERROR()
xReturn

4.2.2.12 SchedulerState_t `typedef enum SchedulerState_e 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().

See also

xSchedulerState
xTaskSuspendAll()
xTaskResumeAll()
xTaskGetSchedulerState()
xTaskStartScheduler()

4.2.2.13 Size_t `typedef size_t Size_t`

The Size_t type is used for the storage requirements of an object in memory and is always represented in bytes.

See also

xSize

4.2.2.14 StreamBuffer_t `typedef void StreamBuffer_t`

The StreamBuffer_t data type is used as a stream buffer. The stream buffer 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()

xStreamDelete()

4.2.2.15 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().

See also

xSystemInfo

xSystemGetSystemInfo()

OS_PRODUCT_NAME_SIZE

xMemFree()

4.2.2.16 Task_t `typedef void Task_t`

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

See also

xTask

xTaskCreate()

xTaskDelete()

4.2.2.17 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 name and state. The TaskInfo_t structure is returned by xTaskGetTaskInfo() and xTaskGetAllTaskInfo(). If only runtime statistics are needed, then TaskRun↩ TimeStats_t should be used because of its smaller memory footprint.

See also

- xTaskInfo
- xTaskGetTaskInfo()
- xTaskGetAllTaskInfo()
- CONFIG_TASK_NAME_BYTES
- xMemFree()

4.2.2.18 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 notification 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().

See also

- xTaskNotification
- xMemFree()
- xTaskNotifyGive()
- xTaskNotifyTake()
- xTaskWait()

4.2.2.19 TaskParm_t `typedef void 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().

See also

- xTaskParm
- xTaskCreate()
- xMemAlloc()
- xMemFree()

4.2.2.20 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.

See also

xTaskRunTimeStats

xTaskGetTaskRunTimeStats()

xTaskGetAllRunTimeStats()

xMemFree()

4.2.2.21 TaskState_t `typedef enum TaskState_e TaskState_t`

Tasks in HeliOS transition between three distinct states that control their execution behavior by the cooperative scheduler. The task state determines whether the scheduler will invoke the task's function during scheduling cycles.

State Transitions:

- New tasks begin in TaskStateSuspended
- xTaskResume() transitions a task to TaskStateRunning
- xTaskSuspend() transitions a task to TaskStateSuspended
- xTaskWait() transitions a task to TaskStateWaiting

Scheduling Behavior:

- TaskStateSuspended: Task will NOT be scheduled for execution
- TaskStateRunning: Task will be scheduled normally based on its period
- TaskStateWaiting: Task will be scheduled only after a task event occurs (e.g., timer expiration, notification)

Note

HeliOS uses cooperative multitasking. Tasks must voluntarily yield control to allow other tasks to execute.

See also

xTaskState

xTaskResume()

xTaskSuspend()

xTaskWait()

xTaskGetTaskState()

4.2.2.22 Ticks_t `typedef uint32_t 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.

See also

xTicks

4.2.2.23 Timer_t `typedef void Timer_t`

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

See also

xTimer
xTimerCreate()
xTimerDelete()

4.2.2.24 Volume_t `typedef struct Volume_s Volume_t`

See also

xVolume
xFSMount()
xFSUnmount()

4.2.2.25 VolumeInfo_t `typedef struct VolumeInfo_s VolumeInfo_t`

See also

xVolumeInfo
xFSGetVolumeInfo()

4.2.2.26 Word_t `typedef uint32_t Word_t`

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

See also

xWord

4.2.2.27 xAddr typedef Addr_t* xAddr

See also

Addr_t

4.2.2.28 xBase typedef Base_t xBase

See also

Base_t

4.2.2.29 xByte typedef Byte_t xByte

See also

Byte_t

4.2.2.30 xDir typedef Dir_t* xDir

See also

Dir_t

xDirOpen()

xDirClose()

4.2.2.31 xDirEntry typedef DirEntry_t* xDirEntry

See also

DirEntry_t

xDirRead()

xFileGetInfo()

4.2.2.32 xFile `typedef File_t* xFile`

See also

File_t
xFileOpen()
xFileClose()

4.2.2.33 xHalfWord `typedef HalfWord_t xHalfWord`

See also

HalfWord_t

4.2.2.34 xQueue `typedef Queue_t* xQueue`

See also

Queue_t

4.2.2.35 xReturn `typedef Return_t xReturn`

See also

Return_t

4.2.2.36 xSchedulerState `typedef SchedulerState_t xSchedulerState`

See also

SchedulerState_t

4.2.2.37 xSize `typedef Size_t xSize`

See also

Size_t

4.2.2.38 xStreamBuffer `typedef StreamBuffer_t* xStreamBuffer`

See also

StreamBuffer_t

4.2.2.39 xTask `typedef Task_t* xTask`

See also

Task_t

4.2.2.40 xTaskNotification `typedef TaskNotification_t* xTaskNotification`

See also

TaskNotification_t

4.2.2.41 xTaskParm `typedef TaskParm_t* xTaskParm`

See also

TaskParm_t

4.2.2.42 xTaskState `typedef TaskState_t xTaskState`

This is the user-facing type name for task states, providing a consistent naming convention across the HeliOS public API where all types are prefixed with 'x'.

See also

TaskState_t

4.2.2.43 xTicks `typedef Ticks_t xTicks`

See also

Ticks_t

4.2.2.44 xTimer `typedef Timer_t* xTimer`

See also

Timer_t

4.2.2.45 xVolume `typedef Volume_t* xVolume`

See also

Volume_t

xFSMount()

xFSUnmount()

4.2.2.46 xVolumeInfo `typedef VolumeInfo_t* xVolumeInfo`

See also

VolumeInfo_t

xFSGetVolumeInfo()

4.2.2.47 xWord `typedef Word_t xWord`

See also

Word_t

4.2.3 Enumeration Type Documentation

4.2.3.1 Return_e `enum Return_e`

All HeliOS syscalls return the 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(xMemGet↩ Used(&size))) {} or if(ERROR(xMemGetUsed(&size))) {}).

See also

OK()

ERROR()

xReturn

Enumerator

ReturnOK	Return value if the syscall was successful.
ReturnError	Return value if the syscall failed.

4.2.3.2 SchedulerState_e `enum SchedulerState_e`

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().

See also

xSchedulerState

xTaskSuspendAll()

xTaskResumeAll()

xTaskGetSchedulerState()

xTaskStartScheduler()

Enumerator

SchedulerStateSuspended	State the scheduler is in after calling xTaskSuspendAll(). TaskStartScheduler() 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.

4.2.3.3 TaskState_e `enum TaskState_e`

Tasks in HeliOS transition between three distinct states that control their execution behavior by the cooperative scheduler. The task state determines whether the scheduler will invoke the task's function during scheduling cycles.

State Transitions:

- New tasks begin in TaskStateSuspended
- xTaskResume() transitions a task to TaskStateRunning
- xTaskSuspend() transitions a task to TaskStateSuspended
- xTaskWait() transitions a task to TaskStateWaiting

Scheduling Behavior:

- TaskStateSuspended: Task will NOT be scheduled for execution
- TaskStateRunning: Task will be scheduled normally based on its period
- TaskStateWaiting: Task will be scheduled only after a task event occurs (e.g., timer expiration, notification)

Note

HeliOS uses cooperative multitasking. Tasks must voluntarily yield control to allow other tasks to execute.

See also

xTaskState

xTaskResume()

xTaskSuspend()

xTaskWait()

xTaskGetTaskState()

Enumerator

TaskStateSuspended	Task is inactive and will not be scheduled. This is the initial state after task creation and the state after calling xTaskSuspend().
TaskStateRunning	Task is active and will be scheduled for execution according to its period. Set by calling xTaskResume().
TaskStateWaiting	Task is waiting for an event and will only be scheduled when that event occurs (timer, notification, etc.). Set by calling xTaskWait().

4.2.4 Function Documentation

4.2.4.1 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.

See also

xReturn

xMemAlloc()

xMemFree()

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

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))) {}).

4.2.4.2 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.

See also

xReturn

Parameters

*uid_*	The unique identifier ("UID") of the device driver to be operated on.

Returns

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))) {}).

4.2.4.3 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.

See also

xReturn

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

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))) {}).

4.2.4.4 xDeviceRead()

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

Reads data from the specified device through its registered driver. The device driver allocates a buffer from the user heap, fills it with data from the device, and returns it to the caller. This provides an abstract interface for device input, allowing applications to read from any registered device using a consistent API.

Unlike xDeviceWrite() where the caller allocates memory, xDeviceRead() has the DEVICE DRIVER allocate the buffer. This is necessary because the driver determines how much data is available to read. The caller receives both the data buffer pointer and the size of data read.

The read operation behavior depends on the device driver implementation:

- Serial devices typically read bytes from a receive buffer or UART
- Storage devices read from specific sectors or blocks
- Network devices receive packets
- Sensor devices read measurement data
- Custom devices implement application-specific read semantics

Data Flow:

1. Application calls xDeviceRead() with device UID 2. xDeviceRead() validates device UID, state, and permissions 3. Driver's read callback is invoked 4. Driver allocates buffer from heap (via xMemAlloc internally) 5. Driver fills buffer with data from device 6. Driver returns buffer pointer and size to HeliOS 7. HeliOS returns buffer to application 8. Application processes data then MUST free buffer with xMemFree()

Warning

The caller is RESPONSIBLE FOR FREEING the returned buffer using xMemFree(). Failing to free the buffer will cause memory leaks. The driver allocates this memory specifically for this read operation.

Note

Device read operations are synchronous by default. The function does not return until the driver completes the read operation and allocates the buffer. Drivers may implement buffering or blocking behavior internally.

The device must be in a readable state (DeviceModeReadOnly or DeviceModeReadWrite) and running state (DeviceStateRunning) for the read to succeed. Check device state via xDeviceConfigDevice() if needed.

If no data is available, driver behavior varies. Some drivers may return ReturnError, others may return an empty buffer (size=0), and others may block waiting for data. Check specific driver documentation.

Example Usage:

```
#define UART0_UID 0x0100
// Read data from UART xByte *rxBuffer = NULL;
xSize rxSize = 0;
if (OK(xDeviceRead(UART0_UID, &rxSize, (xAddr *)&rxBuffer))) {
  // Data successfully read if (rxSize > 0) {
    // Process received data processUARTData(rxBuffer, rxSize);
  }
  // IMPORTANT: Free the buffer allocated by driver
xMemFree((xAddr)rxBuffer);
}
// Reading from a block device
#define BLOCKDEV_UID 0x1000 xByte *sectorData = NULL;
xSize sectorSize = 0;
if (OK(xDeviceRead(BLOCKDEV_UID, &sectorSize, (xAddr *)&sectorData))) {
  // Sector data read (typically 512 or 4096 bytes) if (sectorSize > 0) {
    analyzeSectorData(sectorData, sectorSize);
  }
  // Free the buffer xMemFree((xAddr)sectorData);
}
// Reading in a loop (e.g., serial communication) void
serialReaderTask(xTask task, xTaskParm parm) {
  xByte *data = NULL;
  xSize len = 0;
  if (OK(xDeviceRead(UART0_UID, &len, (xAddr *)&data))) {
    if (len > 0) {
      handleSerialData(data, len);
    }
    xMemFree((xAddr)data);
  }
}
```

Parameters

in	*uid*↩ _	Unique identifier of the target device. Must match a UID previously registered via xDeviceRegisterDevice().
out	*size*↩ _	Pointer to size variable that receives the number of bytes read from the device. Set to 0 if no data available.
out	*data*↩ _	Pointer to buffer pointer variable. Receives address of heap-allocated buffer containing the read data. Caller MUST free this buffer with xMemFree() after use.

Returns

ReturnOK if data was read successfully (size may be 0), ReturnError if the read failed (invalid UID, device not found, device in wrong state/mode, memory allocation failed, or driver read operation failed).

See also

xDeviceWrite() - Write data to a device

xDeviceSimpleRead() - Read a single byte from a device

xDeviceRegisterDevice() - Register a device driver

xDeviceInitDevice() - Initialize a device

xDeviceConfigDevice() - Configure device state or parameters

xMemFree() - Free the buffer returned by this function (REQUIRED!)

4.2.4.5 xDeviceRegisterDevice() `xReturn xDeviceRegisterDevice ( xReturn(*)() device_self_register_ )`

Registers a device driver with the HeliOS kernel, making it available for I/O operations through the device abstraction layer. Device registration must occur before any device I/O functions (xDeviceRead(), xDeviceWrite(), etc.) can be used with that device.

HeliOS uses a self-registration pattern where each device driver provides its own registration function. This function is passed to xDeviceRegisterDevice(), which calls it to obtain the driver's callback functions, configuration, and metadata.

Device Registration Process:

1. Device driver provides a DRIVERNAME_self_register() function 2. Application calls xDeviceRegisterDevice() with this function pointer 3. HeliOS calls the registration function to obtain driver information 4. Driver is added to the internal device list with its unique ID (UID) 5. Device becomes available for I/O operations

Device Driver Requirements:

- Each driver must have a globally unique identifier (UID)
- Driver must provide callback functions for init, config, read, write
- Driver must specify its name, state, and access mode
- UID must not conflict with any other registered device

Warning

Device UIDs must be unique across all drivers in the application. Registering multiple devices with the same UID will cause undefined behavior. Common practice is to use high byte for driver type and low byte for instance (e.g., 0x0100 for first UART, 0x0200 for first SPI).

Note

Once registered, a device cannot be unregistered. However, devices can be placed in suspended state via xDeviceConfigDevice() to disable them.

Device registration should occur during system initialization, before xTaskStartScheduler() is called, to ensure devices are available when tasks begin executing.

The device driver model is extensible. Driver authors define their own state machines, configuration structures, and operational modes within the framework provided by the callback functions.

Example Usage:

```
// In device driver file (e.g., uart_driver.c) xReturn
UART0_self_register(void) {
  // Register driver with HeliOS return __RegisterDevice__(
    0x0100,              // Unique ID for UART0
    "UART0   ",          // 8-byte name DeviceStateRunning,  // Initial
state DeviceModeReadWrite, // Access mode UART0_init,          // Init
callback UART0_config,        // Config callback UART0_read,          //
Read callback UART0_write,         // Write callback UART0_simple_read,
  // Simple read callback UART0_simple_write   // Simple write callback
  );
}
// In application code (main.c) int main(void) {
  // Register UART driver if
(OK(xDeviceRegisterDevice(UART0_self_register))) {
    // Initialize the device if (OK(xDeviceInitDevice(0x0100))) {
      // Device ready for I/O xDeviceWrite(0x0100, &size, data);
    }
  }
  xTaskStartScheduler();
}
```

Parameters

in	*device_self_↩ register_*	Pointer to the device driver's self-registration function. This function must return ReturnOK on successful registration. By convention, this function is named DRIVERNAME_self_register().

Returns

ReturnOK if the device was successfully registered, ReturnError if registration failed (duplicate UID, invalid driver structure, out of memory, or driver registration function returned error).

See also

xDeviceInitDevice() - Initialize a registered device

xDeviceConfigDevice() - Configure device parameters or state

xDeviceRead() - Read data from a device

xDeviceWrite() - Write data to a device

xDeviceIsAvailable() - Check if a device is registered

CONFIG_DEVICE_NAME_BYTES - Device name length configuration

4.2.4.6 xDeviceSimpleRead()

```
xReturn xDeviceSimpleRead (
        const xHalfWord uid_,
        xByte * data_ )
```

The xDeviceSimpleRead() syscall will read a byte of data from a device. 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.

See also

xReturn

Parameters

*uid_*	The unique identifier ("UID") of the device driver to be operated on.
*data_*	The byte of data read from the device.

Returns

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))) {}).

4.2.4.7 xDeviceSimpleWrite()

```
xReturn xDeviceSimpleWrite (
    const xHalfWord uid_,
    xByte data_ )
```

The xDeviceSimpleWrite() syscall will write a byte of data to a device. 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.

See also

xReturn

Parameters

*uid_*	The unique identifier ("UID") of the device driver to be operated on.
*data_*	A byte of data to be written to the device.

Returns

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))) {}).

4.2.4.8 xDeviceWrite()

```
xReturn xDeviceWrite (
    const xHalfWord uid_,
```

```
xSize * size_,
xAddr data_ )
```

Writes a buffer of data to the specified device through its registered driver. The data is transferred from user heap memory to the device via the driver's write callback function. This provides an abstract interface for device output, allowing applications to write to any registered device using a consistent API.

The write operation behavior depends on the device driver implementation:

- Serial devices typically write bytes to a transmit buffer or UART
- Storage devices write to specific sectors or blocks
- Network devices transmit packets
- Custom devices implement application-specific write semantics

Data Flow:

1. Application prepares data in heap-allocated buffer 2. xDeviceWrite() validates device UID, state, and permissions 3. Data is copied from user heap to kernel memory 4. Driver's write callback is invoked with kernel memory 5. Driver performs hardware-specific write operations 6. Driver returns number of bytes written via size_ parameter

Warning

The data buffer MUST be allocated from the user heap using xMemAlloc(). Stack-allocated buffers or static data will cause memory validation errors. HeliOS enforces this to maintain memory safety boundaries between user space and kernel space.

Note

Device write operations are synchronous by default. The function does not return until the driver completes the write operation. Drivers may implement buffering or asynchronous operations internally.

The size_ parameter is both input and output. On input, it specifies the number of bytes to write. On output (if driver supports it), it may reflect the actual number of bytes written, which can be less than requested for devices with limited buffers.

The device must be in a writable state (DeviceModeWriteOnly or DeviceModeReadWrite) and running state (DeviceStateRunning) for the write to succeed. Check device state via xDeviceConfigDevice() if needed.

Example Usage:

```
#define UART0_UID 0x0100
// Write string to UART const char *message = "Hello, World!\n";
xSize messageLen = strlen(message);
xByte *buffer = NULL;
// Allocate buffer from heap (required!) if (OK(xMemAlloc((volatile xAddr
*)&buffer, messageLen))) {
  // Copy data to heap buffer memcpy(buffer, message, messageLen);
  // Write to device if (OK(xDeviceWrite(UART0_UID, &messageLen,
(xAddr)buffer))) {
    // Data written successfully
  }
  // Free the buffer xMemFree((xAddr)buffer);
}
// Writing to a block device xByte *sectorData = NULL;
xSize sectorSize = 512;
if (OK(xMemAlloc((volatile xAddr *)&sectorData, sectorSize))) {
  // Fill sector data...
  prepareSectorData(sectorData, sectorSize);
  // Write to block device if (OK(xDeviceWrite(0x1000, &sectorSize,
(xAddr)sectorData))) {
    // Sector written
  }
  xMemFree((xAddr)sectorData);
}
```

Parameters

in	*uid_*	Unique identifier of the target device. Must match a UID previously registered via xDeviceRegisterDevice().
in,out	*size_*	Pointer to size variable. On input: number of bytes to write. On output: may be updated by driver to reflect actual bytes written (driver-dependent).
in	*data_*	Pointer to data buffer allocated via xMemAlloc(). Contains the data to write to the device.

Returns

ReturnOK if data was written successfully, ReturnError if the write failed (invalid UID, device not found, device in wrong state/mode, data not from heap, or driver write operation failed).

See also

xDeviceRead() - Read data from a device

xDeviceSimpleWrite() - Write a single byte to a device

xDeviceRegisterDevice() - Register a device driver

xDeviceInitDevice() - Initialize a device

xDeviceConfigDevice() - Configure device state or parameters

xMemAlloc() - Allocate heap memory for write buffer

xMemFree() - Free the write buffer after use

4.2.4.9 xDirClose() `xReturn xDirClose ( xDir dir_ )`

Closes a previously opened directory, freeing the directory handle and associated kernel resources. After closing, the directory handle becomes invalid and must not be used in any subsequent directory operations.

Always close directories when finished reading to free kernel resources and prevent resource leaks. Directory handles are limited, and failing to close them may prevent other directories from being opened.

Example 1: Basic directory listing with close

```
 xDir dir;
xDirEntry entry;
if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
  while (OK(xDirRead(dir, &entry))) {
    printf("%s\n", entry.name);
  }
  // Always close when done xDirClose(dir);
}
```

Example 2: Error handling with guaranteed close

```
 xReturn processDirectory(xVolume vol, const char *path) {
  xDir dir;
  xDirEntry entry;
  xReturn result = ReturnError;
  if (OK(xDirOpen(&dir, vol, (xByte*)path))) {
    while (OK(xDirRead(dir, &entry))) {
      if (processEntry(&entry)) {
        result = ReturnOK;
      }
    }
    // Close regardless of processing success/failure xDirClose(dir);
  }
  return result;
}
```

Parameters

in	*dir_*	Handle to the open directory to close. Must be a valid directory handle from xDirOpen(). After closing, becomes invalid.

Returns

ReturnOK if close succeeded, ReturnError if close failed due to invalid directory handle or directory not open.

Warning

After calling xDirClose(), the directory handle becomes invalid and must not be used in any directory operations.

Note

It is good practice to set the directory handle to null after closing to prevent accidental use of an invalid handle.

See also

xDirOpen() - Open directory for reading

xDirRead() - Read directory entries

4.2.4.10 xDirMake()

```
xReturn xDirMake (
        xVolume volume_,
        const xByte * path_ )
```

Creates a new directory at the specified path. The parent directory must already exist. Use this function to organize files into a directory structure.

Example: Create directory and add file

```
 if (OK(xDirMake(vol, (xByte*)"/logs"))) {
  xFile file;
  if (OK(xFileOpen(&file, vol, (xByte*)"/logs/system.log", FS_MODE_WRITE |
FS_MODE_CREATE))) {
    xFileWrite(file, 10, (xByte*)"Log start\n");
    xFileClose(file);
  }
}
```

Parameters

in	*volume_*	Handle to mounted volume.
in	*path_*	Path to new directory.

Returns

ReturnOK if directory created, ReturnError if failed (already exists, parent doesn't exist, or insufficient space).

Warning

Parent directory must exist. Create parent directories first if needed.

See also

xDirRemove() - Delete empty directory

xDirOpen() - Open directory

4.2.4.11 xDirOpen()

```
xReturn xDirOpen (
        xDir * dir_,
        xVolume volume_,
        const xByte * path_ )
```

Opens a directory for reading its contents, creating a directory handle used to iterate through directory entries with xDirRead(). This is the first step in browsing directory contents, listing files, or searching for specific entries.

Opening a directory allocates kernel resources and positions the read pointer at the first entry. Use xDirRead() to retrieve entries sequentially, xDirRewind() to return to the beginning, and xDirClose() to free resources when finished.

Directory reading workflow:

1. Open directory with xDirOpen() 2. Read entries with xDirRead() in a loop
2. Optionally rewind with xDirRewind() to re-read 4. Close with xDirClose() when finished

Common use cases:

- **File listing**: Display all files in a directory
- **File searching**: Find specific files by name or attributes
- **Directory scanning**: Process all files in directory
- **File counting**: Count files or calculate total size
- **Filtering**: Select files matching criteria

Example 1: List all files in directory

```
 xVolume vol;
xDir dir;
xDirEntry entry;
if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
  printf("Files in root directory:\n");
  while (OK(xDirRead(dir, &entry))) {
    if (entry.isDirectory) {
      printf("  [DIR]  %s\n", entry.name);
    } else {
      printf("  [FILE] %s (%lu bytes)\n", entry.name, entry.size);
    }
  }
  xDirClose(dir);
}
```

Example 2: Find specific file in directory

```
 xBase findFileInDirectory(xVolume vol, const char *dirPath, const
char *filename) {
  xDir dir;
  xDirEntry entry;
```

```
  xBase found = 0;
  if (OK(xDirOpen(&dir, vol, (xByte*)dirPath))) {
    while (OK(xDirRead(dir, &entry))) {
      if (strcmp((char*)entry.name, filename) == 0) {
        found = 1;
        break;
      }
    }
    xDirClose(dir);
  }
  return found;
}
```

Example 3: Calculate directory size

```
xWord calculateDirectorySize(xVolume vol, const char *path) {
  xDir dir;
  xDirEntry entry;
  xWord totalSize = 0;
  if (OK(xDirOpen(&dir, vol, (xByte*)path))) {
    while (OK(xDirRead(dir, &entry))) {
      if (!entry.isDirectory) {
        totalSize += entry.size;
      }
    }
    xDirClose(dir);
  }
  return totalSize;
}
```

Example 4: Count files by type

```
void countFileTypes(xVolume vol) {
  xDir dir;
  xDirEntry entry;
  xWord fileCount = 0, dirCount = 0;
  if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
    while (OK(xDirRead(dir, &entry))) {
      if (entry.isDirectory) {
        dirCount++;
      } else {
        fileCount++;
      }
    }
    xDirClose(dir);
    printf("Files: %lu, Directories: %lu\n", fileCount, dirCount);
  }
}
```

Parameters

out	*dir_*	Pointer to xDir handle to be initialized. On success, this handle is used for reading directory entries and must be closed with xDirClose().
in	*volume_*	Handle to mounted volume containing the directory. Must be a valid volume from xFSMount().
in	*path_*	Pointer to null-terminated string containing directory path. Use "/" for root directory, or "/dirname" for subdirectories.

Returns

ReturnOK if directory opened successfully, ReturnError if open failed due to invalid volume, directory not found, path refers to a file, or insufficient resources.

Warning

The directory handle must be closed with xDirClose() when finished. Failure to close directories causes resource leaks.

The path must refer to a directory, not a file. Opening a file path as a directory will fail.

Note

After opening, the read position is at the first entry. Use xDirRead() to retrieve entries sequentially.

Directory handles are allocated from kernel memory and are a limited resource. Always close directories promptly after use.

See also

xDirClose() - Close directory and free resources

xDirRead() - Read next directory entry

xDirRewind() - Reset to beginning of directory

xFileGetInfo() - Get info about specific path

4.2.4.12 xDirRead()

```
xReturn xDirRead (
        xDir dir_,
        xDirEntry * entry_ )
```

Retrieves the next entry from an open directory, advancing the read position. Entries are returned sequentially including both files and subdirectories. Use the isDirectory field in the returned entry to distinguish between them.

Reading continues until all entries have been retrieved. When no more entries are available, xDirRead() returns ReturnError, indicating the end of the directory. Use xDirRewind() to return to the beginning and re-read entries if needed.

Example 1: Process all files (skip directories)

```
 xDir dir;
xDirEntry entry;
if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
  while (OK(xDirRead(dir, &entry))) {
    // Process only files, not directories if (!entry.isDirectory) {
      processFile(vol, (char*)entry.name);
    }
  }
  xDirClose(dir);
}
```

Example 2: Filter files by extension

```
 xBase hasExtension(const char *filename, const char *ext) {
  const char *dot = strrchr(filename, '.');
  return (dot && strcmp(dot, ext) == 0);
}
void processLogFiles(xVolume vol) {
  xDir dir;
  xDirEntry entry;
  if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
    while (OK(xDirRead(dir, &entry))) {
      if (!entry.isDirectory && hasExtension((char*)entry.name, ".log")) {
        printf("Log file: %s (%lu bytes)\n", entry.name, entry.size);
      }
    }
    xDirClose(dir);
  }
}
```

Parameters

in	*dir_*	Handle to open directory from xDirOpen().
out	*entry_*	Pointer to xDirEntry structure to receive entry information.

Returns

ReturnOK if entry read successfully, ReturnError if no more entries (end of directory) or invalid directory handle.

Note

When xDirRead() returns ReturnError, it indicates the end of the directory, not necessarily a read error.

Use xDirRewind() to return to the beginning and re-read entries.

See also

xDirOpen() - Open directory for reading

xDirClose() - Close directory

xDirRewind() - Reset to beginning

4.2.4.13 xDirRemove()

```
xReturn xDirRemove (
        xVolume volume_,
        const xByte * path_ )
```

Removes an empty directory from the filesystem. The directory must be empty (contain no files or subdirectories) for the operation to succeed.

Example: Remove temporary directory

```
// Remove all files first xDir dir;
xDirEntry entry;
if (OK(xDirOpen(&dir, vol, (xByte*)"/temp"))) {
  while (OK(xDirRead(dir, &entry))) {
    if (!entry.isDirectory) {
      xByte path[256];
      snprintf((char*)path, sizeof(path), "/temp/%s", entry.name);
      xFileUnlink(vol, path);
    }
  }
  xDirClose(dir);
}
// Now remove empty directory xDirRemove(vol, (xByte*)"/temp");
```

Parameters

in	*volume_*	Handle to mounted volume.
in	*path_*	Path to directory to remove.

Returns

ReturnOK if directory removed, ReturnError if failed (not empty, doesn't exist, or is root directory).

Warning

Directory must be empty. Remove all files and subdirectories first.

Cannot remove the root directory.

See also

xDirMake() - Create directory

xFileUnlink() - Delete file

4.2.4.14 xDirRewind() `xReturn xDirRewind ( xDir dir_ )`

Resets the read position of an open directory back to the first entry, allowing the directory to be re-read from the start. This is useful when you need to make multiple passes through a directory without closing and reopening it.

Example: Count and then process files

```
 xDir dir;
xDirEntry entry;
xWord fileCount = 0;
if (OK(xDirOpen(&dir, vol, (xByte*)"/"))) {
  // First pass - count files while (OK(xDirRead(dir, &entry))) {
    if (!entry.isDirectory) {
      fileCount++;
    }
  }
  printf("Processing %lu files...\n", fileCount);
  // Rewind to beginning xDirRewind(dir);
  // Second pass - process files while (OK(xDirRead(dir, &entry))) {
    if (!entry.isDirectory) {
      processFile((char*)entry.name);
    }
  }
  xDirClose(dir);
}
```

Parameters

in	*dir_*	Handle to open directory from xDirOpen().

Returns

ReturnOK if rewind succeeded, ReturnError if invalid directory handle.

See also

xDirOpen() - Open directory

xDirRead() - Read entries

xDirClose() - Close directory

4.2.4.15 xFileClose() `xReturn xFileClose ( xFile file_ )`

Closes a previously opened file, flushing any pending writes to storage and freeing the file handle and associated kernel resources. After closing, the file handle becomes invalid and must not be used in any subsequent file operations.

Closing a file ensures data integrity by committing all buffered writes to the filesystem and updating file metadata (size, modification time, etc.). Always close files when finished to prevent data loss and resource leaks.

Close process:

1. Flushes any buffered write data to block device 2. Updates file directory entry (size, attributes) 3. Updates File Allocation Table if file size changed 4. Frees file handle from kernel memory 5. Invalidates file handle

Common scenarios:

- **Normal completion**: Close file after successful read/write operations
- **Error handling**: Close file even if operations failed
- **Resource management**: Close files promptly to free kernel resources
- **Shutdown**: Close all open files before unmounting volume

Example 1: Basic open/close pattern

```
 xFile file;
if (OK(xFileOpen(&file, vol, (xByte*)"/data.txt", FS_MODE_READ))) {
  // Perform file operations xByte *data;
  if (OK(xFileRead(file, 100, &data))) {
    processData(data, 100);
    xMemFree((xAddr)data);
  }
  // Always close when done xFileClose(file);
  // file is now invalid - do not use
}
```

Example 2: Error handling with guaranteed close

```
 xReturn writeData(xVolume vol, const char *path, xByte *data, xSize
size) {
  xFile file;
  xReturn result = ReturnError;
  if (OK(xFileOpen(&file, vol, (xByte*)path, FS_MODE_WRITE |
FS_MODE_CREATE))) {
    if (OK(xFileWrite(file, size, data))) {
      result = ReturnOK;
    }
    // Close file regardless of write success/failure xFileClose(file);
  }
  return result;
}
```

Example 3: Multiple file handling

```
 void processFiles(xVolume vol) {
  xFile input, output;
  // Open input file if (OK(xFileOpen(&input, vol, (xByte*)"/input.dat",
FS_MODE_READ))) {
    // Open output file if (OK(xFileOpen(&output, vol,
(xByte*)"/output.dat", FS_MODE_WRITE | FS_MODE_CREATE))) {
      // Process data from input to output xByte *data;
      xWord size;
      if (OK(xFileGetSize(input, &size)) && OK(xFileRead(input, size,
&data))) {
        xFileWrite(output, size, data);
        xMemFree((xAddr)data);
      }
      xFileClose(output);  // Close output first
    }
    xFileClose(input);  // Close input
  }
}
```

Example 4: Safe file handle cleanup

```
 xFile file = null;
void useFile(xVolume vol) {
  if (OK(xFileOpen(&file, vol, (xByte*)"/temp.dat", FS_MODE_WRITE))) {
    performOperations(file);
    // Close and nullify handle xFileClose(file);
    file = null;  // Prevent use-after-close
  }
}
```

Parameters

in	*file*↩ _	Handle to the open file to close. Must be a valid file handle obtained from a previous successful xFileOpen() call. After closing, this handle becomes invalid.

Returns

ReturnOK if close succeeded, ReturnError if close failed due to invalid file handle, file not open, or flush/write errors during close.

Warning

After calling xFileClose(), the file handle becomes invalid and must not be used in any file operations. Attempting to use a closed file will result in ReturnError.

If xFileClose() fails (returns ReturnError), data may not have been fully written to storage. Consider using xFileSync() before close for critical data to detect write failures earlier.

All files must be closed before unmounting the volume. Unmounting with open files may cause data loss or filesystem corruption.

Note

It is good practice to set the file handle to null after closing to prevent accidental use of an invalid handle (use-after-close bugs).

Closing a file that was opened for writing commits the final file size to the directory entry. File size is updated only on close, not during write.

File handles are a limited resource. Always close files promptly after use to make handles available for other operations.

See also

xFileOpen() - Open a file

xFileSync() - Flush writes before close

xFSUnmount() - Unmount volume (close all files first)

4.2.4.16 xFileEOF()

```
xReturn xFileEOF (
    const xFile file_,
    xBase * eof_ )
```

Determines whether the current file position is at or beyond the end of the file, indicating that all data has been read and no more data is available. This is essential for implementing read loops that process entire files without knowing the file size in advance.

End-of-file (EOF) occurs when the file position equals or exceeds the file size. After reading the last byte of a file, subsequent reads will not advance the position further, and xFileEOF() will return true. EOF is a normal condition, not an error.

EOF behavior:

- True when file position >= file size
- False when more data available to read
- Not an error condition - indicates normal end of data
- Can occur mid-read if reading past end
- Unaffected by write operations that extend file

Common use cases:

- **Read loops**: Continue reading until EOF reached
- **Data validation**: Verify all expected data was read
- **Sequential processing**: Process file completely
- **Stream detection**: Know when to request more data

Example 1: Read file in chunks until EOF

```
 xFile file;
#define CHUNK_SIZE 512
void processEntireFile(xFile file) {
  xBase eof = 0;
  xByte *chunk;
  xWord totalBytesRead = 0;
  // Read until EOF while (!eof) {
    if (OK(xFileRead(file, CHUNK_SIZE, &chunk))) {
      processChunk(chunk, CHUNK_SIZE);
      totalBytesRead += CHUNK_SIZE;
      xMemFree((xAddr)chunk);
      // Check for EOF xFileEOF(file, &eof);
    } else {
      break;  // Read error
    }
  }
  logInfo("Processed %lu bytes", totalBytesRead);
}
```

Example 2: Validate file completely read

```
 xReturn readAndValidate(xFile file, xWord expectedSize) {
  xByte *data;
  xBase eof;
  // Read expected amount if (OK(xFileRead(file, expectedSize, &data))) {
    processData(data, expectedSize);
    xMemFree((xAddr)data);
    // Verify we're at EOF (no extra data) if (OK(xFileEOF(file, &eof))) {
      if (eof) {
        return ReturnOK;  // File size matches expected
      } else {
        logWarning("File has unexpected extra data");
        return ReturnError;
      }
    }
  }
  return ReturnError;
}
```

Example 3: Copy file with EOF detection

```
 xReturn copyFile(xFile source, xFile dest) {
  xBase eof = 0;
  xByte *buffer;
  while (!eof) {
    // Read chunk if (OK(xFileRead(source, 1024, &buffer))) {
      // Write to destination if (ERROR(xFileWrite(dest, 1024, buffer))) {
        xMemFree((xAddr)buffer);
        return ReturnError;
      }
      xMemFree((xAddr)buffer);
      // Check if source exhausted xFileEOF(source, &eof);
    } else {
      return ReturnError;
    }
  }
  return ReturnOK;  // Complete file copied
}
```

Example 4: Skip to end check

```
 xBase atEndOfFile(xFile file) {
  xBase eof;
  if (OK(xFileEOF(file, &eof))) {
    return eof;
  }
  return 0;
}
// Usage in parsing void parseRecords(xFile file) {
  while (!atEndOfFile(file)) {
    Record_t record;
    if (readRecord(file, &record)) {
      processRecord(&record);
    }
  }
}
```

Parameters

in	*file_*	Handle to the open file. Must be a valid file from xFileOpen().
out	*eof_*	Pointer to variable receiving EOF status. Set to non-zero (true) if at end-of-file, zero (false) if more data available.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid file handle or file not open.

Note

EOF is determined by comparing file position to file size. Position equals size when all data has been read.

For files opened with FS_MODE_APPEND, EOF is initially true since position starts at end of file.

After reaching EOF, you can seek back to earlier positions with xFileSeek() to reread data, which will clear the EOF condition.

Write operations that extend the file beyond the current position will clear the EOF condition.

See also

xFileRead() - Read data (may reach EOF)

xFileGetSize() - Get file size to calculate EOF

xFileTell() - Get current position

xFileSeek() - Move position (may clear EOF)

4.2.4.17 xFileExists()

```
xReturn xFileExists (
    xVolume volume_,
    const xByte * path_,
    xBase * exists_ )
```

Determines whether a file exists at the specified path on the mounted volume. This is useful for conditional file operations, avoiding errors when opening files, or checking for the presence of configuration or data files before attempting to access them.

The function searches the filesystem directory structure for an entry matching the specified path. It returns true if a file (not a directory) exists at that path, or false if the path does not exist or refers to a directory rather than a file.

Common use cases:

- **Conditional file access**: Open file only if it exists
- **Configuration detection**: Check for config files before loading
- **Backup verification**: Verify backup files exist before restore
- **File creation logic**: Create new file only if it doesn't exist
- **Validation**: Ensure required files are present

Example 1: Open file only if it exists

```
 xVolume vol;
xBase exists;
if (OK(xFileExists(vol, (xByte*)"/config.txt", &exists)) && exists) {
  // File exists - safe to open xFile file;
  if (OK(xFileOpen(&file, vol, (xByte*)"/config.txt", FS_MODE_READ))) {
    // Process file xFileClose(file);
  }
} else {
  // File doesn't exist - create default createDefaultConfig(vol);
}
```

Example 2: Check multiple configuration sources

```
 xReturn loadConfig(xVolume vol, Config_t *config) {
  xBase exists;
  const char *configPaths[] = {
    "/config.user.txt",
    "/config.default.txt",
    "/config.txt"
  };
  // Try each config file in priority order for (int i = 0; i < 3; i++) {
    if (OK(xFileExists(vol, (xByte*)configPaths[i], &exists)) && exists) {
      return loadConfigFromFile(vol, configPaths[i], config);
    }
  }
  // No config file found - use defaults return loadDefaultConfig(config);
}
```

Example 3: Conditional file creation

```
 xReturn ensureLogFile(xVolume vol) {
  xBase exists;
  // Check if log file exists if (OK(xFileExists(vol,
(xByte*)"/system.log", &exists))) {
    if (!exists) {
      // Create new log file with header xFile logFile;
      if (OK(xFileOpen(&logFile, vol, (xByte*)"/system.log", FS_MODE_WRITE
| FS_MODE_CREATE))) {
        const char *header = "=== System Log ===\n";
        xFileWrite(logFile, strlen(header), (xByte*)header);
        xFileClose(logFile);
        return ReturnOK;
      }
    } else {
      // Log file already exists return ReturnOK;
    }
  }
  return ReturnError;
}
```

Example 4: Validate required files

```
 xReturn validateRequiredFiles(xVolume vol) {
  const char *requiredFiles[] = {
    "/firmware.bin",
    "/config.dat",
    "/calibration.dat"
  };
  xBase exists;
  for (int i = 0; i < 3; i++) {
    if (ERROR(xFileExists(vol, (xByte*)requiredFiles[i], &exists)) ||
!exists) {
      logError("Required file missing: %s", requiredFiles[i]);
      return ReturnError;
    }
  }
  // All required files present return ReturnOK;
}
```

Parameters

in	*volume_*	Handle to the mounted volume to search. Must be a valid volume from xFSMount().
in	*path_*	Pointer to null-terminated string containing file path. Use forward slashes (/) for directory separators.
out	*exists_*	Pointer to variable receiving existence status. Set to non-zero (true) if file exists, zero (false) if not found or path refers to a directory.

Returns

ReturnOK if query succeeded (regardless of whether file exists), ReturnError if query failed due to invalid volume handle or path.

Note

This function returns true only for files, not directories. Use directory operations or xFileGetInfo() to check for directory existence.

File paths are case-sensitive. "/File.txt" and "/file.txt" are different.

Checking for existence and then opening the file is not atomic. In multitasking systems, the file could be deleted between the check and open operations.

See also

xFileOpen() - Open file (fails if doesn't exist without CREATE flag)

xFileGetInfo() - Get detailed file information including type

xDirOpen() - Open directory

4.2.4.18 xFileGetInfo()

```
xReturn xFileGetInfo (
    xVolume volume_,
    const xByte * path_,
    xDirEntry * entry_ )
```

Retrieves comprehensive information about a file including its name, size, attributes (read-only, hidden, system), and whether it's a file or directory. This information is useful for file browsing, validation, or determining how to process a filesystem entry.

The returned DirEntry structure contains all metadata about the file as stored in the directory entry, without needing to open the file. This is more efficient than opening the file when you only need to inspect its properties.

Information provided in xDirEntry:

- **name**: File or directory name (up to 256 characters)
- **size**: File size in bytes (0 for directories)
- **firstCluster**: Starting cluster number on disk
- **isDirectory**: True if entry is a directory, false if file
- **isReadOnly**: True if file is marked read-only
- **isHidden**: True if file is marked hidden
- **isSystem**: True if file is marked as system file

Common use cases:

- **File browser**: Display file details in directory listings
- **Validation**: Check file attributes before processing
- **Filtering**: Select files based on attributes or size

- **Metadata inspection**: Examine file properties without opening

Example 1: Get file size without opening

```
 xVolume vol;
xDirEntry entry;
if (OK(xFileGetInfo(vol, (xByte*)"/data.bin", &entry))) {
  if (!entry.isDirectory) {
    logInfo("File size: %lu bytes", entry.size);
    // Check if file is too large for processing if (entry.size >
MAX_FILE_SIZE) {
      logError("File too large to process");
    }
  }
}
```

Example 2: Check file type before opening

```
 xReturn processPath(xVolume vol, const char *path) {
  xDirEntry entry;
  if (OK(xFileGetInfo(vol, (xByte*)path, &entry))) {
    if (entry.isDirectory) {
      // It's a directory - process directory return processDirectory(vol,
path);
    } else {
      // It's a file - process file return processFile(vol, path);
    }
  }
  return ReturnError;
}
```

Example 3: Display file attributes

```
 void displayFileInfo(xVolume vol, const char *path) {
  xDirEntry entry;
  if (OK(xFileGetInfo(vol, (xByte*)path, &entry))) {
    printf("Name: %s\n", entry.name);
    printf("Size: %lu bytes\n", entry.size);
    printf("Type: %s\n", entry.isDirectory ? "Directory" : "File");
    // Display attributes printf("Attributes:");
    if (entry.isReadOnly) printf(" [READ-ONLY]");
    if (entry.isHidden) printf(" [HIDDEN]");
    if (entry.isSystem) printf(" [SYSTEM]");
    printf("\n");
  }
}
```

Example 4: Filter files by size and attributes

```
 xReturn findLargeFiles(xVolume vol, const char *dirPath, xWord
minSize) {
  xDir dir;
  xDirEntry entry;
  if (OK(xDirOpen(&dir, vol, (xByte*)dirPath))) {
    while (OK(xDirRead(dir, &entry))) {
      // Skip directories and hidden files if (!entry.isDirectory &&
!entry.isHidden) {
        if (entry.size >= minSize) {
          printf("Large file: %s (%lu bytes)\n", entry.name, entry.size);
        }
      }
    }
    xDirClose(dir);
    return ReturnOK;
  }
  return ReturnError;
}
```

Parameters

in	*volume_*	Handle to the mounted volume containing the file. Must be a valid volume from xFSMount().
in	*path_*	Pointer to null-terminated string containing path to file or directory. Use forward slashes (/) for separators.
out	*entry_*	Pointer to xDirEntry structure to receive file information. Structure is populated with all available metadata.

Returns

ReturnOK if information retrieved successfully, ReturnError if operation failed due to invalid volume, file/directory not found, or path error.

Note

This function works for both files and directories. Check the isDirectory field in the returned entry to determine which type it is.

The entry_ structure is populated with a copy of the directory entry data. Changes to this structure do not affect the file on disk.

The size field is 0 for directories, as directories don't have data content in the same way files do.

See also

xFileExists() - Check if file exists (simpler boolean check)

xDirRead() - Read directory entries sequentially

xFileOpen() - Open file to access content

xFileGetSize() - Get size of open file

4.2.4.19 xFileGetSize()

```
xReturn xFileGetSize (
    const xFile file_,
    xWord * size_ )
```

Retrieves the total size of an open file in bytes. This is the number of bytes of actual data content in the file, not including any filesystem metadata or directory entries. File size is useful for validating file integrity, allocating read buffers, calculating progress, or checking available data.

The file size represents the current data content and may change as write operations extend the file. Size is updated dynamically during write operations but is only committed to the directory entry when the file is closed.

Size characteristics:

- Measured in bytes of actual file content
- Does not include filesystem metadata or overhead
- Updates as file is written and extended
- Committed to directory entry on xFileClose()
- Zero for newly created empty files

Common use cases:

- **Buffer allocation**: Allocate exactly enough memory for file contents
- **Validation**: Verify file has expected size
- **Progress tracking**: Calculate percentage read/written
- **Bounds checking**: Ensure reads don't exceed file bounds

- **Empty file detection**: Check if size is zero

Example 1: Read entire file into memory

```
 xFile file;
xReturn readEntireFile(xVolume vol, const char *path, xByte **fileData,
xWord *size) {
  xFile file;
  if (OK(xFileOpen(&file, vol, (xByte*)path, FS_MODE_READ))) {
    // Get file size if (OK(xFileGetSize(file, size))) {
      // Read entire file if (OK(xFileRead(file, *size, fileData))) {
        xFileClose(file);
        return ReturnOK;
      }
    }
    xFileClose(file);
  }
  return ReturnError;
}
// Usage xByte *data;
xWord dataSize;
if (OK(readEntireFile(vol, "/config.txt", &data, &dataSize))) {
  processData(data, dataSize);
  xMemFree((xAddr)data);
}
```

Example 2: Validate file size before processing

```
#define MIN_VALID_SIZE 100
#define MAX_VALID_SIZE 10000
xReturn processValidatedFile(xFile file) {
  xWord size;
  // Check file size is in valid range if (OK(xFileGetSize(file, &size))) {
    if (size < MIN_VALID_SIZE) {
      logError("File too small");
      return ReturnError;
    }
    if (size > MAX_VALID_SIZE) {
      logError("File too large");
      return ReturnError;
    }
    // Size valid - proceed with processing return processFile(file, size);
  }
  return ReturnError;
}
```

Example 3: Copy file with progress reporting

```
 void copyFileWithProgress(xFile src, xFile dst) {
  xWord totalSize;
  xWord bytesProcessed = 0;
  xByte *buffer;
  xFileGetSize(src, &totalSize);
  while (bytesProcessed < totalSize) {
    xSize chunkSize = (totalSize - bytesProcessed > 512) ? 512 : (totalSize
- bytesProcessed);
    if (OK(xFileRead(src, chunkSize, &buffer))) {
      xFileWrite(dst, chunkSize, buffer);
      xMemFree((xAddr)buffer);
      bytesProcessed += chunkSize;
      // Report progress xByte percent = (xByte)((bytesProcessed * 100) /
totalSize);
      updateProgress(percent);
    } else {
      break;
    }
  }
}
```

Example 4: Check for empty file

```
 xBase isFileEmpty(xFile file) {
  xWord size;
  if (OK(xFileGetSize(file, &size))) {
    return (size == 0) ? 1 : 0;
  }
  return 0;
}
// Usage if (isFileEmpty(logFile)) {
  // File is empty - write header xFileWrite(logFile, strlen(HEADER),
(xByte*)HEADER);
}
```

Parameters

in	*file_*	Handle to the open file. Must be a valid file from xFileOpen().
out	*size_*	Pointer to variable receiving the file size in bytes.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid file handle or file not open.

Note

The size returned is the current file content size. For files open for writing, the size may increase as write operations extend the file.

File size is independent of file position. You can get the size at any time without affecting the current read/write position.

The size reported is the logical file size (bytes of content), not the physical storage size (which may be larger due to cluster allocation).

See also

xFileSeek() - Position within file based on size

xFileTell() - Get current position (may equal size at EOF)

xFileEOF() - Check if at end of file

xFileRead() - Read file data

xFSGetVolumeInfo() - Get total volume space

4.2.4.20 xFileOpen()

```
xReturn xFileOpen (
    xFile * file_,
    xVolume volume_,
    const xByte * path_,
    const xByte mode_ )
```

Opens a file for reading, writing, or both, creating a file handle used for subsequent file operations. The file can be opened in various modes controlling read/write access, creation behavior, and positioning. File paths use forward slash notation (/path/to/file.txt) following standard POSIX conventions.

Opening a file allocates kernel resources and establishes the connection between the application and the filesystem. The returned file handle must be used in all subsequent operations on that file and must be closed with xFileClose() when finished to prevent resource leaks.

File open modes (can be combined with bitwise OR):

- **FS_MODE_READ (0x01)**: Open for reading. File must exist.
- **FS_MODE_WRITE (0x02)**: Open for writing. Truncates existing file to zero length.
- **FS_MODE_APPEND (0x04)**: Open for appending. Positions at end of file.

- **FS_MODE_CREATE (0x08)**: Create file if it doesn't exist. Combine with WRITE or APPEND.

Common mode combinations:

- Read-only: FS_MODE_READ
- Write (truncate): FS_MODE_WRITE
- Write (create if needed): FS_MODE_WRITE | FS_MODE_CREATE
- Append: FS_MODE_APPEND | FS_MODE_CREATE
- Read/Write: FS_MODE_READ | FS_MODE_WRITE

Path format:

- Use forward slashes: "/file.txt" or "/dir/subdir/file.dat"
- Root directory files: "/filename.ext"
- Subdirectory files: "/directory/filename.ext"
- Maximum path length: CONFIG_FS_MAX_PATH_LENGTH (default 256 characters)

Example 1: Read existing file

```
 xVolume vol;
xFile file;
if (OK(xFSMount(&vol, blockDeviceUID))) {
  // Open file for reading if (OK(xFileOpen(&file, vol,
(xByte*)"/config.txt", FS_MODE_READ))) {
    xByte *data;
    xWord fileSize;
    // Get file size and read all data if (OK(xFileGetSize(file,
&fileSize))) {
      if (OK(xFileRead(file, fileSize, &data))) {
        processData(data, fileSize);
        xMemFree((xAddr)data);
      }
    }
    xFileClose(file);
  }
  xFSUnmount(vol);
}
```

Example 2: Create and write new file

```
 xVolume vol;
xFile file;
if (OK(xFSMount(&vol, deviceUID))) {
  // Open file for writing, create if doesn't exist if (OK(xFileOpen(&file,
vol, (xByte*)"/data.bin", FS_MODE_WRITE | FS_MODE_CREATE))) {
    xByte buffer[128];
    generateData(buffer, sizeof(buffer));
    // Write data if (OK(xFileWrite(file, sizeof(buffer), buffer))) {
      // Data written successfully
    }
    xFileClose(file);
  }
  xFSUnmount(vol);
}
```

Example 3: Append to log file

```
 xVolume vol;
void logMessage(const char *message) {
  xFile logFile;
  // Open log file in append mode, create if doesn't exist if
(OK(xFileOpen(&logFile, vol, (xByte*)"/system.log", FS_MODE_APPEND |
FS_MODE_CREATE))) {
    xSize msgLen = strlen(message);
    xFileWrite(logFile, msgLen, (xByte*)message);
    xFileWrite(logFile, 1, (xByte*)"\n");  // Add newline
    xFileClose(logFile);
  }
}
```

Example 4: Update existing file (read/write)

```
 xFile file;
// Open for both reading and writing if (OK(xFileOpen(&file, vol,
(xByte*)"/counter.dat", FS_MODE_READ | FS_MODE_WRITE))) {
  xByte *data;
  xWord counter;
  // Read current value if (OK(xFileRead(file, sizeof(xWord), &data))) {
    counter = *(xWord*)data;
    xMemFree((xAddr)data);
    // Increment counter counter++;
    // Seek back to start and write new value xFileSeek(file, 0,
FS_SEEK_SET);
    xFileWrite(file, sizeof(xWord), (xByte*)&counter);
  }
  xFileClose(file);
}
```

Parameters

out	*file_*	Pointer to xFile handle to be initialized. On success, this handle is used for all subsequent operations on the opened file. The handle remains valid until xFileClose() is called.
in	*volume_*	Handle to mounted volume containing the file. Must be a valid volume from xFSMount().
in	*path_*	Pointer to null-terminated string containing file path. Use forward slashes (/) for directory separators. Path is relative to volume root.
in	*mode_*	File open mode flags. Combine flags with bitwise OR (|). See FS_MODE_* constants for available modes.

Returns

ReturnOK if file opened successfully, ReturnError if open failed due to invalid volume, file not found (read mode without CREATE), insufficient space (create mode), invalid path, or too many open files.

Warning

The file handle must be closed with xFileClose() when finished. Failure to close files causes resource leaks and may prevent other files from being opened.

Opening a file with FS_MODE_WRITE (without APPEND) truncates the file to zero length immediately, destroying existing content. Use FS_MODE_READ | FS_MODE_WRITE if you need to modify existing content without truncation.

File paths are case-sensitive. "/File.txt" and "/file.txt" are different files.

Do not attempt to open the same file multiple times simultaneously. This may cause data corruption or filesystem inconsistencies.

Note

File handles are allocated from kernel memory and limited by system resources. Always close files promptly after use.

When creating files with FS_MODE_CREATE, parent directories must already exist. Use xDirMake() to create directories before creating files within them.

File position starts at 0 for READ and WRITE modes, and at end-of-file for APPEND mode. Use xFileSeek() to change position after opening.

See also

xFileClose() - Close file and free resources

xFileRead() - Read data from file

xFileWrite() - Write data to file

xFileSeek() - Change file position

xFSMount() - Mount volume before opening files

4.2.4.21 xFileRead()

```
xReturn xFileRead (
        xFile file_,
        const xSize size_,
        xByte ** data_ )
```

Reads a specified number of bytes from the current file position, allocating memory to hold the retrieved data and advancing the file position. The caller is responsible for freeing the allocated memory with xMemFree() after processing the data.

Reading retrieves data from the file starting at the current file position (initially 0, or set by xFileSeek()). After a successful read, the file position advances by the number of bytes read, positioning for the next sequential read.

Memory allocation pattern:

- xFileRead() allocates memory from user heap for the read data
- Caller receives pointer to allocated buffer via data_ parameter
- Caller must call xMemFree() after processing to prevent memory leaks
- Memory size equals the number of bytes successfully read

Read behavior:

- Reads up to size_ bytes, may read fewer if EOF reached
- Returns actual bytes read (could be less than requested)
- File position advances by number of bytes read
- Reading past EOF returns 0 bytes
- Sequential reads retrieve consecutive file data

Common scenarios:

- **Full file read**: Get file size, read entire file in one call
- **Chunked reading**: Read file in smaller blocks to conserve memory
- **Partial reads**: Read specific portions using xFileSeek()
- **Sequential processing**: Read file sequentially in fixed-size chunks

Example 1: Read entire file

```
 xFile file;
if (OK(xFileOpen(&file, vol, (xByte*)"/config.txt", FS_MODE_READ))) {
  xWord fileSize;
  xByte *data;
  // Get file size if (OK(xFileGetSize(file, &fileSize))) {
    // Read all data if (OK(xFileRead(file, fileSize, &data))) {
      // Process data parseConfig(data, fileSize);
      // Always free allocated memory xMemFree((xAddr)data);
    }
  }
  xFileClose(file);
}
```

Example 2: Read file in chunks

```
 xFile file;
#define CHUNK_SIZE 512
void processLargeFile(xFile file) {
  xBase eof = 0;
  xByte *chunk;
  // Read until end of file while (!eof) {
```

```
    if (OK(xFileRead(file, CHUNK_SIZE, &chunk))) {
      // Process this chunk processChunk(chunk, CHUNK_SIZE);
      xMemFree((xAddr)chunk);
      // Check if reached EOF xFileEOF(file, &eof);
    } else {
      break;
    }
  }
}
```

Example 3: Read specific portion of file

```
 xFile file;
// Read 100 bytes starting at offset 500 xReturn readFileSection(xFile
file, xWord offset, xSize count, xByte **data) {
  // Seek to desired position if (ERROR(xFileSeek(file, offset,
FS_SEEK_SET))) {
    return ReturnError;
  }
  // Read data from that position if (ERROR(xFileRead(file, count, data)))
{
    return ReturnError;
  }
  return ReturnOK;
}
// Usage xByte *sectionData;
if (OK(readFileSection(file, 500, 100, &sectionData))) {
  processData(sectionData, 100);
  xMemFree((xAddr)sectionData);
}
```

Example 4: Read with EOF detection

```
 xFile file;
void copyFileData(xFile srcFile, xFile dstFile) {
  xByte *buffer;
  xBase eof;
  do {
    // Read chunk from source if (OK(xFileRead(srcFile, 1024, &buffer))) {
      // Write to destination xFileWrite(dstFile, 1024, buffer);
      xMemFree((xAddr)buffer);
    }
    // Check if source EOF reached xFileEOF(srcFile, &eof);
  } while (!eof);
}
```

Parameters

in	*file_*	Handle to the open file to read from. Must be a valid file opened with FS_MODE_READ permission.
in	*size_*	Number of bytes to read from file. Actual bytes read may be less if end-of-file is reached before size_ bytes are available.
out	*data_*	Pointer to variable receiving address of newly allocated buffer containing the read data. Memory is allocated from user heap and must be freed with xMemFree() after use.

Returns

ReturnOK if read succeeded (even if fewer than size_ bytes read), ReturnError if read failed due to invalid file handle, file not open for reading, memory allocation failure, or device read error.

Warning

The memory allocated for data_ MUST be freed by the caller using xMemFree() after processing. Failure to free this memory will cause a memory leak. Always pair xFileRead() with xMemFree() in your code.

The file must have been opened with FS_MODE_READ permission. Attempting to read from a write-only file will fail.

xFileRead() may return fewer bytes than requested if EOF is reached. Check actual data size or use xFileEOF() to detect end-of-file condition.

Note

After a successful read, the file position advances by the number of bytes read. Subsequent reads continue from the new position.

Reading 0 bytes or reading past EOF is not an error—it returns ReturnOK with zero bytes allocated. Always check the actual bytes read.

For large files, consider reading in chunks rather than loading the entire file into memory to conserve heap space.

See also

xFileWrite() - Write data to file

xFileSeek() - Change file position before reading

xFileGetSize() - Get total file size

xFileEOF() - Check if end-of-file reached

xMemFree() - Free memory allocated by xFileRead()

xFileOpen() - Open file with read permission

4.2.4.22 xFileRename()

```
xReturn xFileRename (
    xVolume volume_,
    const xByte * oldPath_,
    const xByte * newPath_ )
```

Changes the name and/or location of a file within the filesystem. This operation can rename a file in the same directory or move it to a different directory with an optional new name. The file content and attributes are preserved; only the directory entry is updated.

Rename is an atomic operation that updates the directory structure without copying file data. This makes it efficient even for large files. The file must not be open during the rename operation.

Rename capabilities:

- Rename file in same directory: "/file.txt" → "/newname.txt"
- Move file to different directory: "/file.txt" → "/backup/file.txt"
- Move and rename: "/old.txt" → "/archive/new.txt"
- Preserves file content and attributes
- Efficient (no data copying required)

Common use cases:

- **Versioning**: Rename old file before creating new version
- **Organization**: Move files to appropriate directories
- **Backup**: Rename file to indicate backup or archive status
- **Atomic updates**: Write new file, rename old, rename new to final name
- **Temporary files**: Rename temp file to final name after completion

Example 1: Simple rename in same directory

```
 xVolume vol;
xReturn renameLogFile(void) {
  xBase exists;
  // Check if current log exists if (OK(xFileExists(vol,
(xByte*)"/system.log", &exists)) && exists) {
    // Rename to backup if (OK(xFileRename(vol, (xByte*)"/system.log",
(xByte*)"/system.log.old"))) {
      logInfo("Log file renamed for archival");
      return ReturnOK;
    }
  }
  return ReturnError;
}
```

Example 2: Log rotation with multiple versions

```
 void rotateLogs(xVolume vol) {
  xBase exists;
  // Rotate log.2 → log.3 (delete log.3 if exists) if (OK(xFileExists(vol,
(xByte*)"/log.3", &exists)) && exists) {
    xFileUnlink(vol, (xByte*)"/log.3");
  }
  if (OK(xFileExists(vol, (xByte*)"/log.2", &exists)) && exists) {
    xFileRename(vol, (xByte*)"/log.2", (xByte*)"/log.3");
  }
  // Rotate log.1 → log.2 if (OK(xFileExists(vol, (xByte*)"/log.1",
&exists)) && exists) {
    xFileRename(vol, (xByte*)"/log.1", (xByte*)"/log.2");
  }
  // Rotate current → log.1 if (OK(xFileExists(vol, (xByte*)"/system.log",
&exists)) && exists) {
    xFileRename(vol, (xByte*)"/system.log", (xByte*)"/log.1");
  }
}
```

Example 3: Atomic file replacement

```
 xReturn atomicConfigUpdate(xVolume vol, xByte *newConfig, xSize size)
{
  xFile tmpFile;
  // Write to temporary file first if (OK(xFileOpen(&tmpFile, vol,
(xByte*)"/config.tmp", FS_MODE_WRITE | FS_MODE_CREATE))) {
    if (OK(xFileWrite(tmpFile, size, newConfig))) {
      xFileSync(tmpFile);  // Ensure written to disk xFileClose(tmpFile);
      // Rename old config to backup xBase exists;
      if (OK(xFileExists(vol, (xByte*)"/config.dat", &exists)) && exists) {
        xFileRename(vol, (xByte*)"/config.dat", (xByte*)"/config.bak");
      }
      // Rename temp to final name if (OK(xFileRename(vol,
(xByte*)"/config.tmp", (xByte*)"/config.dat"))) {
        return ReturnOK;  // Atomic update successful
      }
    } else {
      xFileClose(tmpFile);
    }
  }
  return ReturnError;
}
```

Example 4: Move file to archive directory

```
 xReturn archiveOldData(xVolume vol, const char *filename) {
  xByte oldPath[256];
  xByte newPath[256];
  xBase exists;
  // Build paths snprintf((char*)oldPath, sizeof(oldPath), "/%s",
filename);
  snprintf((char*)newPath, sizeof(newPath), "/archive/%s", filename);
  // Check if file exists if (OK(xFileExists(vol, oldPath, &exists)) &&
exists) {
    // Move to archive directory if (OK(xFileRename(vol, oldPath,
newPath))) {
      logInfo("Archived: %s", filename);
      return ReturnOK;
    }
  }
  return ReturnError;
}
```

Parameters

in	*volume*↩ _	Handle to the mounted volume containing the file. Must be a valid volume from xFSMount().

Parameters

in	*oldPath_*	Pointer to null-terminated string containing current file path. The file must exist at this location.
in	*newPath_*	Pointer to null-terminated string containing new file path. If a file already exists at this path, the operation fails.

Returns

ReturnOK if file renamed successfully, ReturnError if rename failed due to invalid volume, source file not found, destination file already exists, file is open, or filesystem error.

Warning

The file must not be open during rename. Close all file handles before calling xFileRename(). Renaming an open file will fail.

If a file already exists at newPath_, the rename operation will fail. Delete the existing destination file first if you want to replace it.

Both paths must be on the same volume. You cannot use rename to move files between different volumes or storage devices.

The destination directory must exist. Create directories with xDirMake() before moving files into them.

Note

Rename is very efficient since it only updates directory entries without copying file data, regardless of file size.

File attributes and content are preserved during rename.

See also

xFileExists() - Check if source/destination exists

xFileUnlink() - Delete file instead of renaming

xFileClose() - Close file before renaming

xDirMake() - Create destination directory

4.2.4.23 xFileSeek()

```
xReturn xFileSeek (
    xFile file_,
    const xWord offset_,
    const xByte origin_ )
```

Moves the file position indicator to a new location within the file, controlling where subsequent read or write operations will occur. The file position can be set relative to the beginning of the file, the current position, or the end of the file using the origin parameter.

File positioning is essential for random access operations, allowing applications to read or write specific portions of a file without processing the entire file sequentially. The position is measured in bytes from the specified origin point.

Seek origins (defined in fs.h):

- **FS_SEEK_SET (0)**: Offset from beginning of file. offset_ is absolute position.
- **FS_SEEK_CUR (1)**: Offset from current position. offset_ is relative (can be negative).
- **FS_SEEK_END (2)**: Offset from end of file. offset_ is typically 0 or negative.

Common positioning operations:

- **Rewind to start**: xFileSeek(file, 0, FS_SEEK_SET)
- **Jump to end**: xFileSeek(file, 0, FS_SEEK_END)
- **Skip forward**: xFileSeek(file, 100, FS_SEEK_CUR)
- **Skip backward**: xFileSeek(file, -50, FS_SEEK_CUR)
- **Absolute position**: xFileSeek(file, 1000, FS_SEEK_SET)

Common scenarios:

- **Read file header then skip to data**: Seek past header to data section
- **Update specific record**: Seek to record position, write new data
- **Append detection**: Seek to end to get file size
- **Reread data**: Seek back to previous position

Example 1: Read file header and data separately

```
 xFile file;
typedef struct {
  xWord magic;
  xWord version;
  xWord dataOffset;
} FileHeader_t;
if (OK(xFileOpen(&file, vol, (xByte*)"/data.bin", FS_MODE_READ))) {
  xByte *headerData;
  FileHeader_t *header;
  // Read header at beginning if (OK(xFileRead(file, sizeof(FileHeader_t),
&headerData))) {
    header = (FileHeader_t*)headerData;
    // Validate and seek to data section if (header->magic == 0xDEADBEEF) {
      xFileSeek(file, header->dataOffset, FS_SEEK_SET);
      // Read data from new position xByte *data;
      xFileRead(file, 1024, &data);
      xMemFree((xAddr)data);
    }
    xMemFree((xAddr)headerData);
  }
  xFileClose(file);
}
```

Example 2: Update specific record in file

```
 typedef struct {
  xWord id;
  xByte status;
  xByte data[64];
} Record_t;
xReturn updateRecord(xFile file, xWord recordIndex, Record_t *newRecord) {
  xWord recordOffset = recordIndex * sizeof(Record_t);
  // Seek to record position if (ERROR(xFileSeek(file, recordOffset,
FS_SEEK_SET))) {
    return ReturnError;
  }
  // Write updated record return xFileWrite(file, sizeof(Record_t),
(xByte*)newRecord);
}
```

Example 3: Get file size using seek to end

```
 xWord getFileSize(xFile file) {
  xWord size;
  xWord currentPos;
```

```
  // Save current position xFileTell(file, &currentPos);
  // Seek to end and get position (which equals file size) xFileSeek(file,
0, FS_SEEK_END);
  xFileTell(file, &size);
  // Restore original position xFileSeek(file, currentPos, FS_SEEK_SET);
  return size;
}
```

Example 4: Skip chunks when processing

```
 xFile file;
#define CHUNK_SIZE 512
#define CHUNK_GAP 128
void processAlternateChunks(xFile file) {
  xByte *chunk;
  xBase eof = 0;
  while (!eof) {
    // Read chunk if (OK(xFileRead(file, CHUNK_SIZE, &chunk))) {
      processChunk(chunk, CHUNK_SIZE);
      xMemFree((xAddr)chunk);
      // Skip gap to next chunk xFileSeek(file, CHUNK_GAP, FS_SEEK_CUR);
      xFileEOF(file, &eof);
    } else {
      break;
    }
  }
}
```

Parameters

in	*file_*	Handle to the open file. Must be a valid file from xFileOpen().
in	*offset_*	Number of bytes to offset from origin. Can be positive or negative depending on origin. For FS_SEEK_SET, must be >= 0.
in	*origin_*	Reference point for offset. Use FS_SEEK_SET for absolute position, FS_SEEK_CUR for relative to current, FS_SEEK_END for relative to end of file.

Returns

ReturnOK if seek succeeded, ReturnError if seek failed due to invalid file handle, invalid origin value, or resulting position would be before start of file or beyond reasonable file bounds.

Warning

Seeking beyond the end of the file and then writing creates a sparse file with undefined data in the gap. The gap will contain whatever data was previously in those disk sectors.

Some seek operations may fail if they would position before the start of the file (negative absolute position).

Note

After a seek operation, the next read or write occurs at the new position. The position indicator is updated immediately.

Seeking does not change the file size. Only writing beyond the current end-of-file extends the file.

For FS_SEEK_CUR, offset_ can be negative to move backwards, or positive to move forwards from the current position.

See also

xFileTell() - Get current file position

xFileGetSize() - Get file size without seeking

xFileRead() - Read from current position

xFileWrite() - Write at current position

xFileEOF() - Check if at end of file

4.2.4.24 xFileSync() xReturn xFileSync (xFile *file_*)

Forces any buffered write data for the file to be written to the underlying storage device immediately, ensuring data persistence. This operation updates the file's directory entry and File Allocation Table (FAT) to reflect the current file size and cluster allocation.

Normally, write operations are buffered and only committed to storage when xFileClose() is called. For critical data that must survive system failures or power loss, call xFileSync() explicitly to guarantee the data is physically written to the storage device.

Sync ensures:

- All buffered write data is written to storage
- Directory entry is updated with current file size
- FAT is updated with current cluster allocation
- Data is persistent even if system crashes after sync
- File remains open for continued operations

When to use sync:

- **Critical data**: Financial transactions, configuration changes
- **Checkpointing**: Periodic saves during long operations
- **Error detection**: Verify writes succeeded before continuing
- **Power-loss protection**: Ensure data written before risky operations
- **Real-time logging**: Guarantee log entries are stored immediately

Example 1: Write critical configuration with sync

```
 xReturn saveConfig(xVolume vol, xByte *configData, xSize dataSize) {
  xFile file;
  if (OK(xFileOpen(&file, vol, (xByte*)"/config.dat", FS_MODE_WRITE |
FS_MODE_CREATE))) {
    // Write configuration data if (OK(xFileWrite(file, dataSize,
configData))) {
      // Force write to storage before continuing if (OK(xFileSync(file)))
{
        // Data guaranteed on disk xFileClose(file);
        return ReturnOK;
      } else {
        // Sync failed - data may be lost xFileClose(file);
        return ReturnError;
      }
    }
    xFileClose(file);
  }
  return ReturnError;
}
```

Example 2: Periodic checkpointing during long write

```
 xFile dataFile;
#define CHECKPOINT_INTERVAL 10
void processLargeDataset(xByte *dataset, xWord recordCount) {
  xWord i;
  for (i = 0; i < recordCount; i++) {
    // Write record xFileWrite(dataFile, RECORD_SIZE, &dataset[i *
RECORD_SIZE]);
    // Checkpoint every N records if ((i % CHECKPOINT_INTERVAL) == 0) {
      if (OK(xFileSync(dataFile))) {
        logInfo("Checkpoint at record %lu", i);
      } else {
        logError("Checkpoint failed - aborting");
        break;
```

```
      }
    }
  }
}
```

Example 3: Real-time event logging

```
 xFile eventLog;
void logCriticalEvent(const char *eventMsg) {
  xSize msgLen = strlen(eventMsg);
  // Write event xFileWrite(eventLog, msgLen, (xByte*)eventMsg);
  xFileWrite(eventLog, 1, (xByte*)"\n");
  // Force immediate write to storage if (ERROR(xFileSync(eventLog))) {
    // Could not guarantee persistence handleLogFailure();
  }
  // Event now safely on disk
}
```

Example 4: Transaction-style write with rollback

```
 xReturn atomicUpdate(xFile file, xByte *newData, xSize dataSize) {
  xWord originalPos;
  xWord originalSize;
  // Save state for rollback xFileTell(file, &originalPos);
  xFileGetSize(file, &originalSize);
  // Attempt write if (OK(xFileWrite(file, dataSize, newData))) {
    // Try to commit if (OK(xFileSync(file))) {
      return ReturnOK;  // Success - data committed
    } else {
      // Sync failed - truncate back to original size xFileTruncate(file,
originalSize);
      xFileSeek(file, originalPos, FS_SEEK_SET);
      return ReturnError;
    }
  }
  return ReturnError;
}
```

Parameters

in	*file*↩ _	Handle to the open file to sync. Must be a valid file opened with write permission (FS_MODE_WRITE or FS_MODE_APPEND).

Returns

ReturnOK if sync succeeded, ReturnError if sync failed due to invalid file handle, file not open for writing, or device write errors during flush operation.

Warning

Sync operations involve multiple disk writes (data, FAT, directory entry) and may take significant time, especially on slow storage devices. Use sync judiciously to balance data safety with performance.

If xFileSync() returns ReturnError, some data may have been written but the filesystem may be in an inconsistent state. Close and reopen the file or check filesystem integrity.

Note

xFileSync() does NOT close the file. After sync completes, the file remains open and can be used for additional read or write operations.

Calling xFileClose() automatically performs a sync before closing, so explicit xFileSync() is not required at the end of normal file operations.

For read-only files, xFileSync() has no effect and returns ReturnOK immediately.

See also

xFileWrite() - Write data that may be buffered until sync

xFileClose() - Close file (automatically syncs)

xFileTruncate() - Change file size

4.2.4.25 xFileTell()

```
xReturn xFileTell (
        const xFile file_,
        xWord * position_ )
```

Retrieves the current byte offset of the file position indicator, which indicates where the next read or write operation will occur. The position is measured in bytes from the beginning of the file, with 0 representing the first byte.

The file position changes automatically after read and write operations, and can be explicitly set with xFileSeek(). Knowing the current position is useful for saving and restoring read/write locations, calculating how much data has been processed, or implementing custom file navigation.

Position characteristics:

- Measured in bytes from beginning of file (0-based)
- Advances automatically after read/write operations
- Can be set explicitly with xFileSeek()
- Equals file size when at end-of-file
- Starts at 0 for READ/WRITE modes, at EOF for APPEND mode

Common use cases:

- **Save/restore position**: Remember position to return later
- **Progress tracking**: Monitor how much of file has been processed
- **Size calculation**: Position after seek-to-end equals file size
- **Offset calculation**: Compute relative offsets for records

Example 1: Save and restore file position

```
 xFile file;
xWord savedPosition;
// Save current position xFileTell(file, &savedPosition);
// Perform some operation that changes position xFileSeek(file, 0,
FS_SEEK_SET);
xByte *header;
xFileRead(file, 64, &header);
processHeader(header);
xMemFree((xAddr)header);
// Restore original position xFileSeek(file, savedPosition, FS_SEEK_SET);
// Continue from where we left off xByte *data;
xFileRead(file, 512, &data);
xMemFree((xAddr)data);
```

Example 2: Track read progress

```
 xFile file;
xWord fileSize;
xWord currentPos;
xFileGetSize(file, &fileSize);
while (1) {
  xByte *chunk;
  if (OK(xFileRead(file, 1024, &chunk))) {
    processChunk(chunk, 1024);
    xMemFree((xAddr)chunk);
    // Show progress xFileTell(file, &currentPos);
    xByte percent = (xByte)((currentPos * 100) / fileSize);
    updateProgressBar(percent);
    if (currentPos >= fileSize) break;
  } else {
    break;
  }
}
```

Example 3: Record file offset of data sections

```
 typedef struct {
```

```
  xWord sectionOffset;
  xWord sectionSize;
} SectionInfo_t;
SectionInfo_t sections[10];
xByte sectionCount = 0;
void indexFileSections(xFile file) {
  xByte *sectionHeader;
  while (sectionCount < 10) {
    // Record where this section starts xFileTell(file,
&sections[sectionCount].sectionOffset);
    // Read section header to get size if (OK(xFileRead(file, 4,
&sectionHeader))) {
      sections[sectionCount].sectionSize = *(xWord*)sectionHeader;
      xMemFree((xAddr)sectionHeader);
      // Skip to next section xFileSeek(file,
sections[sectionCount].sectionSize, FS_SEEK_CUR);
      sectionCount++;
    } else {
      break;
    }
  }
}
```

Parameters

in	*file_*	Handle to the open file. Must be a valid file from xFileOpen().
out	*position_*	Pointer to variable receiving the current file position in bytes. Position 0 is the first byte of the file.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid file handle or file not open.

Note

The position returned is always measured from the beginning of the file, regardless of how the file was opened or what seek operations were used.

When a file is first opened with FS_MODE_APPEND, xFileTell() will return the file size (position is at end of file).

The position advances after each read/write by the number of bytes transferred.

See also

xFileSeek() - Change file position

xFileGetSize() - Get total file size

xFileEOF() - Check if at end of file

xFileRead() - Read from current position

xFileWrite() - Write at current position

4.2.4.26 xFileTruncate()

```
xReturn xFileTruncate (
    xFile file_,
    const xWord size_ )
```

Changes the size of an open file to the specified number of bytes, either extending the file with undefined data or truncating it by discarding data beyond the new size. This operation allows precise control over file size independent of write operations.

Truncation behavior:

- If new size < current size: File is shortened, data beyond new size is lost
- If new size > current size: File is extended, gap filled with undefined data
- If new size == current size: No change, operation succeeds
- File position is preserved if still valid after truncation
- Freed clusters are returned to filesystem free space

Common use cases:

- **Preallocate space**: Extend file to reserve disk space before writing
- **Remove trailing data**: Truncate to exact data size
- **Rollback writes**: Shorten file to previous checkpoint
- **Clear file**: Truncate to zero to erase all content
- **Fixed-size files**: Ensure file is exact required size

Example 1: Truncate file to zero (clear contents)

```
 xFile file;
xReturn clearFile(xVolume vol, const char *path) {
  xFile file;
  // Open for writing if (OK(xFileOpen(&file, vol, (xByte*)path,
FS_MODE_WRITE))) {
    // Truncate to zero bytes if (OK(xFileTruncate(file, 0))) {
      xFileClose(file);
      return ReturnOK;
    }
    xFileClose(file);
  }
  return ReturnError;
}
```

Example 2: Preallocate file space

```
#define PREALLOCATE_SIZE (100 * 1024)  // 100 KB
xReturn createPreallocatedFile(xVolume vol, const char *path) {
  xFile file;
  // Create new file if (OK(xFileOpen(&file, vol, (xByte*)path,
FS_MODE_WRITE | FS_MODE_CREATE))) {
    // Preallocate space if (OK(xFileTruncate(file, PREALLOCATE_SIZE))) {
      // Space reserved - reset position to start xFileSeek(file, 0,
FS_SEEK_SET);
      xFileClose(file);
      return ReturnOK;
    }
    xFileClose(file);
  }
  return ReturnError;
}
```

Example 3: Trim file to actual data size

```
 xFile file;
void trimLogFile(xFile file, xWord actualDataSize) {
  xWord currentSize;
  // Get current file size if (OK(xFileGetSize(file, &currentSize))) {
    if (currentSize > actualDataSize) {
      // File is larger than needed - truncate excess xFileTruncate(file,
actualDataSize);
      logInfo("Trimmed %lu bytes from log file", currentSize -
actualDataSize);
    }
  }
}
```

Example 4: Rollback to checkpoint

```
 xFile dataFile;
xWord checkpointSize;
void saveCheckpoint(xFile file) {
  // Save current size as checkpoint xFileGetSize(file, &checkpointSize);
}
void rollbackToCheckpoint(xFile file) {
```

```
  xWord currentSize;
  xFileGetSize(file, &currentSize);
  if (currentSize > checkpointSize) {
    // Discard data written after checkpoint if (OK(xFileTruncate(file,
checkpointSize))) {
      xFileSeek(file, checkpointSize, FS_SEEK_SET);
      logInfo("Rolled back to checkpoint");
    }
  }
}
```

Parameters

in	*file*↩ _	Handle to the open file to resize. Must be a valid file opened with write permission (FS_MODE_WRITE or FS_MODE_APPEND).
in	*size*↩ _	New file size in bytes. Can be larger (extend) or smaller (truncate) than current size.

Returns

ReturnOK if truncate succeeded, ReturnError if operation failed due to invalid file handle, file not open for writing, insufficient disk space (for extension), or FAT update errors.

Warning

Truncating a file to a smaller size permanently deletes data beyond the new size. This data cannot be recovered. Ensure the new size is correct before calling this function.

If extending a file, the gap between the old size and new size contains undefined data (whatever was previously in those disk sectors). Do not rely on gap data being zero or any specific value.

The file position is not changed by truncation unless it exceeds the new file size, in which case it is clamped to the new size.

Note

Truncating to zero is an efficient way to clear a file while keeping it open.

Extending a file with xFileTruncate() allocates disk space but does not initialize the new space with any specific values.

The new file size is committed when xFileClose() or xFileSync() is called.

See also

xFileGetSize() - Query current file size

xFileWrite() - Write data (changes size automatically)

xFileSeek() - Change file position

xFileSync() - Commit size change to storage

xFileClose() - Close file and commit size

4.2.4.27 xFileUnlink()

```
xReturn xFileUnlink (
        xVolume volume_,
        const xByte * path_ )
```

Removes a file from the filesystem, freeing its disk space and removing its directory entry. This operation is permanent—deleted files cannot be recovered. The file must not be open when deleted; close all file handles before calling xFileUnlink().

Deletion process:

- Removes file's directory entry
- Marks file's clusters as free in FAT
- Returns disk space to available pool
- Operation is permanent and irreversible

Common scenarios:

- **Temporary file cleanup**: Remove temporary or cache files
- **Old log removal**: Delete old log files to free space
- **File replacement**: Delete old file before writing new version
- **Error recovery**: Remove corrupted or incomplete files
- **Space management**: Delete unnecessary files to reclaim storage

Example 1: Delete temporary file

```
 xVolume vol;
void cleanupTempFile(void) {
  xBase exists;
  // Check if temp file exists if (OK(xFileExists(vol, (xByte*)"/temp.dat",
&exists)) && exists) {
    // Delete temp file if (OK(xFileUnlink(vol, (xByte*)"/temp.dat"))) {
      logInfo("Temp file deleted");
    }
  }
}
```

Example 2: Replace file with new version

```
 xReturn updateConfigFile(xVolume vol, xByte *newConfig, xSize size) {
  xBase exists;
  // Check if old config exists if (OK(xFileExists(vol,
(xByte*)"/config.dat", &exists)) && exists) {
    // Delete old version if (ERROR(xFileUnlink(vol,
(xByte*)"/config.dat"))) {
      return ReturnError;
    }
  }
  // Write new config xFile file;
  if (OK(xFileOpen(&file, vol, (xByte*)"/config.dat", FS_MODE_WRITE |
FS_MODE_CREATE))) {
    xFileWrite(file, size, newConfig);
    xFileClose(file);
    return ReturnOK;
  }
  return ReturnError;
}
```

Example 3: Delete old log files to free space

```
 void cleanOldLogs(xVolume vol) {
  const char *oldLogs[] = {
    "/log.old.3",
    "/log.old.2",
    "/log.old.1"
  };
  xBase exists;
  xVolumeInfo volInfo;
```

```
  // Check if we need space if (OK(xFSGetVolumeInfo(vol, &volInfo))) {
    if (volInfo.freeBytes < (50 * 1024)) {  // Less than 50KB free
      // Delete old logs for (int i = 0; i < 3; i++) {
        if (OK(xFileExists(vol, (xByte*)oldLogs[i], &exists)) && exists) {
          xFileUnlink(vol, (xByte*)oldLogs[i]);
        }
      }
    }
  }
}
```

Example 4: Remove corrupted file

```
 xReturn validateAndClean(xVolume vol, const char *path) {
  xFile file;
  xBase isValid = 0;
  // Try to validate file if (OK(xFileOpen(&file, vol, (xByte*)path,
FS_MODE_READ))) {
    xByte *data;
    xWord size;
    if (OK(xFileGetSize(file, &size)) && OK(xFileRead(file, size, &data)))
{
      isValid = validateFileData(data, size);
      xMemFree((xAddr)data);
    }
    xFileClose(file);
  }
  // Delete if corrupted if (!isValid) {
    logWarning("Deleting corrupted file: %s", path);
    return xFileUnlink(vol, (xByte*)path);
  }
  return ReturnOK;
}
```

Parameters

in	*volume_*	Handle to the mounted volume containing the file. Must be a valid volume from xFSMount().
in	*path_*	Pointer to null-terminated string containing path to file to delete. Use forward slashes (/) for directory separators.

Returns

ReturnOK if file deleted successfully, ReturnError if deletion failed due to invalid volume, file not found, file is open, or filesystem error.

Warning

File deletion is permanent and irreversible. Deleted files cannot be recovered. Ensure you have the correct path before calling this function.

The file must not be open. Close all file handles to the file before attempting to delete it. Deleting an open file will fail.

Do not delete files while other tasks may be accessing them. This may cause those tasks' operations to fail unexpectedly.

Note

Deleting a file frees its disk space immediately, making it available for new files.

You cannot delete directories with xFileUnlink(). Use xDirRemove() to delete empty directories.

See also

xFileExists() - Check if file exists before deleting

xFileRename() - Rename file instead of deleting

xFileClose() - Close file before deleting

xDirRemove() - Delete empty directory

4.2.4.28 xFileWrite()

```
xReturn xFileWrite (
        xFile file_,
        const xSize size_,
        const xByte * data_ )
```

Writes a specified number of bytes to the file at the current file position, advancing the file position and potentially extending the file size. Data is written from the provided buffer to the filesystem, and changes are committed to storage when the file is closed or explicitly synced.

Writing appends or overwrites data at the current file position, which depends on the mode the file was opened with (WRITE starts at 0, APPEND starts at EOF). After a successful write, the file position advances by the number of bytes written, and the file size is updated if the write extended beyond the previous end-of-file.

Write behavior:

- Writes size_ bytes from data_ buffer to file
- File position advances by number of bytes written
- File size updated if write extends file
- Data buffered until xFileClose() or xFileSync()
- Allocates clusters from filesystem as needed

File modes and write behavior:

- **FS_MODE_WRITE**: Position starts at 0, overwrites from beginning
- **FS_MODE_APPEND**: Position starts at EOF, extends file
- **FS_MODE_READ** | **FS_MODE_WRITE**: Can write at any position via xFileSeek()

Common scenarios:

- **Create new file**: Open with WRITE | CREATE, write data
- **Append to log**: Open with APPEND | CREATE, write entries
- **Update existing**: Open with READ | WRITE, seek and write
- **Replace content**: Open with WRITE (truncates), write new data

Example 1: Write new file

```
 xFile file;
if (OK(xFileOpen(&file, vol, (xByte*)"/data.bin", FS_MODE_WRITE |
FS_MODE_CREATE))) {
  xByte buffer[256];
  generateData(buffer, sizeof(buffer));
  // Write data if (OK(xFileWrite(file, sizeof(buffer), buffer))) {
    // Data written successfully
  }
  xFileClose(file);  // Commits write to storage
}
```

Example 2: Append to log file

```
 void logEvent(xVolume vol, const char *event) {
  xFile logFile;
  // Open in append mode if (OK(xFileOpen(&logFile, vol,
(xByte*)"/events.log", FS_MODE_APPEND | FS_MODE_CREATE))) {
    xSize eventLen = strlen(event);
    // Write event message xFileWrite(logFile, eventLen, (xByte*)event);
    xFileWrite(logFile, 1, (xByte*)"\n");
    xFileClose(logFile);
  }
```

```
}
```

Example 3: Update specific file location

```
 xFile file;
// Update bytes at specific offset xReturn updateFileBytes(xFile file,
xWord offset, xByte *newData, xSize size) {
  // Seek to update position if (ERROR(xFileSeek(file, offset,
FS_SEEK_SET))) {
    return ReturnError;
  }
  // Write new data at that position return xFileWrite(file, size,
newData);
}
// Usage if (OK(xFileOpen(&file, vol, (xByte*)"/config.dat", FS_MODE_READ |
FS_MODE_WRITE))) {
  xByte newValue = 0x42;
  updateFileBytes(file, 100, &newValue, 1);  // Update byte at offset 100
xFileClose(file);
}
```

Example 4: Write with explicit sync

```
 xFile file;
if (OK(xFileOpen(&file, vol, (xByte*)"/critical.dat", FS_MODE_WRITE |
FS_MODE_CREATE))) {
  xByte importantData[512];
  prepareData(importantData, sizeof(importantData));
  // Write data if (OK(xFileWrite(file, sizeof(importantData),
importantData))) {
    // Force write to storage immediately if (OK(xFileSync(file))) {
      // Data guaranteed on disk
    } else {
      // Sync failed - write may not be persistent handleSyncError();
    }
  }
  xFileClose(file);
}
```

Parameters

in	*file_*	Handle to the open file to write to. Must be a valid file opened with FS_MODE_WRITE or FS_MODE_APPEND permission.
in	*size_*	Number of bytes to write to file. Data buffer must contain at least this many bytes.
in	*data_*	Pointer to buffer containing data to write. Buffer must be at least size_ bytes in length.

Returns

ReturnOK if write succeeded, ReturnError if write failed due to invalid file handle, file not open for writing, insufficient disk space, device write error, or FAT allocation failure.

Warning

The file must have been opened with FS_MODE_WRITE or FS_MODE_APPEND permission. Attempting to write to a read-only file will fail.

Writes are buffered and not guaranteed to be on storage until xFileClose() or xFileSync() is called. For critical data, call xFileSync() explicitly to ensure persistence before continuing.

If insufficient disk space is available, write will fail with ReturnError. Use xFSGetVolumeInfo() to check free space before writing large amounts of data.

Opening a file with FS_MODE_WRITE (without APPEND) truncates the file immediately. The first write starts at position 0, replacing all previous content.

Note

After a successful write, the file position advances by size_ bytes. Use xFileTell() to query current position or xFileSeek() to change it.

The data_ buffer is not modified by xFileWrite(). It can be a const buffer or reused after the call returns.

File size is updated as writes extend the file, but the final size is only committed to the directory entry when xFileClose() is called.

See also

xFileRead() - Read data from file

xFileOpen() - Open file with write permission

xFileClose() - Close file and commit writes

xFileSync() - Flush writes to storage

xFileSeek() - Change write position

xFSGetVolumeInfo() - Check available disk space

4.2.4.29 xFSFormat()

```
xReturn xFSFormat (
    const xHalfWord blockDeviceUID_,
    const xByte * volumeLabel_ )
```

Initializes a block device with a FAT32 filesystem structure, creating the boot sector, file allocation tables, and root directory. This operation prepares a storage device for use with HeliOS filesystem operations by writing the complete FAT32 metadata structures.

Formatting is destructive—all existing data on the block device is lost. This operation should only be performed on new devices or when explicitly reinitializing storage. After formatting, the device must be mounted with xFSMount() before files can be created or accessed.

The formatting process:

1. Validates block device is registered and operational 2. Calculates optimal FAT32 parameters based on device size 3. Writes boot sector with filesystem parameters 4. Initializes two file allocation tables (FAT1 and FAT2 for redundancy) 5. Creates empty root directory 6. Sets volume label if provided

Common scenarios:

- **Initial setup**: Format new RAM disk or flash storage
- **Factory reset**: Erase all data and reinitialize filesystem
- **Corruption recovery**: Reformat after filesystem corruption
- **Testing**: Create clean filesystem for unit tests

Example 1: Format RAM disk with volume label

```
#include "block_driver.h"
#include "ramdisk_driver.h"
// Setup and format RAM disk xHalfWord ramDiskUID = 0;
xHalfWord blockDevUID = 0;
// Initialize drivers if (OK(__RamDiskRegister__(&ramDiskUID))) {
  if (OK(__BlockDeviceRegister__(&blockDevUID))) {
    // Configure block device to use RAM disk BlockDeviceConfig_t config;
    config.protocol = BLOCK_PROTOCOL_RAW;
    config.backingDeviceUID = ramDiskUID;
    if (OK(__DeviceConfigDevice__(blockDevUID, sizeof(config),
(xByte*)&config))) {
      // Format with volume label if (OK(xFSFormat(blockDevUID,
(xByte*)"HELIOS_VOL"))) {
        // Filesystem ready to mount
      }
    }
  }
}
```

Example 2: Format and mount in one operation

```
 xReturn initializeFilesystem(xHalfWord blockDeviceUID, xVolume
volume) {
  // Format the device if (ERROR(xFSFormat(blockDeviceUID,
(xByte*)"STORAGE"))) {
    return ReturnError;
  }
  // Mount the freshly formatted filesystem if (ERROR(xFSMount(volume,
blockDeviceUID))) {
    return ReturnError;
  }
  // Filesystem formatted and ready for use return ReturnOK;
}
```

Example 3: Factory reset function

```
 xVolume systemVolume;
xHalfWord storageDeviceUID;
void factoryReset(void) {
  // Unmount if currently mounted xFSUnmount(systemVolume);
  // Reformat - destroys all data if (OK(xFSFormat(storageDeviceUID,
(xByte*)"FACTORY"))) {
    // Remount fresh filesystem if (OK(xFSMount(&systemVolume,
storageDeviceUID))) {
      // Create default configuration files
createDefaultConfig(systemVolume);
    }
  }
}
```

Parameters

in	*blockDevice↩ UID_*	UID of the block device to format. Must be a registered block device obtained from block device driver registration. The device must support read and write operations.
in	*volumeLabel_*	Pointer to null-terminated string containing volume label (max 11 characters). Can be null for no label. Label is stored in boot sector and visible in volume information queries.

Returns

ReturnOK if format succeeded, ReturnError if format failed due to invalid block device UID, device not available, insufficient device size, or write errors during format operation.

Warning

This operation is DESTRUCTIVE. All existing data on the block device will be permanently lost. Ensure the correct device UID is specified and any important data is backed up before formatting.

The block device must be large enough to hold FAT32 structures. Very small devices may fail to format. Minimum practical size depends on sector size and cluster size calculations.

Do not format a currently mounted volume. Always call xFSUnmount() before formatting to prevent filesystem corruption.

Note

After formatting, the volume must be mounted with xFSMount() before any file operations can be performed.

HeliOS uses FAT32 filesystem format for compatibility with standard FAT32 implementations and tools.

The volume label is optional but recommended for identifying storage volumes in systems with multiple devices.

See also

xFSMount() - Mount formatted volume for file operations

xFSUnmount() - Unmount volume before reformatting

xFSGetVolumeInfo() - Query volume information including label

BlockDeviceRegister() - Register block device before formatting

4.2.4.30 xFSGetVolumeInfo()

```
xReturn xFSGetVolumeInfo (
    const xVolume volume_,
    xVolumeInfo * info_ )
```

Retrieves comprehensive information about a mounted volume including capacity, free space, and filesystem parameters. This information is useful for monitoring storage utilization, checking available space before write operations, and reporting filesystem status.

The volume information includes both logical (cluster-based) and physical (byte-based) size metrics, allowing applications to make informed decisions about storage management and space allocation.

Information provided:

- Total and free space in clusters and bytes
- Bytes per sector (typically 512)
- Sectors per cluster
- Bytes per cluster (sector size × sectors per cluster)

Common use cases:

- **Space checking**: Verify sufficient space before write operations
- **Usage monitoring**: Track filesystem utilization over time
- **Status reporting**: Display storage capacity to user/diagnostic interface
- **Quota management**: Implement storage quotas based on available space

Example 1: Check space before writing file

```
 xVolume vol;
xVolumeInfo info;
xReturn writeDataFile(xVolume vol, xByte *data, xSize dataSize) {
  // Check available space if (OK(xFSGetVolumeInfo(vol, &info))) {
    if (info.freeBytes >= dataSize) {
      // Sufficient space - proceed with write xFile file;
      if (OK(xFileOpen(&file, vol, (xByte*)"/data.bin", FS_MODE_WRITE |
FS_MODE_CREATE))) {
        xFileWrite(file, dataSize, data);
        xFileClose(file);
        return ReturnOK;
      }
    } else {
      // Insufficient space return ReturnError;
    }
  }
  return ReturnError;
}
```

Example 2: Storage utilization monitoring

```
 xVolume vol;
void reportStorageStatus(void) {
  xVolumeInfo info;
  if (OK(xFSGetVolumeInfo(vol, &info))) {
    xWord usedBytes = info.totalBytes - info.freeBytes;
    xByte percentUsed = (xByte)((usedBytes * 100) / info.totalBytes);
    // Report to diagnostic interface printf("Storage: %lu / %lu bytes
(%u%% used)\n", usedBytes, info.totalBytes, percentUsed);

    // Warn if nearly full if (percentUsed > 90) {
      logWarning("Storage nearly full");
    }
  }
}
```

Example 3: Filesystem parameter reporting

```
 void displayVolumeDetails(xVolume vol) {
  xVolumeInfo info;
  if (OK(xFSGetVolumeInfo(vol, &info))) {
    printf("Volume Information:\n");
    printf("  Total: %lu bytes (%lu clusters)\n", info.totalBytes,
info.totalClusters);
    printf("  Free:  %lu bytes (%lu clusters)\n", info.freeBytes,
info.freeClusters);
    printf("  Cluster size: %lu bytes\n", info.bytesPerCluster);
    printf("  Sector size:  %u bytes\n", info.bytesPerSector);
    printf("  Sectors/cluster: %u\n", info.sectorsPerCluster);
  }
}
```

Parameters

in	*volume_*	Handle to the mounted volume to query. Must be a valid volume obtained from xFSMount().
out	*info_*	Pointer to xVolumeInfo structure to receive volume information. On success, this structure is populated with current volume statistics and parameters.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid volume handle or volume not mounted.

Note

Free space calculation requires scanning the File Allocation Table (FAT), which may take time on large volumes. Cache the result if you need to check free space frequently.

The free space reported is the total free space on the volume. Actual usable space may be less due to filesystem overhead and cluster alignment.

All byte values are in units of bytes. Cluster values represent the number of allocation units (clusters) used by the filesystem.

See also

xFSMount() - Mount volume before querying information

xFSFormat() - Format operation sets volume parameters

xFileGetSize() - Get size of individual files

4.2.4.31 xFSMount()

```
xReturn xFSMount (
    xVolume * volume_,
    const xHalfWord blockDeviceUID_ )
```

Mounts a FAT32 filesystem from a formatted block device, making it available for file and directory operations. Mounting reads and validates the filesystem metadata (boot sector, FAT tables) and creates a volume handle used in all subsequent filesystem operations.

A volume must be mounted before any file operations (open, read, write, etc.) can be performed. The volume handle returned by this function is required by all file and directory functions to identify which filesystem to operate on.

Mount process:

1. Validates block device UID and availability 2. Reads boot sector from block device 3. Validates FAT32 boot sector signature and parameters 4. Allocates volume structure in kernel memory 5. Stores filesystem parameters for subsequent operations 6. Returns volume handle for use in file/directory operations

Common usage patterns:

- **System initialization**: Mount filesystem during startup
- **Removable media**: Mount when device inserted, unmount when removed
- **Multi-volume systems**: Mount multiple block devices as separate volumes
- **Filesystem switching**: Unmount and remount different filesystems

Example 1: Basic mount operation

```
 xVolume myVolume;
xHalfWord blockDeviceUID = 1;  // From block device registration
// Mount the filesystem if (OK(xFSMount(&myVolume, blockDeviceUID))) {
  // Volume mounted - can now open files xFile file;
  if (OK(xFileOpen(&file, myVolume, (xByte*)"/data.txt", FS_MODE_READ))) {
    // File opened successfully xFileClose(file);
  }
  // Always unmount when done xFSUnmount(myVolume);
}
```

Example 2: System initialization with mount

```
 xVolume systemVolume;
xHalfWord storageUID;
void initStorage(void) {
  // Register and configure block device if
(OK(__BlockDeviceRegister__(&storageUID))) {
    BlockDeviceConfig_t config;
    config.protocol = BLOCK_PROTOCOL_RAW;
    config.backingDeviceUID = ramDiskUID;
    if (OK(__DeviceConfigDevice__(storageUID, sizeof(config),
(xByte*)&config))) {
      // Try to mount existing filesystem if (ERROR(xFSMount(&systemVolume,
storageUID))) {
        // Mount failed - format and mount fresh filesystem if
(OK(xFSFormat(storageUID, (xByte*)"SYSTEM"))) {
          xFSMount(&systemVolume, storageUID);
        }
```

```
      }
    }
  }
}
```

Example 3: Multi-volume system

```
 xVolume dataVolume;
xVolume configVolume;
xHalfWord dataDeviceUID = 1;
xHalfWord configDeviceUID = 2;
void mountAllVolumes(void) {
  // Mount data volume if (OK(xFSMount(&dataVolume, dataDeviceUID))) {
    // Mount config volume if (OK(xFSMount(&configVolume,
configDeviceUID))) {
      // Both volumes mounted - use different handles xFileOpen(&file1,
dataVolume, (xByte*)"/data.bin", FS_MODE_READ);
      xFileOpen(&file2, configVolume, (xByte*)"/config.txt", FS_MODE_READ);
    }
  }
}
```

Example 4: Mount with volume information query

```
 xVolume vol;
xHalfWord devUID = 1;
if (OK(xFSMount(&vol, devUID))) {
  // Query volume information xVolumeInfo info;
  if (OK(xFSGetVolumeInfo(vol, &info))) {
    // Check available space if (info.freeBytes > (100 * 1024)) {  // 100
KB free
      // Sufficient space for operations performFileOperations(vol);
    }
  }
  xFSUnmount(vol);
}
```

Parameters

out	*volume_*	Pointer to xVolume handle to be initialized. On success, this handle is used in all subsequent file and directory operations. The handle remains valid until xFSUnmount() is called.
in	*blockDevice↩ UID_*	UID of the block device containing the FAT32 filesystem. Must be a registered block device that has been formatted with xFSFormat().

Returns

ReturnOK if mount succeeded, ReturnError if mount failed due to invalid block device UID, device not available, invalid FAT32 filesystem, corrupted boot sector, or memory allocation failure.

Warning

The block device must contain a valid FAT32 filesystem created by xFSFormat() or another FAT32-compatible tool. Attempting to mount an unformatted or corrupted device will fail.

Only one volume can be mounted per block device at a time. Attempting to mount an already-mounted device will fail.

The volume handle must remain valid for the duration of all file operations. Do not unmount a volume while files are still open.

Note

The volume handle is allocated from kernel memory and persists until xFSUnmount() is called. Always unmount volumes when finished to free kernel resources.

Multiple volumes can be mounted simultaneously if you have multiple block devices, each with its own volume handle.

After mounting, use xFSGetVolumeInfo() to query volume capacity, free space, and other filesystem parameters.

See also

xFSUnmount() - Unmount volume and free resources

xFSFormat() - Format block device with FAT32 filesystem

xFSGetVolumeInfo() - Query volume information

xFileOpen() - Open file on mounted volume

xDirOpen() - Open directory on mounted volume

4.2.4.32 xFSUnmount() `xReturn xFSUnmount ( xVolume volume_ )`

Unmounts a previously mounted FAT32 volume, flushing any pending writes and freeing all associated kernel resources. After unmounting, the volume handle becomes invalid and must not be used in any subsequent filesystem operations.

Unmounting ensures filesystem consistency by finalizing all pending operations and is essential for proper resource management in embedded systems. Always unmount volumes before system shutdown, device removal, or when the filesystem is no longer needed.

Unmount process:

1. Validates volume handle 2. Flushes any cached filesystem data to block device 3. Updates FAT tables if necessary 4. Frees volume structure from kernel memory 5. Invalidates volume handle

Common scenarios:

- **System shutdown**: Unmount all volumes before power-off
- **Resource cleanup**: Free kernel memory when filesystem not needed
- **Device removal**: Safely unmount before removing storage media
- **Filesystem switching**: Unmount before reformatting or mounting different volume

Example 1: Basic mount/unmount lifecycle

```
 xVolume vol;
xHalfWord deviceUID = 1;
// Mount volume if (OK(xFSMount(&vol, deviceUID))) {
  // Perform file operations xFile file;
  xFileOpen(&file, vol, (xByte*)"/data.txt", FS_MODE_WRITE |
FS_MODE_CREATE);
  xFileWrite(file, 5, (xByte*)"Hello");
  xFileClose(file);
  // Always unmount when done xFSUnmount(vol);
  // vol is now invalid - do not use
}
```

Example 2: System shutdown cleanup

```
 xVolume dataVolume;
xVolume configVolume;
void shutdownFilesystems(void) {
  // Unmount all volumes before shutdown xFSUnmount(dataVolume);
  xFSUnmount(configVolume);
  // Volumes now unmounted and resources freed
}
```

Example 3: Safe volume handle cleanup

```
 xVolume vol = null;
xHalfWord devUID = 1;
void useVolume(void) {
  if (OK(xFSMount(&vol, devUID))) {
    performOperations(vol);
    // Unmount and nullify handle xFSUnmount(vol);
    vol = null;  // Prevent use-after-unmount
  }
}
```

Parameters

in	*volume_*	Handle to the mounted volume to unmount. Must be a valid volume obtained from a previous successful xFSMount() call. After unmounting, this handle becomes invalid.

Returns

ReturnOK if unmount succeeded, ReturnError if unmount failed due to invalid volume handle or volume not found.

Warning

All files and directories must be closed before unmounting. Unmounting a volume with open file handles may cause resource leaks or undefined behavior.

After calling xFSUnmount(), the volume handle becomes invalid and must not be used in any filesystem operations. Attempting to use an unmounted volume will result in ReturnError.

Always unmount volumes before reformatting the block device or removing storage media to prevent filesystem corruption.

Note

It is good practice to set the volume handle to null after unmounting to prevent accidental use of an invalid handle.

Unmounting frees kernel memory allocated during xFSMount(). In long-running systems, always unmount volumes when they are no longer needed.

See also

xFSMount() - Mount a FAT32 filesystem volume

xFileClose() - Close files before unmounting

xDirClose() - Close directories before unmounting

xFileSync() - Explicitly flush file data before unmount

4.2.4.33 xMemAlloc()

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

Allocates a block of memory from the HeliOS user heap and returns a pointer to the allocated memory. The allocated memory is automatically zeroed (similar to calloc() in standard C), ensuring predictable initialization.

HeliOS maintains separate user and kernel memory regions. This function allocates from the user heap, which is intended for application data structures, buffers, and general-purpose memory needs. The total heap size is determined by CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS × CONFIG_MEMORY_REGION_BLOCK_SIZE.

Memory Allocation Features:

- Automatic zero-initialization of allocated memory
- Best-fit allocation strategy to minimize fragmentation
- Memory safety checks to prevent heap corruption
- Tracking of allocation statistics (via xMemGetHeapStats())

Memory allocated by this function must be freed using xMemFree() when no longer needed to prevent memory leaks. HeliOS does not provide automatic garbage collection.

Warning

The addr_ parameter must be cast to (volatile xAddr *) to avoid compiler warnings. This is because the function modifies the pointer variable itself (by-reference parameter passing).

Note

Allocation may fail if insufficient contiguous memory is available, even if the total free memory exceeds the requested size. This can occur due to heap fragmentation.

Several HeliOS functions allocate heap memory internally and return it to the caller (e.g., xDeviceRead(), xMemGetHeapStats()). Memory returned by these functions must also be freed with xMemFree().

Example Usage:

```
// Allocate memory for a structure typedef struct {
  int temperature;
  int humidity;
} SensorData_t;
SensorData_t *data = NULL;
if (OK(xMemAlloc((volatile xAddr *)&data, sizeof(SensorData_t)))) {
  // Memory allocated successfully and zeroed data->temperature =
readTemperature();
  data->humidity = readHumidity();
  // Use the data...
  // Free when done xMemFree((xAddr)data);
} else {
  // Handle allocation failure reportError("Out of memory");
}
// Allocate an array uint8_t *buffer = NULL;
if (OK(xMemAlloc((volatile xAddr *)&buffer, 256))) {
  // Use buffer...
  xMemFree((xAddr)buffer);
}
```

Parameters

out	*addr_*	Pointer to a pointer variable that will receive the address of the allocated memory. Must be cast to (volatile xAddr *). On failure, this pointer is not modified.
in	*size_*	Number of bytes to allocate. Must be greater than zero.

Returns

ReturnOK if memory was successfully allocated, ReturnError if allocation failed (insufficient memory, invalid parameters, or memory system error).

See also

xMemFree() - Free allocated memory

xMemFreeAll() - Free all allocated memory

xMemGetUsed() - Get total allocated memory

xMemGetSize() - Get size of an allocation

xMemGetHeapStats() - Get detailed heap statistics

CONFIG_MEMORY_REGION_SIZE_IN_BLOCKS - Heap size configuration

CONFIG_MEMORY_REGION_BLOCK_SIZE - Memory block size configuration

4.2.4.34 xMemFree()

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

Deallocates a block of memory that was previously allocated by xMemAlloc() or returned by HeliOS functions that allocate memory (such as xDeviceRead(), xMemGetHeapStats(), or xTaskGetAllRunTimeStats()). The freed memory becomes available for future allocations.

After freeing memory, the pointer should be considered invalid and must not be dereferenced. HeliOS does not automatically NULL the pointer; the caller is responsible for proper pointer management.

Memory Management Best Practices:

- Always free memory when it is no longer needed to prevent leaks
- Set pointers to NULL after freeing to avoid use-after-free bugs
- Never free the same memory twice (double-free)
- Never free memory that was not allocated by xMemAlloc() or HeliOS
- Never free stack-allocated variables or static memory

Warning

Freeing invalid memory addresses or freeing the same memory twice will cause heap corruption and undefined behavior. HeliOS performs validation checks, but cannot detect all misuse scenarios.

After calling xMemFree(), do not access the freed memory. Doing so results in undefined behavior and may cause data corruption or system crashes.

Note

Some HeliOS functions allocate memory and return it to the caller. The caller is responsible for freeing this memory. Check function documentation to determine if memory management is required.

Example Usage:

```
// Proper memory management uint8_t *buffer = NULL;
if (OK(xMemAlloc((volatile xAddr *)&buffer, 128))) {
  // Use the buffer processData(buffer, 128);
  // Free when done if (OK(xMemFree((xAddr)buffer))) {
    buffer = NULL;  // Good practice: NULL the pointer
  }
}
// Freeing memory returned by HeliOS functions xMemoryRegionStats *stats =
NULL;
if (OK(xMemGetHeapStats(&stats))) {
  // Use stats...
  printf("Free bytes: %lu\n", stats->availableSpaceInBytes);
  // Must free the stats structure xMemFree((xAddr)stats);
  stats = NULL;
}
```

Parameters

in	*addr_*	Pointer to the memory block to free. Must be a valid pointer previously returned by xMemAlloc() or a HeliOS allocation function. Passing NULL is safe and results in no operation.

Returns

ReturnOK if memory was successfully freed, ReturnError if the operation failed (invalid address, double-free attempt, or memory system error).

See also

xMemAlloc() - Allocate memory from the heap

xMemFreeAll() - Free all allocated memory at once

xMemGetUsed() - Get total allocated memory

xMemGetSize() - Get size of an allocation

xMemGetHeapStats() - Get detailed heap statistics

4.2.4.35 xMemFreeAll()

```
xReturn xMemFreeAll (
        void  )
```

The xMemFreeAll() syscall frees (i.e., de-allocates) all heap memory allocated by xMemAlloc(). Caution should be used when calling xMemFreeAll() as all references to the heap memory region will be made invalid.

Returns

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))) {}).

4.2.4.36 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.

See also

xReturn

xMemoryRegionStats

xMemFree()

Parameters

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

Returns

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))) {}).

4.2.4.37 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.

See also

xReturn

xMemoryRegionStats

xMemFree()

Parameters

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

Returns

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))) {}).

4.2.4.38 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().

See also

xReturn

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

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))) {}).

4.2.4.39 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.

See also

xReturn

xMemGetHeapStats()

Parameters

*size_*	The size (i.e., amount), in bytes, of in-use heap memory.

Returns

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))) {}).

4.2.4.40 xQueueCreate()

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

Creates a message queue that enables tasks to exchange data safely in a producer-consumer pattern. Queues provide a First-In-First-Out (FIFO) mechanism for passing messages between tasks, making them ideal for decoupling tasks and implementing asynchronous communication.

Message queues in HeliOS store variable-length byte arrays, allowing flexible message formats. Each queue has a configurable maximum capacity (limit), and operations fail gracefully when the queue is full or empty.

Queue Characteristics:

- FIFO ordering: Messages are retrieved in the order they were sent
- Bounded capacity: Queue has a maximum message limit
- Non-blocking: Send and receive operations return immediately
- Thread-safe: Can be safely accessed from multiple tasks
- Variable message size: Each message can be a different size

Common Use Cases:

- Passing sensor data from interrupt handlers to processing tasks
- Command queues for serial communication
- Event queues for state machines
- Data buffering between producer and consumer tasks
- Decoupling hardware drivers from application logic

Queue Operation Flow:

1. Producer task sends message with xQueueSend() 2. Message is stored in queue if space available 3. Consumer task receives message with xQueueReceive() 4. Message is removed from queue

Note

The limit parameter must be at least CONFIG_QUEUE_MINIMUM_LIMIT (default 5). This ensures reasonable queue capacity and prevents overly small queues that could cause frequent overflow conditions.

Messages are copied into the queue, so the original data buffer can be reused or freed after calling xQueueSend(). Similarly, xQueueReceive() provides a copy of the message.

Queues can be locked with xQueueLockQueue() to prevent modifications during critical operations, then unlocked with xQueueUnLockQueue().

Example Usage:

```
 xQueue sensorDataQueue;
xQueue commandQueue;
// Create a queue that can hold up to 10 messages if
(OK(xQueueCreate(&sensorDataQueue, 10))) {
  // Queue created successfully
  // Producer task: Send sensor readings void sensorTask(xTask task,
xTaskParm parm) {
    uint8_t sensorData[4];
    readSensor(sensorData, sizeof(sensorData));
    // Send to queue xSize dataSize = sizeof(sensorData);
    if (OK(xQueueSend(sensorDataQueue, dataSize, sensorData))) {
      // Data queued successfully
    } else {
      // Queue full - handle overflow
    }
  }
  // Consumer task: Process sensor readings void processingTask(xTask task,
xTaskParm parm) {
    xQueueMessage message;
    // Check if messages available xBase messagesWaiting;
    if (OK(xQueueMessagesWaiting(sensorDataQueue, &messagesWaiting))) {
      if (messagesWaiting > 0) {
        // Receive message if (OK(xQueueReceive(sensorDataQueue,
&message))) {
          // Process the data processSensorData(message.message,
message.size);
        }
      }
    }
  }
}
// Create a command queue if (OK(xQueueCreate(&commandQueue, 5))) {
  // Use queue for command processing...
  // Clean up when done xQueueDelete(commandQueue);
}
```

Parameters

out	*queue_*	Pointer to xQueue variable that will receive the queue handle. This handle is used in subsequent queue operations.
in	*limit_*	Maximum number of messages the queue can hold. Must be at least CONFIG_QUEUE_MINIMUM_LIMIT (default 5). When this limit is reached, the queue is full and xQueueSend() will fail.

Returns

ReturnOK if the queue was successfully created, ReturnError if creation failed (out of memory, invalid limit, or system error).

See also

xQueueDelete() - Delete a queue and free its resources

xQueueSend() - Send a message to a queue

xQueueReceive() - Receive a message from a queue

xQueuePeek() - Examine next message without removing it

xQueueMessagesWaiting() - Get number of messages in queue

xQueueIsQueueFull() - Check if queue is at capacity

xQueueIsQueueEmpty() - Check if queue has no messages

xQueueLockQueue() - Lock queue to prevent modifications

CONFIG_QUEUE_MINIMUM_LIMIT - Minimum queue capacity configuration

4.2.4.41 xQueueDelete() `xReturn xQueueDelete ( xQueue queue_ )`

Permanently removes a message queue and releases all associated kernel resources, including all queued messages. After deletion, the queue handle becomes invalid and must not be used in any subsequent queue operations. This operation is typically performed during cleanup, reconfiguration, or when a communication channel is no longer needed.

xQueueDelete() immediately removes the queue and all its messages from the kernel, freeing both the queue structure and all message memory. Any messages that were waiting in the queue are lost—there is no mechanism to retrieve them after deletion. Tasks attempting to send or receive on a deleted queue will receive ReturnError.

Key characteristics:

- **Immediate deletion**: Queue and all messages removed immediately
- **Message loss**: All pending messages are discarded
- **Resource cleanup**: All kernel memory associated with queue is freed
- **Handle invalidation**: Queue handle cannot be reused after deletion
- **Non-blocking**: Returns immediately after cleanup completes

Common deletion scenarios:

- **Task cleanup**: Remove queues during task shutdown
- **Dynamic reconfiguration**: Delete and recreate queues with different limits
- **Resource reclamation**: Free unused queues to reduce memory usage
- **Error recovery**: Clean up queues after initialization failures

Example 1: Queue lifecycle management

```
 xQueue commandQueue;
void initCommandQueue(void) {
  if (OK(xQueueCreate(&commandQueue, 10))) {
    // Use queue for communication
  }
}
void shutdownCommandQueue(void) {
  // Delete queue when no longer needed if (OK(xQueueDelete(commandQueue)))
{
    // Queue and all messages freed
  }
}
```

Example 2: Dynamic queue reconfiguration

```
 xQueue dataQueue;
void resizeQueue(xBase newLimit) {
  // Delete existing queue if (dataQueue != null) {
    xQueueDelete(dataQueue);
  }
  // Create new queue with different limit if (OK(xQueueCreate(&dataQueue,
newLimit))) {
    // New queue ready
  }
}
```

Example 3: Multiple queue cleanup

```
 xQueue queues[MAX_CHANNELS];
xBase queueCount = 0;
void initQueues(void) {
  for (xBase i = 0; i < MAX_CHANNELS; i++) {
    if (OK(xQueueCreate(&queues[i], 5))) {
      queueCount++;
    }
  }
}
void shutdownQueues(void) {
  for (xBase i = 0; i < queueCount; i++) {
    xQueueDelete(queues[i]);
  }
  queueCount = 0;
}
```

Example 4: Conditional queue cleanup

```
 xQueue eventQueue;
xBase queueActive = 0;
void disableEvents(void) {
  if (queueActive) {
    xQueueDelete(eventQueue);
    queueActive = 0;
  }
}
void enableEvents(void) {
  if (!queueActive) {
    if (OK(xQueueCreate(&eventQueue, 8))) {
      queueActive = 1;
    }
  }
}
```

Parameters

in	*queue*↩ _	Handle to the queue to delete. Must be a valid queue created with xQueueCreate(). After deletion, this handle becomes invalid.

Returns

ReturnOK if queue deleted successfully, ReturnError if deletion failed due to invalid queue handle or queue not found.

Warning

All messages pending in the queue are permanently lost when the queue is deleted. If you need to preserve messages, drain the queue with xQueueReceive() before calling xQueueDelete().

After xQueueDelete() returns successfully, the queue handle is invalid and must not be used in any subsequent operations. Using a deleted queue handle will result in ReturnError.

Deleting a queue that is actively used by multiple tasks can lead to errors if those tasks attempt to access the queue after deletion. Coordinate queue deletion across all tasks using the queue.

Note

xQueueDelete() frees all memory associated with the queue, including the queue structure and all message memory. This is the only way to reclaim kernel memory allocated for a queue.

Unlike stopping a timer, which preserves the timer for reuse, deleting a queue permanently removes it. To reuse queue functionality, create a new queue with xQueueCreate().

See also

xQueueCreate() - Create a message queue

xQueueReceive() - Receive messages before deletion

xQueueMessagesWaiting() - Check for pending messages

xTaskDelete() - Delete a task (similar resource cleanup pattern)

xTimerDelete() - Delete a timer (similar resource cleanup pattern)

4.2.4.42 xQueueDropMessage() `xReturn xQueueDropMessage ( xQueue queue_ )`

Removes and discards the oldest message from the queue without retrieving its contents. This operation decrements the message count and frees the message memory, but does not return the message data to the caller. Use this when you want to skip or discard messages without processing them.

Unlike xQueueReceive() which allocates and returns the message, xQueueDropMessage() simply removes the message and frees its memory immediately. This is more efficient when message content is not needed—for example, when clearing stale messages, handling overflow conditions, or implementing message filtering.

Common use cases:

- **Queue clearing**: Discard all messages when resetting state
- **Message filtering**: Skip unwanted messages after peeking
- **Overflow handling**: Drop old messages when queue backs up
- **Selective processing**: Discard messages based on peek inspection

Example 1: Clear all messages from queue

```
 xQueue eventQueue;
void clearQueue(void) {
  xBase isEmpty;
  if (OK(xQueueIsQueueEmpty(eventQueue, &isEmpty))) {
    while (!isEmpty) {
      xQueueDropMessage(eventQueue);
      xQueueIsQueueEmpty(eventQueue, &isEmpty);
    }
  }
}
```

Example 2: Selective message processing with peek and drop

```
 xQueue dataQueue;
void processOnlyValidMessages(void) {
  xBase waiting;
  if (OK(xQueueMessagesWaiting(dataQueue, &waiting))) {
    for (xBase i = 0; i < waiting; i++) {
      xQueueMessage msg;
      if (OK(xQueuePeek(dataQueue, &msg))) {
        xByte msgType = msg.value[0];
        xMemFree((xAddr)msg.value);  // Free peek copy
        if (isValidMessageType(msgType)) {
          // Valid - receive and process if (OK(xQueueReceive(dataQueue,
&msg))) {
            processMessage(&msg);
            xMemFree((xAddr)msg.value);
          }
        } else {
          // Invalid - drop without retrieving
xQueueDropMessage(dataQueue);
        }
      }
    }
  }
}
```

Example 3: Drop stale messages on timeout

```
 xQueue timeStampedQueue;
xTimer timeoutTimer;
void dropStaleMessages(void) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(timeoutTimer, &expired)) && expired) {
    // Timeout elapsed - drop oldest message
xQueueDropMessage(timeStampedQueue);
    xTimerReset(timeoutTimer);
  }
}
```

Parameters

in	*queue*↩_	Handle to the queue from which to drop a message. Must be a valid queue created with xQueueCreate().

Returns

ReturnOK if message dropped successfully, ReturnError if operation failed due to empty queue or invalid queue handle.

Warning

The dropped message is permanently lost—there is no way to retrieve it after calling xQueueDropMessage(). If you need the message content, use xQueueReceive() instead.

Note

xQueueDropMessage() is more efficient than xQueueReceive() followed by xMemFree() because it avoids allocating memory for the message copy.

If the queue is empty, xQueueDropMessage() returns ReturnError immediately.

See also

xQueueReceive() - Receive and retrieve message

xQueuePeek() - Examine message without removing it

xQueueMessagesWaiting() - Check message count before dropping

xQueueIsQueueEmpty() - Check if queue has messages

4.2.4.43 xQueueGetLength()

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

Retrieves the maximum number of messages (limit) that a queue can hold, as configured during queue creation with xQueueCreate(). This non-destructive query returns the queue's capacity, not the number of messages currently waiting. The limit represents the upper bound on how many messages can be queued before xQueueSend() returns ReturnError due to queue full.

Note: Despite the name "GetLength", this function returns the queue's maximum capacity (limit), not the current message count. To get the number of messages currently waiting, use xQueueMessagesWaiting().

Common use cases:

- **Capacity validation**: Verify queue created with expected limit
- **Flow control**: Calculate available space before bulk sends
- **Diagnostics**: Report queue configuration for debugging
- **Dynamic adjustment**: Determine if queue needs resizing

Example 1: Validate queue configuration

```
 xQueue dataQueue;
xBase expectedLimit = 10;
if (OK(xQueueCreate(&dataQueue, expectedLimit))) {
  xBase actualLimit;
  if (OK(xQueueGetLength(dataQueue, &actualLimit))) {
    if (actualLimit == expectedLimit) {
      // Queue configured correctly
    }
  }
}
```

Example 2: Calculate available queue space

```
 xQueue commandQueue;
xBase getAvailableSpace(void) {
  xBase limit, waiting;
  if (OK(xQueueGetLength(commandQueue, &limit)) &&
      OK(xQueueMessagesWaiting(commandQueue, &waiting))) {
    return limit - waiting;
  }
  return 0;
}
```

Example 3: Queue utilization monitoring

```
 xQueue eventQueue;
void reportQueueStatus(void) {
  xBase limit, waiting;
  if (OK(xQueueGetLength(eventQueue, &limit)) &&
      OK(xQueueMessagesWaiting(eventQueue, &waiting))) {
    xBase utilization = (waiting * 100) / limit;
    printf("Queue: %d/%d messages (%d%% full)\n", waiting, limit,
utilization);
  }
}
```

Parameters

in	*queue_*	Handle to the queue to query. Must be a valid queue created with xQueueCreate().
out	*res_*	Pointer to variable receiving the queue limit (maximum capacity). On success, contains the limit value specified during xQueueCreate().

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid queue handle or queue not found.

Warning

This function returns the queue's maximum capacity (limit), NOT the current number of messages. Use xQueueMessagesWaiting() to get the current message count.

Note

The returned limit is constant for the lifetime of the queue—it cannot be changed after creation. To change capacity, delete and recreate the queue.

See also

xQueueMessagesWaiting() - Get current message count

xQueueIsQueueFull() - Check if queue is at capacity

xQueueIsQueueEmpty() - Check if queue has no messages

xQueueCreate() - Create queue with specified limit

4.2.4.44 xQueueIsQueueEmpty()

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

Queries whether a message queue is empty (contains zero messages). This non-destructive check allows tasks to determine if messages are available before attempting to receive, enabling conditional receive logic and avoiding unnecessary xQueueReceive() calls on empty queues. A queue is considered empty when it contains no messages, regardless of its maximum capacity.

Common use cases:

- **Conditional receive**: Only call xQueueReceive() if messages available
- **Flow control**: Check for messages before processing
- **Diagnostics**: Monitor queue activity
- **Polling loops**: Detect when queue has been drained

Example 1: Conditional message processing

```
 xQueue eventQueue;
void processEvents(void) {
  xBase isEmpty;
  if (OK(xQueueIsQueueEmpty(eventQueue, &isEmpty)) && !isEmpty) {
    // Queue has messages - process them xQueueMessage msg;
    while (OK(xQueueReceive(eventQueue, &msg))) {
      handleEvent(&msg);
      xMemFree((xAddr)msg.value);
    }
  }
}
```

Example 2: Wait for messages

```
 xQueue dataQueue;
void waitForData(void) {
  xBase isEmpty = 1;
  // Poll until message arrives while (isEmpty) {
    if (OK(xQueueIsQueueEmpty(dataQueue, &isEmpty))) {
      if (isEmpty) {
        xTaskWait(10);  // Wait before checking again
      }
    }
  }
  // Message available - process it xQueueMessage msg;
  if (OK(xQueueReceive(dataQueue, &msg))) {
    processData(&msg);
    xMemFree((xAddr)msg.value);
  }
}
```

Example 3: Drain queue completely

```
 xQueue commandQueue;
void drainQueue(void) {
  xBase isEmpty;
  if (OK(xQueueIsQueueEmpty(commandQueue, &isEmpty))) {
    while (!isEmpty) {
      xQueueMessage msg;
      if (OK(xQueueReceive(commandQueue, &msg))) {
        xMemFree((xAddr)msg.value);
      }
      xQueueIsQueueEmpty(commandQueue, &isEmpty);
    }
  }
}
```

Parameters

in	*queue_*	Handle to the queue to query. Must be a valid queue created with xQueueCreate().
out	*res_*	Pointer to variable receiving the empty status. Set to non-zero (true) if queue is empty, zero (false) if queue contains one or more messages.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid queue handle or queue not found.

Note

This is a snapshot query—in multitasking environments, the queue state may change immediately after the check if other tasks send messages.

xQueueIsQueueEmpty() is equivalent to checking if xQueueMessagesWaiting() returns 0, but provides a more readable API for empty checks.

See also

xQueueMessagesWaiting() - Get exact message count

xQueueIsQueueFull() - Check if queue is at capacity

xQueueReceive() - Receive message from queue

xQueuePeek() - Check message without removing it

4.2.4.45 xQueueIsQueueFull()

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

Queries whether a message queue is full (at maximum capacity). This check allows tasks to determine if there is space available before attempting to send, enabling flow control and preventing message loss. A queue is considered full when the number of waiting messages equals the limit specified during xQueueCreate(). When full, xQueueSend() will return ReturnError.

Common use cases:

- **Conditional send**: Only send if space available
- **Flow control**: Back off when queue approaches capacity
- **Overflow prevention**: Detect and handle full queue conditions
- **Diagnostics**: Monitor queue utilization and congestion

Example 1: Conditional message send

```
 xQueue commandQueue;
xReturn sendCommand(xByte *cmd, xBase size) {
  xBase isFull;
  if (OK(xQueueIsQueueFull(commandQueue, &isFull))) {
    if (isFull) {
      logWarning("Command queue full - dropping command");
      return ReturnError;
    }
    // Space available - send message return xQueueSend(commandQueue, size,
cmd);
  }
  return ReturnError;
}
```

Example 2: Flow control with backoff

```
 xQueue dataQueue;
void sendDataWithBackoff(xByte *data, xBase size) {
  xBase isFull;
  xBase retries = 0;
  while (retries < 10) {
    if (OK(xQueueIsQueueFull(dataQueue, &isFull))) {
      if (!isFull) {
        // Space available - send now if (OK(xQueueSend(dataQueue, size,
data))) {
          return;
        }
      }
    }
    // Queue full - wait and retry xTaskWait(10);
    retries++;
  }
  logError("Failed to send data - queue full");
}
```

Example 3: Monitor queue pressure

```
 xQueue eventQueue;
void monitorQueuePressure(void) {
  xBase isFull, waiting;
  if (OK(xQueueIsQueueFull(eventQueue, &isFull)) && isFull) {
    logWarning("Event queue at capacity");
  } else if (OK(xQueueMessagesWaiting(eventQueue, &waiting))) {
    xBase limit;
    if (OK(xQueueGetLength(eventQueue, &limit))) {
      if (waiting > (limit * 80) / 100) {
        logWarning("Event queue nearly full: %d/%d", waiting, limit);
      }
    }
  }
}
```

Parameters

in	*queue_*	Handle to the queue to query. Must be a valid queue created with xQueueCreate().
out	*res_*	Pointer to variable receiving the full status. Set to non-zero (true) if queue is full, zero (false) if queue has space for more messages.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid queue handle or queue not found.

Note

This is a snapshot query—in multitasking environments, the queue state may change immediately after the check if other tasks receive messages.

xQueueIsQueueFull() checks if message count equals the limit. To check for nearly-full conditions, use xQueueMessagesWaiting() and compare to xQueueGetLength().

See also

xQueueMessagesWaiting() - Get exact message count

xQueueIsQueueEmpty() - Check if queue has no messages

xQueueGetLength() - Get queue capacity

xQueueSend() - Send message (fails when queue is full)

4.2.4.46 xQueueLockQueue() `xReturn xQueueLockQueue ( xQueue queue_ )`

Places a queue in locked state, preventing tasks from sending new messages with xQueueSend(). While locked, send operations return ReturnError immediately. However, locked queues still allow receiving, peeking, and dropping messages— only send operations are blocked. This enables controlled queue drainage and prevents queue overflow during critical processing.

Queue locking is useful for implementing flow control, preventing message accumulation during batch processing, or ensuring a queue is drained before reconfiguration. Locks must be explicitly released with xQueueUnLockQueue() to resume normal operation—they do not timeout or automatically unlock.

Common use cases:

- **Batch processing**: Lock queue while processing all pending messages
- **Flow control**: Prevent producer overrun during high processing load
- **Queue drainage**: Ensure queue is empty before shutdown or reconfiguration
- **Critical sections**: Prevent message accumulation during critical operations

Example 1: Batch processing with lock

```
 xQueue eventQueue;
void processBatch(void) {
  // Lock to prevent new messages during batch xQueueLockQueue(eventQueue);
  // Process all current messages xBase isEmpty;
  if (OK(xQueueIsQueueEmpty(eventQueue, &isEmpty))) {
    while (!isEmpty) {
      xQueueMessage msg;
      if (OK(xQueueReceive(eventQueue, &msg))) {
        handleEvent(&msg);
        xMemFree((xAddr)msg.value);
      }
      xQueueIsQueueEmpty(eventQueue, &isEmpty);
    }
  }
  // Unlock to resume message sending xQueueUnLockQueue(eventQueue);
}
```

Example 2: Controlled shutdown

```
 xQueue commandQueue;
void shutdownCommandProcessor(void) {
  // Lock queue to prevent new commands xQueueLockQueue(commandQueue);
  // Process remaining commands xQueueMessage msg;
  while (OK(xQueueReceive(commandQueue, &msg))) {
    executeCommand(&msg);
    xMemFree((xAddr)msg.value);
  }
  // Delete queue (no need to unlock) xQueueDelete(commandQueue);
}
```

Parameters

in	*queue*↩ _	Handle to the queue to lock. Must be a valid queue created with xQueueCreate().

Returns

ReturnOK if queue locked successfully, ReturnError if operation failed due to invalid queue handle or queue not found.

Warning

Locking a queue does NOT prevent receiving, peeking, or dropping messages—only xQueueSend() is blocked. Consumers can still drain the queue while it is locked.

Locks must be explicitly released with xQueueUnLockQueue(). There is no automatic timeout or unlock mechanism.

Note

Locking an already-locked queue is safe and has no effect.

xQueueSend() on a locked queue returns ReturnError immediately without blocking or queuing the message.

See also

xQueueUnLockQueue() - Unlock queue to resume sending

xQueueSend() - Send message (fails on locked queue)

xQueueReceive() - Receive message (works on locked queue)

4.2.4.47 xQueueMessagesWaiting()

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

Retrieves the count of messages currently waiting in a queue. This non-destructive query provides precise queue occupancy information, allowing tasks to make informed decisions about message processing, flow control, and resource allocation. The count represents messages successfully sent with xQueueSend() but not yet received with xQueueReceive().

Common use cases:

- **Batch processing**: Process multiple messages when threshold reached
- **Queue utilization**: Monitor queue load and congestion
- **Flow control**: Throttle senders based on queue depth
- **Diagnostics**: Report queue activity for debugging

Example 1: Batch message processing

```
 xQueue eventQueue;
#define BATCH_THRESHOLD 5
void processBatchEvents(void) {
  xBase waiting;
  if (OK(xQueueMessagesWaiting(eventQueue, &waiting))) {
    if (waiting >= BATCH_THRESHOLD) {
      // Process messages in batch for efficiency for (xBase i = 0; i <
waiting; i++) {
        xQueueMessage msg;
        if (OK(xQueueReceive(eventQueue, &msg))) {
          handleEvent(&msg);
          xMemFree((xAddr)msg.value);
        }
      }
    }
  }
}
```

Example 2: Queue utilization monitoring

```
 xQueue commandQueue;
void reportQueueStats(void) {
  xBase waiting, limit;
  if (OK(xQueueMessagesWaiting(commandQueue, &waiting)) &&
      OK(xQueueGetLength(commandQueue, &limit))) {
    xBase utilization = (waiting * 100) / limit;
    printf("Queue: %d/%d messages (%d%% full)\n", waiting, limit,
utilization);
    if (utilization > 90) {
      logWarning("Queue nearly full - consider increasing capacity");
    }
  }
}
```

Example 3: Producer flow control

```
 xQueue dataQueue;
#define HIGH_WATER_MARK 8
#define LOW_WATER_MARK 3
static xBase producerThrottled = 0;
void produceData(xByte *data, xBase size) {
  xBase waiting;
  if (OK(xQueueMessagesWaiting(dataQueue, &waiting))) {
    // Implement hysteresis for flow control if (waiting >=
HIGH_WATER_MARK) {
      producerThrottled = 1;
      logInfo("Producer throttled - queue depth: %d", waiting);
    } else if (waiting <= LOW_WATER_MARK) {
      producerThrottled = 0;
    }
    if (!producerThrottled) {
      xQueueSend(dataQueue, size, data);
    }
  }
}
```

Parameters

in	*queue_*	Handle to the queue to query. Must be a valid queue created with xQueueCreate().
out	*res_*	Pointer to variable receiving the message count. On success, contains the number of messages currently waiting (0 to limit).

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid queue handle or queue not found.

Note

This is a snapshot query—in multitasking environments, the message count may change immediately after the query if other tasks send or receive messages.

The count includes all messages from the head to the tail of the queue. Messages accessed with xQueuePeek() are still counted as waiting.

See also

xQueueIsQueueEmpty() - Check if count is zero (no messages)

xQueueIsQueueFull() - Check if count equals limit (at capacity)

xQueueGetLength() - Get queue maximum capacity

xQueueReceive() - Remove and receive next message

xQueueSend() - Add message to queue

4.2.4.48 xQueuePeek()

```
xReturn xQueuePeek (
    const xQueue queue_,
    xQueueMessage * message_ )
```

Retrieves a copy of the oldest message from the queue without removing it, allowing inspection of queue contents without consuming messages. Unlike xQueueReceive(), which removes the message, xQueuePeek() leaves the queue unchanged—the message remains available for subsequent peek or receive operations. This enables conditional message processing and queue inspection.

The peeked message is allocated from the heap as a copy—the original message stays in the queue. The caller must free the peeked message copy with xMemFree() after use. Multiple peek calls return the same message until it is removed with xQueueReceive() or xQueueDropMessage().

Common use cases:

- **Conditional processing**: Inspect message before deciding to receive
- **Message filtering**: Check message type/priority before consuming
- **Queue monitoring**: Examine next message without affecting queue state
- **Priority handling**: Process high-priority messages first

Example 1: Priority message handling

```
 xQueue eventQueue;
#define MSG_PRIORITY_HIGH 1
#define MSG_PRIORITY_NORMAL 0
void processEvents(void) {
  xQueueMessage msg;
  // Peek at next message if (OK(xQueuePeek(eventQueue, &msg))) {
    xByte priority = msg.value[0];  // Assume first byte is priority
    if (priority == MSG_PRIORITY_HIGH) {
      // High priority - receive and process immediately
xMemFree((xAddr)msg.value);  // Free peek copy xQueueReceive(eventQueue,
&msg);
      handleHighPriority(&msg);
      xMemFree((xAddr)msg.value);
    } else {
      // Normal priority - skip for now xMemFree((xAddr)msg.value);  //
Free peek copy
    }
  }
}
```

Example 2: Message type filtering

```
 xQueue commandQueue;
xBase hasCommandType(xByte type) {
  xQueueMessage msg;
  // Peek without consuming if (OK(xQueuePeek(commandQueue, &msg))) {
    xByte cmdType = msg.value[0];
    xMemFree((xAddr)msg.value);
    return (cmdType == type);
  }
  return 0;
}
```

Parameters

in	*queue_*	Handle to the queue to peek. Must be a valid queue created with xQueueCreate().
out	*message_*	Pointer to xQueueMessage structure receiving a copy of the message. Memory is allocated for message->value and must be freed with xMemFree() after use.

Returns

ReturnOK if message peeked successfully, ReturnError if peek failed due to empty queue, invalid queue handle, or memory allocation failure.

Warning

The peeked message is a heap-allocated COPY. You must free it with xMemFree() even though the original message remains in the queue.

Multiple peeks return the SAME message until it is removed. If you peek, then another task receives the message, subsequent peeks will return the next message in the queue.

Note

xQueuePeek() does not modify the queue state—message count and queue position remain unchanged.

See also

xQueueReceive() - Receive and remove message

xQueueDropMessage() - Remove message without retrieving

xQueueMessagesWaiting() - Check if messages available before peeking

xMemFree() - Free peeked message copy

4.2.4.49 xQueueReceive()

```
xReturn xQueueReceive (
        xQueue queue_,
        xQueueMessage * message_ )
```

Retrieves the oldest message from the specified queue and removes it, making space for new messages. This is the primary mechanism for tasks to receive data from other tasks in a producer-consumer pattern. The operation is non-blocking and returns immediately whether a message is available or not.

This function combines the operations of xQueuePeek() (examine message) and xQueueDropMessage() (remove message) into a single atomic operation, ensuring thread-safe message retrieval even with multiple consumers.

The received message is allocated from the heap and must be freed by the caller using xMemFree() after processing. This heap allocation is necessary because message size is variable.

Message Receiving Process:

1. Check if queue has messages (not empty) 2. Allocate xQueueMessage structure from heap 3. Copy oldest message data into structure 4. Remove message from queue 5. Return message structure to caller 6. Caller processes message then frees with xMemFree()

Typical Consumer Pattern:

1. Check if messages are waiting (optional optimization) 2. Call xQueueReceive() to get message 3. Check return value - success means message received 4. Process message->message bytes (size is in message->size) 5. Free message structure with xMemFree()

Warning

The returned message structure is allocated from the heap and MUST be freed by the caller using xMemFree(). Failing to free messages will cause memory leaks and eventually exhaust available heap memory.

Note

If the queue is empty, xQueueReceive() returns ReturnError immediately without blocking. Check queue status with xQueueIsQueueEmpty() or xQueueMessagesWaiting() before receiving if needed.

The xQueueMessage structure contains:

- message->size: Number of bytes in the message
- message->message: Pointer to the message data bytes

Unlike xQueuePeek() which leaves the message in the queue, xQueueReceive() removes the message, allowing the next message to be retrieved on the next call.

Example Usage:

```
 xQueue commandQueue;
xQueueCreate(&commandQueue, 10);
// Consumer task: Process commands from queue void
commandProcessorTask(xTask task, xTaskParm parm) {
  xQueueMessage *msg = NULL;
  // Try to receive a message if (OK(xQueueReceive(commandQueue, &msg))) {
    // Message received successfully
    // Process based on message size and content if (msg->size == 1) {
      // Single byte command uint8_t cmd = msg->message[0];
      handleCommand(cmd);
    } else if (msg->size == sizeof(SensorData_t)) {
      // Structure message SensorData_t *data = (SensorData_t
*)msg->message;
      processSensorData(data);
    }
```

```
    // IMPORTANT: Free the message structure xMemFree((xAddr)msg);
  } else {
    // No messages available - queue empty
  }
}
// Complete producer-consumer example xQueue dataQueue;
// Producer void producerTask(xTask task, xTaskParm parm) {
  uint8_t data[4] = {0x01, 0x02, 0x03, 0x04};
  xQueueSend(dataQueue, sizeof(data), data);
}
// Consumer void consumerTask(xTask task, xTaskParm parm) {
  xQueueMessage *msg = NULL;
  // Check for messages first (optional optimization) xBase numMessages;
  if (OK(xQueueMessagesWaiting(dataQueue, &numMessages))) {
    if (numMessages > 0) {
      // Messages available, receive one if (OK(xQueueReceive(dataQueue,
&msg))) {
        // Process msg->message[0] through msg->message[msg->size-1]
processData(msg->message, msg->size);
        xMemFree((xAddr)msg);
      }
    }
  }
}
// Process all pending messages xBase isEmpty;
while (OK(xQueueIsQueueEmpty(dataQueue, &isEmpty)) && !isEmpty) {
  xQueueMessage *msg = NULL;
  if (OK(xQueueReceive(dataQueue, &msg))) {
    processMessage(msg);
    xMemFree((xAddr)msg);
  }
}
```

Parameters

in	*queue_*	Handle of the queue to receive from. Must be a valid queue handle previously returned by xQueueCreate().
out	*message_*	Pointer to xQueueMessage pointer variable. On success, receives a pointer to the message structure containing the message data and size. Caller MUST free this with xMemFree().

Returns

ReturnOK if a message was successfully received and removed from the queue, ReturnError if the receive failed (queue empty, invalid queue handle, memory allocation failed, or null message pointer).

See also

xQueueSend() - Send a message to the queue (producer operation)

xQueuePeek() - Examine next message without removing it

xQueueDropMessage() - Remove message after peeking

xQueueIsQueueEmpty() - Check if queue has no messages

xQueueMessagesWaiting() - Get number of messages in queue

xQueueCreate() - Create a new message queue

xMemFree() - Free the message structure (REQUIRED!)

4.2.4.50 xQueueSend()

```
xReturn xQueueSend (
        xQueue queue_,
        const xBase bytes_,
        const xByte * value_ )
```

Adds a message to the specified queue, copying the provided data into the queue's internal storage. This is the primary mechanism for tasks to send data to other tasks in a producer-consumer pattern. The operation is non-blocking and returns immediately whether successful or not.

Messages are stored in FIFO order and will be retrieved by xQueueReceive() in the same order they were sent. Each message is a byte array up to CONFIG_MESSAGE_VALUE_BYTES (default 8) bytes, allowing flexible message formats including structures, commands, sensor readings, or any other data that fits within the size limit.

Message Sending Process:

1. Check if queue has space (not full) 2. Copy message data into queue storage 3. Increment queue message count 4. Return success or failure immediately (non-blocking)

Common Message Patterns:

- **Commands**: Single byte command codes (e.g., 0x01 = START, 0x02 = STOP)
- **Sensor data**: Multi-byte readings packed into message
- **Events**: Event type and optional data
- **Pointers**: Address of larger data structure (use with caution)
- **Structures**: Small structs that fit within byte limit

Warning

Messages are limited to CONFIG_MESSAGE_VALUE_BYTES bytes (default 8). Attempting to send larger messages will fail. For larger data, consider sending a pointer to heap-allocated data or using multiple messages.

The message data is COPIED into the queue. The original buffer can be reused or freed immediately after xQueueSend() returns. The queue maintains its own copy.

Note

If the queue is full, xQueueSend() returns ReturnError immediately without blocking. Check queue status with xQueueIsQueueFull() before sending if needed, or handle send failures appropriately.

Messages are copied, not moved. This ensures the sender retains control of its data and prevents aliasing issues.

Example Usage:

```
 xQueue dataQueue;
xQueueCreate(&dataQueue, 10);
// Send a simple command byte uint8_t command = 0x42;
if (OK(xQueueSend(dataQueue, 1, &command))) {
  // Command queued successfully
}
// Send sensor reading (multi-byte) typedef struct {
  uint16_t temperature;
  uint16_t humidity;
} SensorReading_t;
SensorReading_t reading;
reading.temperature = 2350;  // 23.50°C reading.humidity = 6500;     //
65.00%
if (OK(xQueueSend(dataQueue, sizeof(SensorReading_t), (xByte *)&reading)))
{
  // Sensor data queued
} else {
  // Queue full - handle overflow handleQueueOverflow();
}
// Producer task example void sensorTask(xTask task, xTaskParm parm) {
  uint8_t sensorData[4];
  readSensors(sensorData);
  // Try to send, handle failure if (ERROR(xQueueSend(dataQueue,
sizeof(sensorData), sensorData))) {
    // Queue full - either drop data or wait dataSamplesDropped++;
  }
}
// Check queue space before sending xBase isFull;
if (OK(xQueueIsQueueFull(dataQueue, &isFull)) && !isFull) {
  // Safe to send xQueueSend(dataQueue, msgSize, msgData);
}
```

Parameters

in	*queue_*	Handle of the queue to send to. Must be a valid queue handle previously returned by xQueueCreate().
in	*bytes_*	Size of the message in bytes. Must be greater than 0 and not exceed CONFIG_MESSAGE_VALUE_BYTES (default 8).
in	*value_*	Pointer to the message data to send. The data will be copied into the queue, so this buffer can be reused after the call.

Returns

ReturnOK if the message was successfully added to the queue, ReturnError if the send failed (queue full, invalid queue handle, invalid size, or null value pointer).

See also

xQueueReceive() - Receive a message from the queue (consumer operation)

xQueuePeek() - Examine next message without removing it

xQueueIsQueueFull() - Check if queue is at capacity

xQueueMessagesWaiting() - Get number of messages in queue

xQueueCreate() - Create a new message queue

xQueueLockQueue() - Lock queue during critical operations

CONFIG_MESSAGE_VALUE_BYTES - Maximum message size configuration

4.2.4.51 xQueueUnLockQueue()

```
xReturn xQueueUnLockQueue (
        xQueue queue_ )
```

Removes the locked state from a queue, allowing tasks to resume sending messages with xQueueSend(). After unlocking, the queue returns to normal operation where send operations succeed (if queue is not full). This function must be called to restore normal queue behavior after locking with xQueueLockQueue().

Unlocking affects only send operations—receive, peek, and drop operations were unaffected by the lock and continue to work normally after unlock.

Common use cases:

- **Resume normal operation**: Re-enable message sending after batch processing
- **End critical section**: Allow producers to continue after critical operations
- **Flow control**: Resume accepting messages after backpressure relieved

Example 1: Lock/unlock pattern for batch processing

```
 xQueue dataQueue;
void batchProcessor(void) {
  // Lock to get consistent snapshot xQueueLockQueue(dataQueue);
  xBase count;
  if (OK(xQueueMessagesWaiting(dataQueue, &count))) {
    // Process exactly this many messages for (xBase i = 0; i < count; i++)
{
      xQueueMessage msg;
      if (OK(xQueueReceive(dataQueue, &msg))) {
        processBatchItem(&msg);
```

```
        xMemFree((xAddr)msg.value);
      }
    }
  }
  // Unlock to allow new messages xQueueUnLockQueue(dataQueue);
}
```

Example 2: Conditional unlock based on processing result

```
 xQueue eventQueue;
xBase processCriticalEvents(void) {
  xQueueLockQueue(eventQueue);
  xBase success = 1;
  xQueueMessage msg;
  while (OK(xQueueReceive(eventQueue, &msg)) && success) {
    if (ERROR(handleCriticalEvent(&msg))) {
      success = 0;  // Processing failed
    }
    xMemFree((xAddr)msg.value);
  }
  if (success) {
    // Success - resume normal operation xQueueUnLockQueue(eventQueue);
  } else {
    // Failure - leave locked for manual intervention logError("Critical
event processing failed - queue locked");
  }

  return success;
}
```

Parameters

in	*queue*↩ _	Handle to the queue to unlock. Must be a valid queue created with xQueueCreate().

Returns

ReturnOK if queue unlocked successfully, ReturnError if operation failed due to invalid queue handle or queue not found.

Warning

Unlocking an already-unlocked queue is safe and has no effect.

Note

Always pair xQueueLockQueue() with xQueueUnLockQueue() to avoid leaving queues permanently locked, which would prevent all future message sending.

If you delete a locked queue with xQueueDelete(), you do not need to unlock it first—the deletion frees all queue resources including lock state.

See also

xQueueLockQueue() - Lock queue to prevent sending

xQueueSend() - Send message (enabled after unlock)

xQueueReceive() - Receive message (always works)

4.2.4.52 xStreamBytesAvailable() xReturn xStreamBytesAvailable (const xStreamBuffer *stream_*, xHalfWord * *bytes_*)

Returns the count of bytes currently available for retrieval in the specified stream buffer without removing them. This non-destructive query allows tasks to check data availability before committing to receive operations, enabling more sophisticated flow control and buffering strategies.

Unlike xStreamReceive() which retrieves and removes bytes from the stream, xStreamBytesAvailable() is a read-only query that leaves the stream contents unchanged. This makes it useful for making decisions about when to process data, whether to wait for more data, or how to handle buffer capacity.

Common use cases:

- **Threshold-based processing**: Wait until minimum amount of data available
- **Flow control**: Monitor buffer levels to regulate producer rate
- **Batch optimization**: Collect data until optimal batch size reached
- **Buffer monitoring**: Check capacity before sending more data
- **Conditional receive**: Decide whether to retrieve data based on quantity

Example 1: Threshold-based data processing

```
 xStreamBuffer dataStream;
#define MIN_DATA_SIZE 16
void processDataTask(xTask task, xTaskParm parm) {
  xHalfWord available;
  // Check if enough data accumulated if
(OK(xStreamBytesAvailable(dataStream, &available))) {
    if (available >= MIN_DATA_SIZE) {
      // Enough data - retrieve and process xHalfWord count;
      xByte *data;
      if (OK(xStreamReceive(dataStream, &count, &data))) {
        processBatch(data, count);
        xMemFree((xAddr)data);
      }
    }
    // Otherwise wait for more data
  }
}
```

Example 2: Producer flow control

```
 xStreamBuffer txStream;
#define HIGH_WATER_MARK 28
#define LOW_WATER_MARK 8
void sendDataTask(xTask task, xTaskParm parm) {
  static xBase throttled = 0;
  xHalfWord buffered;
  if (OK(xStreamBytesAvailable(txStream, &buffered))) {
    // Implement hysteresis for flow control if (buffered >=
HIGH_WATER_MARK) {
      throttled = 1;  // Stop sending
    } else if (buffered <= LOW_WATER_MARK) {
      throttled = 0;  // Resume sending
    }
    if (!throttled) {
      xByte data = getNextByte();
      xStreamSend(txStream, data);
    }
  }
}
```

Example 3: Capacity checking before bulk send

```
 xStreamBuffer protocolStream;
xReturn sendFrame(xByte *frame, xHalfWord frameLen) {
  xHalfWord available;
  xHalfWord capacity;
  // Check current buffer usage if
(OK(xStreamBytesAvailable(protocolStream, &available))) {
    capacity = CONFIG_STREAM_BUFFER_BYTES - available;
    // Verify enough space for entire frame if (capacity >= frameLen) {
      // Send all bytes for (xHalfWord i = 0; i < frameLen; i++) {
```

```
        if (ERROR(xStreamSend(protocolStream, frame[i]))) {
          return ReturnError;  // Unexpected failure
        }
      }
      return ReturnOK;
    } else {
      // Not enough space - return error or wait return ReturnError;
    }
  }
  return ReturnError;
}
```

Example 4: Monitoring with status reporting

```
 xStreamBuffer sensorStream;
void monitorTask(xTask task, xTaskParm parm) {
  static xHalfWord maxObserved = 0;
  xHalfWord current;
  if (OK(xStreamBytesAvailable(sensorStream, &current))) {
    // Track high-water mark if (current > maxObserved) {
      maxObserved = current;
    }
    // Calculate utilization percentage xByte utilization =
(xByte)((current * 100) / CONFIG_STREAM_BUFFER_BYTES);
    // Report if approaching capacity if (utilization > 90) {
      logWarning("Stream buffer near capacity", utilization);
    }
  }
}
```

Parameters

in	*stream_*	Handle to the stream buffer to query. Must be a valid stream created with xStreamCreate().
out	*bytes_*	Pointer to variable receiving the byte count. On success, contains the number of bytes currently waiting in the stream (0 to CONFIG_STREAM_BUFFER_BYTES).

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid stream handle or stream not found.

Warning

This function only reports the current state of the stream buffer. In multitasking environments, the byte count may change immediately after the query if other tasks send or receive data. Use appropriate synchronization if you need atomic check-and-receive operations.

Note

xStreamBytesAvailable() is a non-destructive query—it does not modify the stream contents or affect subsequent operations.

The returned byte count includes all bytes sent with xStreamSend() that have not yet been retrieved with xStreamReceive().

To calculate free space in the stream, subtract bytes_ from CONFIG_STREAM_BUFFER_BYTES: freeSpace = CONFIG_STREAM_BUFFER_BYTES - bytes_

See also

xStreamReceive() - Retrieve and remove bytes from stream

xStreamSend() - Send byte to stream

xStreamIsEmpty() - Check if stream has no waiting bytes

xStreamIsFull() - Check if stream is at capacity

xStreamReset() - Clear all bytes from stream

xStreamCreate() - Create a stream buffer

4.2.4.53 xStreamCreate() xReturn xStreamCreate (xStreamBuffer * stream_)

Creates a new stream buffer for byte-by-byte data transfer between tasks. Stream buffers provide a lightweight alternative to message queues when communicating variable-length byte sequences or continuous data streams. Unlike queues which handle discrete messages, streams accept individual bytes that accumulate in a circular buffer until consumed by a receiver.

Stream buffers are ideal for:

- **Serial/UART data**: Buffering incoming bytes from serial ports
- **Protocol parsing**: Accumulating bytes until a complete frame is ready
- **Sensor streaming**: Continuous data collection from sensors
- **Character-based I/O**: Terminal or text-based interfaces
- **Variable-length data**: When message size isn't known in advance

Key characteristics:

- Fixed capacity: CONFIG_STREAM_BUFFER_BYTES (default 32 bytes)
- FIFO ordering: Bytes are retrieved in the order they were sent
- Byte-level granularity: Send and receive one byte at a time
- No message boundaries: Unlike queues, streams don't preserve message structure
- Lightweight: Lower overhead than message queues for byte-oriented data

Stream Lifecycle:

1. Create stream with xStreamCreate() (allocates kernel resources) 2. Producer tasks send bytes with xStreamSend() 3. Consumer tasks receive bytes with xStreamReceive() 4. Optionally query status with xStreamBytesAvailable(), xStreamIsEmpty(), xStreamIsFull() 5. Optionally clear stream with xStreamReset() 6. Delete stream with xStreamDelete() when no longer needed

Example 1: UART receive buffer

```
 xStreamBuffer uartRxStream;
// Create stream during initialization if
(OK(xStreamCreate(&uartRxStream))) {
  // Stream ready for use
}
// ISR or receive task sends bytes as they arrive void uartISR(void) {
  xByte rxByte = UART_DATA_REG;
  xStreamSend(uartRxStream, rxByte);
}
// Processing task receives accumulated data void processUART(xTask task,
xTaskParm parm) {
  xHalfWord count;
  xByte *data;
  if (OK(xStreamReceive(uartRxStream, &count, &data))) {
    if (count > 0) {
      processReceivedData(data, count);
      xMemFree((xAddr)data);  // Always free received data
    }
  }
}
```

Example 2: Protocol parser with framing

```
 xStreamBuffer protocolStream;
void setupProtocol(void) {
  xStreamCreate(&protocolStream);
}
```

```
// Producer accumulates bytes void receiveTask(xTask task, xTaskParm parm)
{
  xByte receivedByte = getByteFromSource();
  xStreamSend(protocolStream, receivedByte);
  // Check if frame delimiter received if (receivedByte ==
FRAME_END_MARKER) {
    xTaskNotifyGive(parserTask);  // Signal parser
  }
}
// Consumer processes complete frames void parserTask(xTask task, xTaskParm
parm) {
  xHalfWord frameSize;
  xByte *frame;
  xTaskWait(1);  // Wait for notification
  if (OK(xStreamReceive(protocolStream, &frameSize, &frame))) {
    parseProtocolFrame(frame, frameSize);
    xMemFree((xAddr)frame);
  }
}
```

Example 3: Sensor data streaming

```
 xStreamBuffer sensorStream;
void initSensors(void) {
  xStreamCreate(&sensorStream);
}
// High-frequency sensor sampling void sampleSensor(xTask task, xTaskParm
parm) {
  xByte sample = readADC();
  xBase isFull;
  if (OK(xStreamIsFull(sensorStream, &isFull)) && isFull) {
    // Stream full - handle overflow xStreamReset(sensorStream);  //
Discard old data
  }
  xStreamSend(sensorStream, sample);
}
// Lower-frequency processing void processSamples(xTask task, xTaskParm
parm) {
  xHalfWord sampleCount;
  xByte *samples;
  if (OK(xStreamReceive(sensorStream, &sampleCount, &samples))) {
    if (sampleCount >= MIN_SAMPLES_FOR_PROCESSING) {
      computeStatistics(samples, sampleCount);
    }
    xMemFree((xAddr)samples);
  }
}
```

Parameters

out	*stream_*	Pointer to xStreamBuffer handle to be initialized. After successful creation, this handle is used in all subsequent stream operations. The handle remains valid until xStreamDelete() is called.

Returns

ReturnOK if stream created successfully, ReturnError if creation failed due to insufficient memory or invalid parameters.

Warning

Stream buffers have a fixed capacity defined by CONFIG_STREAM_BUFFER_BYTES (typically 32 bytes). Sending to a full stream will fail with ReturnError. Use xStreamIsFull() to check capacity before sending critical data.

The stream_ parameter must point to valid memory. Passing null will result in ReturnError.

Note

Stream buffers are allocated from kernel memory and persist until explicitly deleted with xStreamDelete(). Always delete streams when no longer needed to prevent memory leaks.

Unlike message queues, streams do not preserve message boundaries. If you send bytes "ABC" and "DEF" separately, the receiver sees "ABCDEF" as a continuous byte sequence. Implement your own framing if you need to distinguish between separate transmissions.

Streams operate on individual bytes (xByte/Byte_t). To send multi-byte values, send each byte separately or use a message queue instead.

See also

xStreamDelete() - Delete stream and free resources

xStreamSend() - Send byte to stream

xStreamReceive() - Receive all waiting bytes from stream

xStreamBytesAvailable() - Query number of bytes waiting in stream

xStreamReset() - Clear all bytes from stream

xStreamIsEmpty() - Check if stream has no waiting bytes

xStreamIsFull() - Check if stream is at capacity

xQueueCreate() - Alternative for message-oriented communication

4.2.4.54 xStreamDelete() `xReturn xStreamDelete ( const xStreamBuffer stream_ )`

Deletes a stream buffer created by xStreamCreate(), freeing all associated kernel memory and resources. After deletion, the stream handle becomes invalid and must not be used in any subsequent stream operations. Any bytes waiting in the stream at the time of deletion are discarded.

This function should be called when a stream buffer is no longer needed to prevent memory leaks in long-running embedded systems. Proper resource cleanup is essential for system reliability, especially in applications that dynamically create and destroy communication channels.

Deletion scenarios:

- **Application shutdown**: Clean up resources during de-initialization
- **Dynamic reconfiguration**: Remove old streams when changing communication topology
- **Error recovery**: Delete and recreate streams after communication failures
- **Resource management**: Free streams in resource-constrained systems

Example 1: Basic stream cleanup

```
 xStreamBuffer tempStream;
// Create stream for temporary operation if
(OK(xStreamCreate(&tempStream))) {
  // Use stream for data transfer xStreamSend(tempStream, 0x42);
  // ... perform operations ...
  // Clean up when done xStreamDelete(tempStream);
}
```

Example 2: Communication channel lifecycle

```
 xStreamBuffer channelStream = null;
void openChannel(void) {
  if (OK(xStreamCreate(&channelStream))) {
```

```
    // Channel ready for use
  }
}
void closeChannel(void) {
  if (channelStream != null) {
    xStreamDelete(channelStream);
    channelStream = null;  // Prevent use-after-delete
  }
}
```

Example 3: Error recovery with stream recreation

```
 xStreamBuffer dataStream;
void setupDataStream(void) {
  xStreamCreate(&dataStream);
}
void resetCommunication(void) {
  // Delete corrupted stream xStreamDelete(dataStream);
  // Wait briefly for cleanup delayMs(10);
  // Create fresh stream if (OK(xStreamCreate(&dataStream))) {
    // Stream reset successful
  }
}
```

Parameters

in	*stream*↩ _	Handle to the stream buffer to delete. Must be a valid handle obtained from a previous successful call to xStreamCreate(). After deletion, this handle becomes invalid.

Returns

ReturnOK if stream deleted successfully, ReturnError if deletion failed due to invalid handle or stream not found.

Warning

After calling xStreamDelete(), the stream handle becomes invalid and must not be used in any subsequent stream operations. Attempting to use a deleted stream will result in ReturnError.

If other tasks hold references to the stream buffer, ensure they stop using the stream before deletion. Deleting a stream while other tasks are actively sending or receiving data may cause those operations to fail with ReturnError.

Any bytes waiting in the stream buffer at the time of deletion are permanently lost. If you need to preserve data, call xStreamReceive() to retrieve all pending bytes before deleting the stream.

Note

It is good practice to set the stream handle to null after deletion to prevent accidental use of an invalid handle (use-after-delete bugs).

xStreamDelete() only affects the stream buffer itself. Any data previously received with xStreamReceive() and stored in heap memory remains allocated until explicitly freed with xMemFree().

See also

xStreamCreate() - Create a new stream buffer

xStreamReset() - Clear stream contents without deleting the stream

xStreamReceive() - Retrieve pending bytes before deletion

xMemFree() - Free memory allocated by xStreamReceive()

4.2.4.55 xStreamIsEmpty() xReturn xStreamIsEmpty (const xStreamBuffer *stream_*, xBase * *res_*)

Queries whether the specified stream buffer is empty (contains zero bytes). This is a boolean status check that provides a simple, readable way to test for data availability without needing to interpret byte counts. An empty stream has no data waiting to be retrieved.

This function is logically equivalent to checking if xStreamBytesAvailable() returns 0, but offers more expressive code when you only need a boolean empty/not-empty status rather than the exact byte count.

Common use cases:

- **Conditional processing**: Skip receive operations when no data available
- **Stream state verification**: Confirm stream cleared after reset
- **Idle detection**: Determine if communication channel is idle
- **Polling optimization**: Avoid unnecessary memory allocations for empty receives

Example 1: Conditional receive

```
 xStreamBuffer dataStream;
void processTask(xTask task, xTaskParm parm) {
  xBase isEmpty;
  // Check before attempting receive if (OK(xStreamIsEmpty(dataStream,
&isEmpty)) && !isEmpty) {
    // Data available - retrieve and process xHalfWord count;
    xByte *data;
    if (OK(xStreamReceive(dataStream, &count, &data))) {
      processData(data, count);
      xMemFree((xAddr)data);
    }
  }
  // Otherwise skip processing this cycle
}
```

Example 2: Verify reset completion

```
 xStreamBuffer protocolStream;
void resetProtocol(void) {
  xStreamReset(protocolStream);
  // Verify stream actually empty xBase isEmpty;
  if (OK(xStreamIsEmpty(protocolStream, &isEmpty))) {
    if (isEmpty) {
      // Reset confirmed - ready for new data protocolState =
PROTOCOL_IDLE;
    } else {
      // Unexpected - reset failed?
      handleError();
    }
  }
}
```

Example 3: Communication idle detection

```
 xStreamBuffer rxStream;
xTimer idleTimer;
void monitorActivity(xTask task, xTaskParm parm) {
  xBase isEmpty;
  xBase timerExpired;
  if (OK(xStreamIsEmpty(rxStream, &isEmpty)) && isEmpty) {
    // No data waiting - check how long idle if
(OK(xTimerHasTimerExpired(idleTimer, &timerExpired)) && timerExpired) {
      // Idle timeout - enter power-saving mode enterLowPowerMode();
    }
  } else {
    // Activity detected - reset idle timer xTimerReset(idleTimer);
  }
}
```

Parameters

in	*stream_*	Handle to the stream buffer to query. Must be a valid stream created with xStreamCreate().
out	*res_*	Pointer to variable receiving the empty status. Set to non-zero (true) if stream is empty (0 bytes waiting), zero (false) if stream contains data.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid stream handle or stream not found.

Note

This is a non-destructive query that does not modify stream contents.

For more detailed information about stream state, use xStreamBytesAvailable() to get the exact byte count instead of just empty/not-empty status.

See also

xStreamBytesAvailable() - Get exact byte count

xStreamIsFull() - Check if stream is at capacity

xStreamReceive() - Retrieve bytes from stream

xStreamReset() - Clear stream to empty state

xStreamCreate() - Create a stream buffer

4.2.4.56 xStreamIsFull()

```
xReturn xStreamIsFull (
    const xStreamBuffer stream_,
    xBase * res_ )
```

Queries whether the specified stream buffer is full (contains the maximum number of bytes defined by CONFIG_STREAM_BUFFER_BYTES, typically 32). This boolean status check provides a simple way to test for buffer saturation before attempting send operations or to implement overflow handling strategies.

A full stream cannot accept additional bytes via xStreamSend() until space is freed by xStreamReceive() or xStreamReset(). This function is logically equivalent to checking if xStreamBytesAvailable() equals CONFIG_STREAM_BUFFER_BYTES, but offers more expressive code for capacity-related logic.

Common use cases:

- **Pre-send validation**: Check capacity before attempting xStreamSend()
- **Overflow prevention**: Detect full condition to trigger consumer notification
- **Flow control**: Throttle producers when buffer reaches capacity
- **Overflow strategies**: Decide whether to drop data, reset buffer, or block
- **Buffer health monitoring**: Track how often buffer reaches capacity

Example 1: Pre-send capacity check

```
 xStreamBuffer txStream;
xReturn sendByte(xByte data) {
  xBase isFull;
  // Check capacity before sending if (OK(xStreamIsFull(txStream,
&isFull))) {
    if (isFull) {
      // Buffer full - return error or wait return ReturnError;
    }
    // Space available - send byte return xStreamSend(txStream, data);
  }
  return ReturnError;
}
```

Example 2: Overflow handling with notification

```
 xStreamBuffer dataStream;
xTask consumerTask;
void producerTask(xTask task, xTaskParm parm) {
  xByte data = generateData();
  xBase isFull;
  // Check if buffer full if (OK(xStreamIsFull(dataStream, &isFull)) &&
isFull) {
    // Wake up consumer to drain buffer xTaskNotifyGive(consumerTask);
    // Wait briefly for consumer delayMs(5);
  }
  // Attempt send if (ERROR(xStreamSend(dataStream, data))) {
    // Still full - data lost logDataLoss();
  }
}
```

Example 3: Overflow strategy selection

```
 typedef enum {
  OVERFLOW_DROP_OLDEST, OVERFLOW_DROP_NEWEST, OVERFLOW_ERROR
} OverflowStrategy_t;
xStreamBuffer sensorStream;
OverflowStrategy_t strategy = OVERFLOW_DROP_OLDEST;
void recordSample(xByte sample) {
  xBase isFull;
  if (OK(xStreamIsFull(sensorStream, &isFull)) && isFull) {
    switch (strategy) {
      case OVERFLOW_DROP_OLDEST:
        // Clear buffer and add new sample xStreamReset(sensorStream);
        xStreamSend(sensorStream, sample);
        break;
      case OVERFLOW_DROP_NEWEST:
        // Discard new sample, keep old data logDroppedSample();
        break;
      case OVERFLOW_ERROR:
        // Report error condition handleBufferOverflow();
        break;
    }
  } else {
    // Normal send xStreamSend(sensorStream, sample);
  }
}
```

Example 4: Buffer utilization monitoring

```
 xStreamBuffer commStream;
typedef struct {
  xWord fullCount;
  xWord totalChecks;
  xByte peakUtilization;
} BufferStats_t;
BufferStats_t stats = {0, 0, 0};
void monitorBuffer(xTask task, xTaskParm parm) {
  xBase isFull;
  xHalfWord available;
  stats.totalChecks++;
  if (OK(xStreamIsFull(commStream, &isFull)) && isFull) {
    stats.fullCount++;
  }
  // Track peak utilization if (OK(xStreamBytesAvailable(commStream,
&available))) {
    xByte utilization = (xByte)((available * 100) /
CONFIG_STREAM_BUFFER_BYTES);
    if (utilization > stats.peakUtilization) {
      stats.peakUtilization = utilization;
    }
  }
  // Report statistics periodically if (stats.totalChecks % 1000 == 0) {
    xByte fullPercent = (xByte)((stats.fullCount * 100) /
stats.totalChecks);
    reportStats(fullPercent, stats.peakUtilization);
  }
}
```

Parameters

in	*stream_*	Handle to the stream buffer to query. Must be a valid stream created with xStreamCreate().
out	*res_*	Pointer to variable receiving the full status. Set to non-zero (true) if stream is full (CONFIG_STREAM_BUFFER_BYTES bytes waiting), zero (false) if stream has available space.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid stream handle or stream not found.

Warning

A full stream will cause xStreamSend() to fail with ReturnError. Always check for full condition before critical send operations, or implement appropriate error handling for send failures.

Note

This is a non-destructive query that does not modify stream contents.

The buffer capacity is defined by CONFIG_STREAM_BUFFER_BYTES at compile time (default 32 bytes). To calculate free space: freeSpace = CONFIG_STREAM_BUFFER_BYTES - currentBytes

In multitasking environments, the full status may change immediately after the query if another task performs a receive operation. Use appropriate synchronization for atomic check-and-send operations if needed.

See also

xStreamBytesAvailable() - Get exact byte count and calculate free space

xStreamIsEmpty() - Check if stream has no waiting bytes

xStreamSend() - Send byte to stream (fails if full)

xStreamReset() - Clear stream to free all space

xStreamReceive() - Retrieve bytes to free space

xStreamCreate() - Create a stream buffer

4.2.4.57 xStreamReceive()

```
xReturn xStreamReceive (
    const xStreamBuffer stream_,
    xHalfWord * bytes_,
    xByte ** data_ )
```

Retrieves all bytes currently waiting in the specified stream buffer and returns them in a newly allocated memory buffer. This is the consumer-side operation for stream-based inter-task communication. After retrieval, the bytes are removed from the stream, freeing space for new data.

xStreamReceive() allocates memory from the user heap to hold the retrieved bytes. The caller is responsible for freeing this memory with xMemFree() after processing the data. This memory management pattern ensures safe data transfer between tasks without buffer ownership conflicts.

Key characteristics:

- **Retrieves all waiting bytes**: Returns complete contents of stream in one call
- **Allocates memory**: Creates new buffer in user heap for received data
- **Clears stream**: Removes received bytes from stream buffer
- **Returns byte count**: Indicates number of bytes retrieved via bytes_ parameter
- **Non-blocking**: Returns immediately even if stream is empty (bytes_ = 0)
- **FIFO ordering**: Bytes returned in the order they were sent

Typical consumer workflow:

1. Call xStreamReceive() to get all waiting bytes 2. Check if any bytes were retrieved (bytes_ $>$ 0) 3. Process the received data 4. Free the allocated buffer with xMemFree() 5. Repeat as needed for continuous data flow

Example 1: UART receive processing

```
 xStreamBuffer uartRxStream;
// Producer (ISR or task) sends bytes to stream void uartISR(void) {
  if (UART_RX_READY) {
    xByte rxByte = UART_DATA_REG;
    xStreamSend(uartRxStream, rxByte);
  }
}
// Consumer task processes received data void processUARTTask(xTask task,
xTaskParm parm) {
  xHalfWord byteCount;
  xByte *rxData;
  if (OK(xStreamReceive(uartRxStream, &byteCount, &rxData))) {
    if (byteCount > 0) {
      // Process received bytes for (xHalfWord i = 0; i < byteCount; i++) {
        processCharacter(rxData[i]);
      }
      // Always free allocated memory xMemFree((xAddr)rxData);
    }
  }
}
```

Example 2: Protocol frame parsing

```
 xStreamBuffer protocolStream;
void parseProtocolTask(xTask task, xTaskParm parm) {
  xHalfWord frameSize;
  xByte *frame;
  if (OK(xStreamReceive(protocolStream, &frameSize, &frame))) {
    if (frameSize > 0) {
      // Look for start byte (0xAA) for (xHalfWord i = 0; i < frameSize;
i++) {
        if (frame[i] == 0xAA && (i + 3) < frameSize) {
          xByte command = frame[i + 1];
          xByte length = frame[i + 2];
          // Verify complete frame available if ((i + 3 + length) <
frameSize) {
            xByte *payload = &frame[i + 3];
            processCommand(command, payload, length);
            i += 3 + length;  // Skip past this frame
          }
        }
      }
      xMemFree((xAddr)frame);
    }
  }
}
```

Example 3: Sensor data batch processing

```
 xStreamBuffer sensorStream;
#define MIN_SAMPLES 10
void analyzeSensorTask(xTask task, xTaskParm parm) {
  xHalfWord sampleCount;
  xByte *samples;
  if (OK(xStreamReceive(sensorStream, &sampleCount, &samples))) {
    // Only process if we have enough samples if (sampleCount >=
MIN_SAMPLES) {
      // Calculate statistics xWord sum = 0;
      xByte min = 255, max = 0;
      for (xHalfWord i = 0; i < sampleCount; i++) {
        sum += samples[i];
```

```
        if (samples[i] < min) min = samples[i];
        if (samples[i] > max) max = samples[i];
      }
      xByte average = (xByte)(sum / sampleCount);
      reportStatistics(average, min, max);
    } else if (sampleCount > 0) {
      // Put samples back by re-sending them for (xHalfWord i = 0; i <
sampleCount; i++) {
        xStreamSend(sensorStream, samples[i]);
      }
    }
    // Always free even if we didn't process if (sampleCount > 0) {
      xMemFree((xAddr)samples);
    }
  }
}
```

Example 4: Complete producer-consumer pattern

```
 xStreamBuffer dataStream;
xTask producerTask, consumerTask;
// Producer accumulates data void producer(xTask task, xTaskParm parm) {
  xByte dataPoint = collectData();
  if (OK(xStreamSend(dataStream, dataPoint))) {
    // Check if buffer is getting full xHalfWord available;
    if (OK(xStreamBytesAvailable(dataStream, &available))) {
      if (available >= 16) {
        // Signal consumer to process data xTaskNotifyGive(consumerTask);
      }
    }
  }
}
// Consumer processes batches void consumer(xTask task, xTaskParm parm) {
  xTaskWait(1);  // Wait for notification from producer
  xHalfWord count;
  xByte *data;
  if (OK(xStreamReceive(dataStream, &count, &data))) {
    if (count > 0) {
      processBatch(data, count);
      xMemFree((xAddr)data);
    }
  }
}
```

Parameters

in	*stream_*	Handle to the stream buffer to receive from. Must be a valid stream created with xStreamCreate().
out	*bytes_*	Pointer to variable receiving the number of bytes retrieved. Set to 0 if stream is empty. The caller should check this value before processing data_.
out	*data_*	Pointer to variable receiving address of newly allocated buffer containing the retrieved bytes. Memory is allocated from user heap and must be freed with xMemFree() after use. If no bytes are available, this may point to null or uninitialized memory.

Returns

ReturnOK if receive operation completed (even if 0 bytes received), ReturnError if operation failed due to invalid stream handle, memory allocation failure, or stream not found.

Warning

The memory allocated for data_ MUST be freed by the caller using xMemFree() after processing. Failure to free this memory will cause a memory leak. Always pair xStreamReceive() with xMemFree() in your code.

Always check bytes_ before accessing data_. If bytes_ is 0, the stream was empty and data_ should not be accessed.

xStreamReceive() retrieves ALL waiting bytes in a single call. If you need to process bytes one at a time or in smaller chunks, you must implement your own buffering or use the retrieved data array with indexing.

Receiving from an invalid or deleted stream handle will return ReturnError.

Note

xStreamReceive() is non-blocking. If the stream is empty, it returns immediately with bytes_ set to 0. This differs from some RTOS queue implementations that support blocking receives with timeouts.

After xStreamReceive() completes, the retrieved bytes are removed from the stream buffer, freeing space for new data from producers.

Memory allocation failures are rare but possible in heap-constrained systems. If xStreamReceive() returns ReturnError and bytes_ indicates data was available, suspect memory allocation failure and consider increasing heap size.

See also

xStreamSend() - Send bytes to stream (producer operation)

xMemFree() - Free memory allocated by xStreamReceive()

xStreamCreate() - Create a stream buffer

xStreamBytesAvailable() - Query number of bytes waiting without receiving

xStreamIsEmpty() - Check if stream has any waiting bytes

xStreamReset() - Clear stream without retrieving bytes

xQueueReceive() - Alternative for message-oriented communication

4.2.4.58 xStreamReset() `xReturn xStreamReset ( const xStreamBuffer stream_ )`

Resets the specified stream buffer to an empty state by discarding all waiting bytes. After reset, the stream behaves as if newly created— xStreamBytesAvailable() returns 0 and xStreamIsEmpty() returns true. This operation is useful for error recovery, protocol resyncs, or discarding stale data.

Unlike xStreamDelete() which destroys the stream entirely, xStreamReset() preserves the stream structure and handle while only clearing its contents. The stream remains fully functional and can immediately accept new data via xStreamSend().

Common scenarios for stream reset:

- **Error recovery**: Clear corrupted or incomplete protocol data
- **Protocol resync**: Discard partial frames when reestablishing sync
- **Buffer overflow handling**: Clear old data when capacity exceeded
- **Timeout handling**: Discard stale data after communication timeout
- **State machine reset**: Clear buffered data when returning to idle state
- **Channel cleanup**: Prepare stream for reuse without deallocating

Example 1: Protocol error recovery

```
 xStreamBuffer protocolStream;
void handleProtocolError(void) {
  // Error detected - discard partial/corrupted data
xStreamReset(protocolStream);
  // Send resync request to peer sendResyncCommand();
  // Stream now empty and ready for fresh data
}
```

Example 2: Timeout-based data discard

```
 xStreamBuffer rxStream;
xTimer rxTimeout;
void receiveTask(xTask task, xTaskParm parm) {
  xBase timerExpired;
  // Check for receive timeout if (OK(xTimerHasTimerExpired(rxTimeout,
&timerExpired)) && timerExpired) {
    xHalfWord staleBytes;
    // Check if incomplete data waiting if
(OK(xStreamBytesAvailable(rxStream, &staleBytes)) && staleBytes > 0) {
      // Data incomplete after timeout - discard it xStreamReset(rxStream);
      logWarning("Receive timeout - data discarded");
    }
    // Reset timeout for next receive xTimerReset(rxTimeout);
  }
}
```

Example 3: Overflow handling with reset

```
 xStreamBuffer sensorStream;
void sampleSensor(xTask task, xTaskParm parm) {
  xByte sample = readADC();
  xBase isFull;
  // Check if buffer full if (OK(xStreamIsFull(sensorStream, &isFull)) &&
isFull) {
    // Option 1: Drop oldest data, keep newest xStreamReset(sensorStream);
    xStreamSend(sensorStream, sample);
    // Log overflow event overflowCount++;
  } else {
    // Normal send xStreamSend(sensorStream, sample);
  }
}
```

Example 4: State machine with stream cleanup

```
 typedef enum {
  STATE_IDLE, STATE_RECEIVING, STATE_PROCESSING
} CommState_t;
xStreamBuffer commStream;
CommState_t commState = STATE_IDLE;
void communicationTask(xTask task, xTaskParm parm) {
  switch (commState) {
    case STATE_IDLE:
      // Ensure stream clean when entering new transaction
xStreamReset(commStream);
      commState = STATE_RECEIVING;
      break;
    case STATE_RECEIVING:
      // Accumulate data...
      if (frameComplete()) {
        commState = STATE_PROCESSING;
      }
      break;
    case STATE_PROCESSING:
      // Process received frame processFrame();
      commState = STATE_IDLE;  // Will clear on next iteration break;
  }
}
```

Parameters

in	*stream*↩ _	Handle to the stream buffer to reset. Must be a valid stream created with xStreamCreate().

Returns

ReturnOK if stream reset successfully, ReturnError if reset failed due to invalid stream handle or stream not found.

Warning

All bytes waiting in the stream are permanently discarded. If you need to preserve data, call xStreamReceive() before calling xStreamReset().

In multitasking environments, ensure no other task is actively sending to or receiving from the stream during reset. Resetting a stream while another task is mid-operation may lead to unexpected behavior or data loss.

Note

xStreamReset() does not delete or invalidate the stream handle. After reset, the stream remains fully functional and ready for new send/receive operations.

This operation is typically faster than deleting and recreating a stream, making it preferable for error recovery scenarios where the stream structure should be reused.

After reset, xStreamBytesAvailable() returns 0, xStreamIsEmpty() returns true, and xStreamIsFull() returns false.

See also

xStreamCreate() - Create a stream buffer

xStreamDelete() - Delete stream entirely (vs. just clearing contents)

xStreamReceive() - Retrieve data before reset if needed

xStreamBytesAvailable() - Check byte count before reset

xStreamIsEmpty() - Verify stream empty after reset

4.2.4.59 xStreamSend()

```
xReturn xStreamSend (
    xStreamBuffer stream_,
    const xByte byte_ )
```

Sends one byte to the specified stream buffer, adding it to the end of the FIFO byte sequence. This is the producer-side operation for stream-based inter-task communication. Bytes are stored in the stream until consumed by a receiver task using xStreamReceive().

Stream buffers provide byte-level granularity for communication, making them ideal for character-oriented protocols, serial data buffering, and continuous data streaming. Unlike message queues, streams don't preserve message boundaries— bytes are retrieved in the exact order they were sent, but without any structure or framing.

Operational characteristics:

- **FIFO ordering**: Bytes are retrieved in the order they were sent
- **Fixed capacity**: Stream holds up to CONFIG_STREAM_BUFFER_BYTES (default 32)
- **Blocking behavior**: Returns ReturnError immediately if stream is full
- **Atomic operation**: Single byte send is atomic (safe from interrupts)
- **Low overhead**: Minimal processing compared to message queue operations

Common usage patterns:

- **Serial/UART buffering**: Send received bytes from ISR to stream for processing
- **Character output**: Build strings character-by-character for display
- **Protocol assembly**: Accumulate protocol bytes until complete frame ready
- **Sensor sampling**: Stream continuous ADC or sensor readings
- **Event logging**: Record byte-oriented event codes or timestamps

Example 1: UART transmit buffering

```
 xStreamBuffer uartTxStream;
void setupUART(void) {
  xStreamCreate(&uartTxStream);
}
// Application sends string to UART void sendString(const char *str) {
  while (*str) {
    if (ERROR(xStreamSend(uartTxStream, (xByte)*str))) {
      // Stream full - wait for transmitter to drain delayMs(1);
    } else {
      str++;
    }
  }
}
// UART task transmits buffered bytes void uartTransmitTask(xTask task,
xTaskParm parm) {
  xHalfWord count;
  xByte *data;
  if (OK(xStreamReceive(uartTxStream, &count, &data))) {
    if (count > 0) {
      uartWriteBytes(data, count);
      xMemFree((xAddr)data);
    }
  }
}
```

Example 2: Protocol frame assembly

```
 xStreamBuffer protocolStream;
// Send protocol header, payload, and checksum xReturn
sendProtocolFrame(xByte command, xByte *payload, xHalfWord len) {
  // Send start byte if (ERROR(xStreamSend(protocolStream, 0xAA))) {
    return ReturnError;
  }
  // Send command if (ERROR(xStreamSend(protocolStream, command))) {
    return ReturnError;
  }
  // Send length if (ERROR(xStreamSend(protocolStream, (xByte)len))) {
    return ReturnError;
  }
  // Send payload bytes for (xHalfWord i = 0; i < len; i++) {
    if (ERROR(xStreamSend(protocolStream, payload[i]))) {
      return ReturnError;
    }
  }
  // Send checksum xByte checksum = calculateChecksum(command, payload,
len);
  return xStreamSend(protocolStream, checksum);
}
```

Example 3: Sensor data streaming with overflow handling

```
 xStreamBuffer sensorStream;
void sampleSensorTask(xTask task, xTaskParm parm) {
  xByte sample = readADC();
  // Try to send sample if (ERROR(xStreamSend(sensorStream, sample))) {
    // Stream full - check how to handle overflow xBase isFull;
    if (OK(xStreamIsFull(sensorStream, &isFull)) && isFull) {
      // Option 1: Drop oldest data and reset xStreamReset(sensorStream);
      xStreamSend(sensorStream, sample);
      // Option 2: Notify error handler
      // logOverflowError();
    }
  }
}
```

Parameters

in	*stream_*	Handle to the stream buffer to send data to. Must be a valid stream created with xStreamCreate().
in	*byte_*	The byte value to send to the stream. Accepts any value from 0x00 to 0xFF.

Returns

ReturnOK if byte sent successfully, ReturnError if send failed due to stream full, invalid stream handle, or stream not found.

Warning

If the stream is full (contains CONFIG_STREAM_BUFFER_BYTES bytes), xStreamSend() returns ReturnError immediately without blocking. Check stream capacity with xStreamIsFull() or xStreamBytesAvailable() before sending critical data, or implement error handling for full conditions.

Sending to an invalid or deleted stream handle will return ReturnError. Always verify stream creation succeeded before calling xStreamSend().

Note

xStreamSend() operates on individual bytes only. To send multi-byte values (integers, floats, structures), send each byte separately in the appropriate order (considering endianness if necessary).

Stream buffers do not preserve message boundaries. Bytes sent in separate xStreamSend() calls are concatenated into a continuous sequence. If you need message framing, implement your own protocol with headers/delimiters.

For high-throughput applications, consider checking xStreamIsFull() before sending large amounts of data to avoid repeated ReturnError conditions.

See also

xStreamReceive() - Receive all waiting bytes from stream (consumer operation)

xStreamCreate() - Create a stream buffer

xStreamBytesAvailable() - Query number of bytes waiting in stream

xStreamIsFull() - Check if stream is at capacity

xStreamReset() - Clear all bytes from stream

xQueueSend() - Alternative for message-oriented communication

4.2.4.60 xSystemAssert()

```
xReturn xSystemAssert (
    const char * file_,
    const int line_ )
```

The xSystemAssert() syscall is used to raise a system assert. In order fot xSystemAssert() to have an effect the configuration setting CONFIG_SYSTEM_ASSERT_BEHAVIOR must be defined. That said, it is recommended that the **AssertOnElse**() C macro be used in place of xSystemAssert(). In order for the **AssertOnElse**() C macro to have any effect, the configuration setting CONFIG_ENABLE_SYSTEM_ASSERT must be defined.

See also

xReturn

CONFIG_SYSTEM_ASSERT_BEHAVIOR

CONFIG_ENABLE_SYSTEM_ASSERT

AssertOnElse()

Parameters

*file_*	The C file where the assert occurred. This will be set by the **AssertOnElse**() C macro.
*line_*	The C file line where the assert occurred. This will be set by the **AssertOnElse**() C macro.

Returns

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))) {}).

4.2.4.61 xSystemGetSystemInfo()

```
xReturn xSystemGetSystemInfo (
    xSystemInfo * info_ )
```

The xSystemGetSystemInfo() syscall is used to inquire about the system. The information bout the system that may be obtained is the product (i.e., OS) name, version and number of tasks.

See also

xReturn

xSystemInfo

xMemFree()

Parameters

*info_*	The system information. The system information must be freed by xMemFree().

Returns

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))) {}).

4.2.4.62 xSystemHalt()

```
xReturn xSystemHalt (
    void  )
```

The xSystemHalt() syscall is used to halt HeliOS. Once called, xSystemHalt() will disable all interrupts and stops the execution of further statements. The system will have to be reset to recover.

See also

xReturn

Returns

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))) {}).

4.2.4.63 xSystemInit()

```
xReturn xSystemInit (
    void  )
```

The xSystemInit() syscall is used to bootstrap HeliOS and must be the first syscall made in the user's application. The xSystemInit() syscall initializes memory and calls initialization functions through the port layer.

See also

xReturn

Returns

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))) {}).

4.2.4.64 xTaskChangePeriod()

```
xReturn xTaskChangePeriod (
    xTask task_,
    const xTicks period_ )
```

The xTaskChangePeriod() is used to change the interval period of a task timer. The period is measured in ticks. While architecture and/or platform dependent, a tick is often one millisecond. In order for the task timer to have an effect, the task must be in the "waiting" state which can be set using xTaskWait().

See also

xReturn

xTask

xTicks

xTaskWait()

Parameters

*task_*	The task to be operated on.
*period_*	The interval period in ticks.

Returns

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))) {}).

4.2.4.65 xTaskChangeWDPeriod()

```
xReturn xTaskChangeWDPeriod (
    xTask task_,
    const xTicks period_ )
```

The xTaskChangeWDPeriod() syscall is used to change the task watchdog timer period. This has no effect unless CONFIG_TASK_WD_TIMER_ENABLE is defined and the watchdog timer period is greater than nil. The task watchdog timer will place a task in a suspended state if a task's runtime exceeds the watchdog timer period. The task watchdog timer period is set on a per task basis.

See also

xReturn

xTask

xTicks

CONFIG_TASK_WD_TIMER_ENABLE

Parameters

*task_*	The task to be operated on.
*period_*	The task watchdog timer period measured in ticks. Ticks is platform and/or architecture dependent. However, most platforms and/or architectures have a one millisecond tick duration.

Returns

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))) {}).

4.2.4.66 xTaskCreate()

```
xReturn xTaskCreate (
        xTask * task_,
        const xByte * name_,
        void(*)(xTask task_, xTaskParm parm_) callback_,
        xTaskParm taskParameter_ )
```

Creates a new task and registers it with the HeliOS scheduler. Tasks are the fundamental unit of execution in HeliOS and represent functions that execute cooperatively under scheduler control. Each task has a name, callback function, optional parameters, and scheduling properties.

When created, tasks begin in TaskStateSuspended state and will not execute until xTaskResume() is called. Tasks can have an associated period that controls how frequently they are scheduled for execution.

Task Lifecycle:

1. Create task with xTaskCreate() (starts in TaskStateSuspended) 2. Optionally set task period with xTaskChangePeriod() 3. Resume task with xTaskResume() (transitions to TaskStateRunning) 4. Task executes when scheduler runs 5. Delete task with xTaskDelete() when no longer needed

Warning

This function must be called BEFORE xTaskStartScheduler() is invoked. It cannot be called from within a task while the scheduler is running. Creating or deleting tasks during scheduler execution will result in undefined behavior.

Note

Task names are used for identification and debugging. They must be exactly CONFIG_TASK_NAME_BYTES characters (default 8). Shorter names should be space-padded; longer names will be truncated.

The taskParameter_ allows passing data to the task. This is commonly used to pass pointers to configuration structures or shared data. The memory management of this parameter is the responsibility of the caller.

Example Usage:

```
 void blinkLED(xTask task, xTaskParm parm) {
  // Access parameter if needed int *ledPin = (int*)parm;
  // Task logic here toggleLED(*ledPin);
  // Task yields when function returns
}
int main(void) {
  xTask ledTask;
  static int pin = 13;  // Must persist beyond task creation
  // Create task (starts in suspended state) if (OK(xTaskCreate(&ledTask,
"LED     ", blinkLED, (xTaskParm)&pin))) {
    // Set task to run every 100ms xTaskChangePeriod(ledTask, 100);
    // Activate the task xTaskResume(ledTask);
    // Start scheduler xTaskStartScheduler();
  }
}
```

Parameters

out	*task_*	Pointer to xTask variable that will receive the task handle. This handle is used in subsequent operations on the task.
in	*name_*	Task name string, exactly CONFIG_TASK_NAME_BYTES bytes. Common pattern: "TaskName" (8 bytes with space padding).
in	*callback_*	Pointer to the task function that will be executed. Function signature: void func(xTask task, xTaskParm parm).
in	*taskParameter_*	Optional parameter passed to the task function. Use NULL if no parameter is needed. Can be a pointer to any data type.

Returns

ReturnOK if task was successfully created, ReturnError if creation failed (e.g., out of memory, invalid parameters, or scheduler is already running).

See also

xTaskDelete() - Remove a task from the scheduler

xTaskResume() - Activate a task for scheduling

xTaskSuspend() - Deactivate a task

xTaskChangePeriod() - Set task execution frequency

xTaskStartScheduler() - Begin executing tasks

CONFIG_TASK_NAME_BYTES - Configuration for task name length

4.2.4.67 xTaskDelete()

```
xReturn xTaskDelete (
        const xTask task_ )
```

Removes a task from the HeliOS scheduler and frees all memory associated with the task. Once deleted, the task will no longer be scheduled for execution and its task handle becomes invalid.

This function performs cleanup including:

- Removing the task from the scheduler's task list
- Freeing internal task control structures
- Invalidating the task handle

Warning

This function must be called BEFORE xTaskStartScheduler() starts or AFTER the scheduler has been suspended with xTaskSuspendAll(). It cannot be called from within a running task while the scheduler is active. Attempting to delete a task during scheduler execution will result in undefined behavior.

After deleting a task, the task handle becomes invalid and must not be used in any subsequent HeliOS function calls. Using a deleted task handle will result in undefined behavior.

Note

If a task needs to be temporarily deactivated rather than permanently removed, use xTaskSuspend() instead. Suspended tasks can be resumed later without recreating them.

Memory allocated by the task (via xMemAlloc()) is NOT automatically freed when the task is deleted. The application is responsible for managing any dynamically allocated memory used by the task.

Example Usage:

```
 xTask temporaryTask;
// Create a task for one-time initialization if
(OK(xTaskCreate(&temporaryTask, "InitTask", initFunction, NULL))) {
  xTaskResume(temporaryTask);
  xTaskStartScheduler();
  // Later, after initialization is complete (scheduler suspended)
xTaskSuspendAll();
  // Delete the task as it's no longer needed if
(OK(xTaskDelete(temporaryTask))) {
    // Task successfully deleted
  }
  xTaskResumeAll();
  xTaskStartScheduler();
}
```

Parameters

in	*task_*	Handle of the task to delete. Must be a valid task handle previously returned by xTaskCreate().

Returns

ReturnOK if the task was successfully deleted, ReturnError if deletion failed (invalid task handle, scheduler is running, or system error).

See also

xTaskCreate() - Create a new task

xTaskSuspend() - Temporarily deactivate a task (without deleting it)

xTaskSuspendAll() - Suspend scheduler to allow task deletion

xTaskStartScheduler() - Start the scheduler

4.2.4.68 xTaskGetAllRunTimeStats()

```
xReturn xTaskGetAllRunTimeStats (
    xTaskRunTimeStats * stats_,
    xBase * tasks_ )
```

The xTaskGetAllRunTimeStats() syscall is used to obtain the runtime statistics of all tasks.

See also

xReturn

xTask

xTaskRunTimeStats

xMemFree()

Parameters

*stats_*	The runtime statistics. The runtime statics must be freed by xMemFree().
*tasks_*	The number of tasks in the runtime statistics.

Returns

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))) {}).

4.2.4.69 xTaskGetAllTaskInfo() `xReturn xTaskGetAllTaskInfo (`

```
    xTaskInfo * info_,
    xBase * tasks_ )
```

The xTaskGetAllTaskInfo() syscall is used to get info about all tasks. xTaskGetAllTaskInfo() is similar to xTaskGetAllRunTimeStats() with one difference, xTaskGetAllTaskInfo() provides the state and name of the task along with the task's runtime statistics.

See also

xReturn

xTaskInfo

xMemFree()

Parameters

*info_*	Information about the tasks. The task information must be freed by xMemFree().
*tasks_*	The number of tasks.

Returns

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))) {}).

4.2.4.70 xTaskGetHandleById() `xReturn xTaskGetHandleById (`

```
    xTask * task_,
    const xBase id_ )
```

The xTaskGetHandleById() syscall will get the task handle using the task id.

See also

xReturn

xTask

Parameters

*task_*	The task to be operated on.
*id_*	The task id.

Returns

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))) {}).

4.2.4.71 xTaskGetHandleByName()

```
xReturn xTaskGetHandleByName (
    xTask * task_,
    const xByte * name_ )
```

The xTaskGetHandleByName() syscall will get the task handle using the task name.

See also

xReturn

xTask

CONFIG_TASK_NAME_BYTES

Parameters

*task_*	The task to be operated on.
*name_*	The name of the task which must be exactly CONFIG_TASK_NAME_BYTES (default is 8) bytes in length. Shorter task names must be padded.

Returns

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))) {}).

4.2.4.72 xTaskGetId()

```
xReturn xTaskGetId (
    const xTask task_,
    xBase * id_ )
```

The xTaskGetId() syscall is used to obtain the id of a task.

See also

xReturn

xTask

Parameters

*task_*	The task to be operated on.
*id_*	The id of the task.

Returns

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))) {}).

4.2.4.73 xTaskGetName()

```
xReturn xTaskGetName (
    const xTask task_,
    xByte ** name_ )
```

The xTaskGetName() syscall is used to get the ASCII name of a task. The size of the task name is CONFIG_TASK_NAME_BYTES (default is 8) bytes in length.

See also

xReturn

xTask

xMemFree()

Parameters

*task_*	The task to be operated on.
*name_*	The task name which must be precisely CONFIG_TASK_NAME_BYTES (default is 8) bytes in length. The task name must be freed by xMemFree().

Returns

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))) {}).

4.2.4.74 xTaskGetNumberOfTasks() xReturn xTaskGetNumberOfTasks (xBase * tasks_)

The xTaskGetNumberOfTasks() syscall is used to obtain the number of tasks regardless of their state (i.e., suspended, running or waiting).

See also

xReturn

Parameters

*tasks_*	The number of tasks.

Returns

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))) {}).

4.2.4.75 xTaskGetPeriod() xReturn xTaskGetPeriod (const xTask task_, xTicks * period_)

The xTaskGetPeriod() syscall is used to obtain the current task timer period.

See also

xReturn

xTask

xTicks

Parameters

*task_*	The task to be operated on.
*period_*	The task timer period in ticks. Ticks is platform and/or architecture dependent. However, most platforms and/or architect

Returns

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))) {}).

4.2.4.76 xTaskGetSchedulerState()

```
xReturn xTaskGetSchedulerState (
    xSchedulerState * state_ )
```

The xTaskGetSchedulerState() is used to get the state of the scheduler.

See also

xReturn

xSchedulerState

Parameters

*state_*	The state of the scheduler.

Returns

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))) {}).

4.2.4.77 xTaskGetTaskInfo()

```
xReturn xTaskGetTaskInfo (
    const xTask task_,
    xTaskInfo * info_ )
```

The xTaskGetTaskInfo() syscall is used to get info about a single task. xTaskGetTaskInfo() is similar to xTaskGetTaskRunTimeStats() with one difference, xTaskGetTaskInfo() provides the state and name of the task along with the task's runtime statistics.

See also

xReturn

xMemFree()

xTask

xTaskInfo

Parameters

*task_*	The task to be operated on.
*info_*	Information about the task. The task information must be freed by xMemFree().

Returns

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))) {}).

4.2.4.78 xTaskGetTaskRunTimeStats()

```
xReturn xTaskGetTaskRunTimeStats (
    const xTask task_,
    xTaskRunTimeStats * stats_ )
```

The xTaskGetTaskRunTimeStats() syscall is used to get the runtime statistics for a single task.

See also

xReturn

xTask

xTaskRunTimeStats

xMemFree()

Parameters

*task_*	The task to be operated on.
*stats_*	The runtime statistics. The runtime statistics must be freed by xMemFree().

Returns

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))) {}).

4.2.4.79 xTaskGetTaskState()

```
xReturn xTaskGetTaskState (
    const xTask task_,
    xTaskState * state_ )
```

The xTaskGetTaskState() syscall is used to obtain the state of a task (i.e., suspended, running or waiting).

See also

xReturn

xTask

xTaskState

Parameters

*task_*	The task to be operated on.
*state_*	The state of the task.

Returns

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))) {}).

4.2.4.80 xTaskGetWDPeriod()

```
xReturn xTaskGetWDPeriod (
    const xTask task_,
    xTicks * period_ )
```

The xTaskGetWDPeriod() syscall is used to obtain the task watchdog timer period.

See also

xReturn

xTask

xTicks

CONFIG_TASK_WD_TIMER_ENABLE

Parameters

*task_*	The task to be operated on.
*period_*	The task watchdog timer period, measured in ticks. Ticks are platform and/or architecture dependent. However, on must platforms and/or architectures the tick represents one millisecond.

Returns

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))) {}).

4.2.4.81 xTaskNotificationIsWaiting()

```
xReturn xTaskNotificationIsWaiting (
    const xTask task_,
    xBase * res_ )
```

The xTaskNotificationIsWaiting() syscall is used to inquire as to whether a direct-to-task notification is waiting for the given task.

See also

xReturn

xTask

Parameters

*task_*	Task to be operated on.
*res_*	The result of the inquiry; taken here to mean "true" if there is a waiting direct-to-task notification. Otherwise "false", if there is not a waiting direct-to-notification.

Returns

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))) {}).

4.2.4.82 xTaskNotifyGive()

```
xReturn xTaskNotifyGive (
    xTask task_,
    const xBase bytes_,
    const xByte * value_ )
```

The xTaskNotifyGive() syscall is used to give (i.e., send) a direct-to-task notification to the given task.

See also

xReturn

xTask

CONFIG_NOTIFICATION_VALUE_BYTES

Parameters

*task_*	The task to be operated on.
*bytes_*	The number of bytes contained in the notification value. The number of bytes in the notification value cannot exceed CONFIG_NOTIFICATION_VALUE_BYTES (default is 8) bytes.
*value_*	The notification value which is a byte array whose length is defined by "bytes_".

Returns

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))) {}).

4.2.4.83 xTaskNotifyStateClear()

```
xReturn xTaskNotifyStateClear (
        xTask task_ )
```

The xTaskNotifyStateClear() syscall is used to clear a waiting direct-to-task notification for the given task.

See also

xReturn

xTask

Parameters

*task_*	The task to be operated on.

Returns

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))) {}).

4.2.4.84 xTaskNotifyTake()

```
xReturn xTaskNotifyTake (
        xTask task_,
        xTaskNotification * notification_ )
```

The xTaskNotifyTake() syscall is used to take (i.e., receive) a waiting direct-to-task notification.

See also

xReturn

xTask

CONFIG_NOTIFICATION_VALUE_BYTES

xTaskNotification

Parameters

*task_*	The task to be operated on.
*notification_*	The direct-to-task notification.

Returns

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))) {}).

4.2.4.85 xTaskResetTimer()

```
xReturn xTaskResetTimer (
        xTask task_ )
```

The xTaskResetTimer() syscall is used to reset the task timer. In effect, this sets the elapsed time, measured in ticks, back to nil.

See also

xReturn

xTask

xTicks

Parameters

*task_*	The task to be operated on.

Returns

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))) {}).

4.2.4.86 xTaskResume() xReturn xTaskResume (xTask *task_*)

Transitions a task to the TaskStateRunning state, making it eligible for execution by the cooperative scheduler. Tasks in the running state are scheduled according to their configured period and will execute continuously until suspended, deleted, or transitioned to waiting state.

When a task is created with xTaskCreate(), it begins in TaskStateSuspended and will not execute until this function is called. This allows tasks to be fully configured (period, parameters, etc.) before activation.

State Transition:

- TaskStateSuspended → TaskStateRunning: Task becomes active
- TaskStateWaiting → TaskStateRunning: Task resumes normal scheduling
- TaskStateRunning → TaskStateRunning: No effect (already running)

Scheduling Behavior: Once resumed, the task will be scheduled for execution based on its period setting. Tasks with period 0 execute on every scheduler cycle. Tasks with non-zero periods execute when their elapsed time exceeds the period.

Note

This function can be called before or during scheduler execution. It safely transitions the task state and the change takes effect on the next scheduling cycle.

A task in the running state will continue executing until explicitly suspended with xTaskSuspend(), transitioned to waiting with xTaskWait(), or deleted with xTaskDelete().

Example Usage:

```
 xTask sensorTask;
xTask displayTask;
// Create tasks (both start in suspended state) xTaskCreate(&sensorTask,
"Sensor  ", readSensor, NULL);
xTaskCreate(&displayTask, "Display ", updateDisplay, NULL);
// Configure task periods xTaskChangePeriod(sensorTask, 100);   // Run
every 100ms xTaskChangePeriod(displayTask, 500);  // Run every 500ms
// Activate tasks xTaskResume(sensorTask);   // Now scheduled for execution
xTaskResume(displayTask);  // Now scheduled for execution
// Start scheduler - both tasks will now execute xTaskStartScheduler();
// Can also resume tasks while scheduler is running void controlTask(xTask
task, xTaskParm parm) {
  if (systemReady) {
    xTaskResume(sensorTask);  // Activate sensor readings
  }
}
```

Parameters

in	*task_*	Handle of the task to resume. Must be a valid task handle previously returned by xTaskCreate().

Returns

ReturnOK if the task state was successfully changed to running, ReturnError if the operation failed (invalid task handle or system error).

See also

xTaskCreate() - Create a new task (starts in suspended state)

xTaskSuspend() - Deactivate a task

xTaskWait() - Place a task in event-waiting state

xTaskGetTaskState() - Query current task state

xTaskChangePeriod() - Set task execution frequency

TaskState_t - Task state enumeration

4.2.4.87 xTaskResumeAll() `xReturn xTaskResumeAll ( void )`

Sets the scheduler state to SchedulerStateRunning, allowing the scheduler to execute tasks when xTaskStartScheduler() is called. This function does not start the scheduler itself; it only changes the scheduler's state flag.

This function is used in two scenarios:

1. After calling xTaskSuspendAll() to re-enable scheduling 2. During system initialization to set the scheduler to running state before calling xTaskStartScheduler()

Typical Usage Pattern:

- Call xTaskSuspendAll() to stop scheduler and regain control
- Perform critical operations while scheduler is suspended
- Call xTaskResumeAll() to mark scheduler as ready
- Call xTaskStartScheduler() to resume task execution

Note

This function only changes the scheduler state flag. You must still call xTaskStartScheduler() to actually begin executing tasks.

If xTaskStartScheduler() is called while the scheduler state is SchedulerStateSuspended, it will return immediately without executing any tasks.

Example Usage:

```
// Suspend scheduler from within a task xTaskSuspendAll();
// Perform time-sensitive operations performCriticalOperation();
// Resume scheduler state xTaskResumeAll();
// Restart scheduler xTaskStartScheduler();
```

Returns

ReturnOK on success, ReturnError on failure.

See also

xTaskStartScheduler() - Start executing tasks

xTaskSuspendAll() - Suspend scheduler and return control

xTaskGetSchedulerState() - Query current scheduler state

SchedulerState_t - Scheduler state enumeration

4.2.4.88 xTaskStartScheduler() `xReturn xTaskStartScheduler ( void )`

Transfers control from application code to the HeliOS scheduler, which begins executing tasks in a cooperative multitasking fashion. This is typically the last function called in main() after all tasks, timers, and other objects have been created and configured.

The scheduler operates in a continuous loop, executing tasks based on their state and elapsed time since last execution. Tasks in TaskStateRunning or TaskStateWaiting (with pending events) will be scheduled according to their configured period.

Control Flow:

- If the scheduler state is SchedulerStateRunning, execution enters the scheduler loop and does not return unless xTaskSuspendAll() is called from within a running task
- If the scheduler state is SchedulerStateSuspended, this function returns immediately without scheduling any tasks
- To resume after suspension, call xTaskResumeAll() followed by xTaskStartScheduler() again

Note

HeliOS uses cooperative multitasking. Tasks must explicitly yield control by returning from their task function to allow other tasks to execute.

Warning

Do not call this function multiple times without first suspending the scheduler using xTaskSuspendAll() and resuming it with xTaskResumeAll().

Example Usage:

```
int main(void) {
  xTask myTask;
  // Create tasks xTaskCreate(&myTask, "Task1", myTaskFunction, NULL);
  xTaskResume(myTask);
  // Start scheduler (does not return) xTaskStartScheduler();
  return 0;
}
```

Returns

ReturnOK if the scheduler successfully starts or is already suspended. ReturnError if a critical error occurs preventing scheduler operation. Note that under normal operation with tasks running, this function does not return until xTaskSuspendAll() is called.

See also

xTaskResumeAll() - Resume scheduler after suspension

xTaskSuspendAll() - Suspend scheduler and return control

xTaskGetSchedulerState() - Query current scheduler state

xTaskCreate() - Create a new task

SchedulerState_t - Scheduler state enumeration

4.2.4.89 xTaskSuspend() xReturn xTaskSuspend (xTask *task_*)

Transitions a task to the TaskStateSuspended state, preventing it from being scheduled for execution. The task remains registered with the scheduler and retains all its configuration (period, parameters, etc.), but will not execute until reactivated with xTaskResume().

Suspending a task is useful for temporarily disabling functionality without the overhead of deleting and recreating the task. Unlike xTaskWait(), which waits for specific events, suspended tasks remain inactive indefinitely until explicitly resumed.

State Transition:

- TaskStateRunning → TaskStateSuspended: Task becomes inactive
- TaskStateWaiting → TaskStateSuspended: Task becomes inactive
- TaskStateSuspended → TaskStateSuspended: No effect (already suspended)

Use Cases:

- Temporarily disable a task based on system state or mode
- Reduce CPU usage by deactivating unused functionality
- Disable tasks during power-saving modes
- Pause task execution during system reconfiguration

Note

This function can be called before or during scheduler execution. When called on an active task, the task will not be scheduled on subsequent scheduler cycles until resumed.

Suspending a task does NOT free its resources or invalidate its handle. The task can be resumed at any time with xTaskResume(). To permanently remove a task, use xTaskDelete() instead.

This function (xTaskSuspend) operates on individual tasks. For suspending the entire scheduler, use xTaskSuspendAll() instead.

Example Usage:

```
 xTask bluetoothTask;
xTask wifiTask;
xTaskCreate(&bluetoothTask, "BT Task", bluetoothHandler, NULL);
xTaskCreate(&wifiTask, "WiFi    ", wifiHandler, NULL);
xTaskResume(bluetoothTask);
xTaskResume(wifiTask);
xTaskStartScheduler();
// In a control task, disable wireless when not needed void
powerManagementTask(xTask task, xTaskParm parm) {
  if (lowPowerMode) {
    // Suspend wireless tasks to save power xTaskSuspend(bluetoothTask);
    xTaskSuspend(wifiTask);
  } else {
    // Resume wireless tasks when needed xTaskResume(bluetoothTask);
    xTaskResume(wifiTask);
  }
}
```

Parameters

in	*task_*	Handle of the task to suspend. Must be a valid task handle previously returned by xTaskCreate().

Returns

ReturnOK if the task state was successfully changed to suspended, ReturnError if the operation failed (invalid task handle or system error).

See also

xTaskCreate() - Create a new task

xTaskResume() - Reactivate a suspended task

xTaskWait() - Place a task in event-waiting state

xTaskDelete() - Permanently remove a task

xTaskSuspendAll() - Suspend the entire scheduler

xTaskGetTaskState() - Query current task state

TaskState_t - Task state enumeration

4.2.4.90 xTaskSuspendAll() `xReturn xTaskSuspendAll ( void )`

Sets the scheduler state to SchedulerStateSuspended and causes xTaskStartScheduler() to exit its scheduling loop, returning control back to the point where xTaskStartScheduler() was called. This provides a mechanism to temporarily halt cooperative multitasking and regain direct control of program execution.

When called from within a running task, this function causes the scheduler loop to terminate after the current task completes its execution. Control returns to the line immediately following the original xTaskStartScheduler() call.

Use Cases:

- Temporarily halt all task execution for critical operations
- Enter a low-power mode that requires stopping the scheduler
- Transition to a different operating mode
- Debug or diagnostic operations that require full control

To resume task execution:

1. Call xTaskResumeAll() to set scheduler state to running 2. Call xTaskStartScheduler() to re-enter the scheduling loop

Warning

When the scheduler is suspended, no tasks will execute, including timer tasks and periodic operations. Ensure critical system functions are maintained during suspension.

Note

This function must be called from within a task context, not from main() or initialization code.

Example Usage:

```
 void myTaskFunction(xTask task, xTaskParm parm) {
  // Normal task operations...
  if (needToSuspendScheduler) {
    // Suspend scheduler and return control xTaskSuspendAll();
    // Execution returns to line after xTaskStartScheduler() in main()
  }
}
int main(void) {
  xTask task;
  xTaskCreate(&task, "MyTask", myTaskFunction, NULL);
  xTaskResume(task);
  xTaskStartScheduler(); // Will return here when xTaskSuspendAll() called
  // Code here executes after scheduler suspension
performCriticalOperation();
  // Resume if needed xTaskResumeAll();
  xTaskStartScheduler();
}
```

Returns

ReturnOK on success, ReturnError on failure.

See also

xTaskStartScheduler() - Start the scheduler

xTaskResumeAll() - Resume scheduler state

xTaskGetSchedulerState() - Query current scheduler state

SchedulerState_t - Scheduler state enumeration

4.2.4.91 xTaskWait() `xReturn xTaskWait ( xTask task_ )`

Transitions a task to the TaskStateWaiting state, where it will not be scheduled for execution until a specific event occurs. This enables efficient event-driven multitasking where tasks sleep until notified, reducing unnecessary CPU usage.

Unlike TaskStateSuspended (which requires explicit xTaskResume() to reactivate), tasks in TaskStateWaiting automatically become schedulable when their awaited event occurs. Once the event is consumed (via xTaskNotifyTake() or xTaskNotificationStateClear()), the task automatically returns to waiting state.

State Transition:

- TaskStateRunning → TaskStateWaiting: Task enters event-waiting mode
- TaskStateSuspended → TaskStateWaiting: Task enters event-waiting mode
- TaskStateWaiting → TaskStateWaiting: No effect (already waiting)

Supported Event Types:

1. Task Timers: Task wakes when its timer expires (via xTaskResetTimer())
2. Direct-to-Task Notifications: Task wakes when another task sends a notification (via xTaskNotifyGive())

Event-Driven Workflow:

1. Task calls xTaskWait() to enter waiting state 2. Task stops executing and does not consume CPU 3. Another task or timer triggers an event 4. Scheduler automatically schedules the waiting task 5. Task executes and processes the event 6. Task consumes the event (xTaskNotifyTake() or xTaskNotificationStateClear()) 7. Task automatically returns to waiting state

Note

Event-driven tasks are more efficient than polling-based tasks as they only execute when needed, reducing CPU usage and power consumption.

A task in waiting state will remain inactive until its event occurs. To unconditionally activate a waiting task, use xTaskResume().

The event mechanism is edge-triggered. If an event occurs before xTaskWait() is called, the task will execute once to process it.

Example Usage:

```
 xTask buttonTask;
xTask ledTask;
// Button task waits for button press notifications void
buttonHandler(xTask task, xTaskParm parm) {
  xTaskNotification notification;
  // Wait for button press event xTaskWait(task);
  // Check if notification is waiting if
(OK(xTaskNotificationIsWaiting(task, &notification))) {
    // Process button press if (OK(xTaskNotifyTake(task, &notification))) {
      handleButtonPress();
      // Task automatically returns to waiting state
    }
  }
}
// Interrupt handler or another task sends notification void
buttonISR(void) {
  // Wake up button task xTaskNotifyGive(buttonTask);
}
int main(void) {
  xTaskCreate(&buttonTask, "Button  ", buttonHandler, NULL);
  xTaskResume(buttonTask);  // Initially resume the task
xTaskStartScheduler();
}
```

Parameters

in	*task*↩ _	Handle of the task to place in waiting state. Must be a valid task handle previously returned by xTaskCreate().

Returns

ReturnOK if the task state was successfully changed to waiting, ReturnError if the operation failed (invalid task handle or system error).

See also

xTaskCreate() - Create a new task

xTaskResume() - Activate a task (overrides waiting state)

xTaskSuspend() - Deactivate a task

xTaskNotifyGive() - Send a notification to wake a waiting task

xTaskNotifyTake() - Consume a notification event

xTaskNotificationIsWaiting() - Check for pending notifications

xTaskNotificationStateClear() - Clear notification state

xTaskResetTimer() - Reset task timer for timer-based events

TaskState_t - Task state enumeration

4.2.4.92 xTimerChangePeriod() xReturn xTimerChangePeriod (

```
        xTimer timer_,
        const xTicks period_ )
```

Modifies the expiration period of an application timer without requiring timer deletion and recreation. The new period takes effect immediately and will be used for all subsequent timer expirations. This operation allows dynamic adjustment of timer intervals based on runtime conditions, enabling adaptive timing behavior in response to system state or application needs.

When xTimerChangePeriod() is called, the timer's period is updated to the new value, but the timer's current state (active/stopped) and elapsed time are preserved. If the timer is running, it continues running with the new period. The timer will expire when the elapsed time reaches the new period value, which may be sooner or later than the original period.

Key characteristics:

- **Immediate effect**: New period applies to current and future timing cycles
- **State preservation**: Timer remains active or stopped as before the change
- **Elapsed time preserved**: Current elapsed time is not reset automatically
- **No recreation needed**: Avoids overhead of deleting and recreating timer
- **Non-blocking**: Returns immediately after period update

Common period change scenarios:

- **Adaptive sampling**: Adjust sensor polling rate based on value changes
- **Power management**: Slow down non-critical timers to conserve energy
- **Load balancing**: Distribute timer expirations to avoid processing spikes
- **Configuration updates**: Apply new timing settings from user preferences
- **Error recovery**: Increase retry intervals during communication failures

Example 1: Adaptive sensor polling

```
 xTimer sensorPollTimer;
xByte lastReading = 0;
void sensorTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(sensorPollTimer, &expired)) && expired) {
    xByte reading = readSensor();
    xByte change = abs(reading - lastReading);
    // Adapt polling rate based on change magnitude if (change > 20) {
      // Fast changes - poll rapidly xTimerChangePeriod(sensorPollTimer,
10);
    } else if (change > 5) {
      // Moderate changes - normal rate xTimerChangePeriod(sensorPollTimer,
50);
    } else {
      // Stable - slow down polling xTimerChangePeriod(sensorPollTimer,
200);
    }
    lastReading = reading;
    xTimerReset(sensorPollTimer);
  }
}
```

Example 2: Power-aware timing

```
 xTimer backgroundTimer;
typedef enum {
  POWER_MODE_ACTIVE, POWER_MODE_SLEEP, POWER_MODE_DEEP_SLEEP
} PowerMode_t;
void setPowerMode(PowerMode_t mode) {
  switch (mode) {
```

```
    case POWER_MODE_ACTIVE:
      xTimerChangePeriod(backgroundTimer, 50);   // Fast updates break;
    case POWER_MODE_SLEEP:
      xTimerChangePeriod(backgroundTimer, 500);  // Slower updates break;
    case POWER_MODE_DEEP_SLEEP:
      xTimerChangePeriod(backgroundTimer, 5000); // Minimal updates break;
  }
}
```

Example 3: Communication retry with exponential backoff

```
 xTimer retryTimer;
xTicks currentRetryInterval = 100;
void attemptConnection(void) {
  if (connectionSuccessful()) {
    // Reset to initial interval on success currentRetryInterval = 100;
    xTimerChangePeriod(retryTimer, currentRetryInterval);
    xTimerStop(retryTimer);
  } else {
    // Exponential backoff on failure (max 10 seconds) if
(currentRetryInterval < 10000) {
      currentRetryInterval *= 2;
      xTimerChangePeriod(retryTimer, currentRetryInterval);
    }
    xTimerReset(retryTimer);
  }
}
```

Example 4: User-configurable timer

```
 xTimer userTimer;
void applyUserSettings(xTicks newInterval) {
  // Validate interval range if (newInterval < 10) {
    newInterval = 10;     // Minimum 10 ticks
  } else if (newInterval > 60000) {
    newInterval = 60000;  // Maximum 60 seconds
  }
  // Apply new interval if (OK(xTimerChangePeriod(userTimer, newInterval)))
{
    // Reset to start timing with new period xTimerReset(userTimer);
    saveSettings(newInterval);
  }
}
```

Parameters

in	*timer*↩_	Handle to the timer whose period should be changed. Must be a valid timer created with xTimerCreate().
in	*period*↩_	The new timer period in system ticks. Must be greater than zero. The timer will expire when elapsed time equals this value.

Returns

ReturnOK if period changed successfully, ReturnError if operation failed due to invalid timer handle, timer not found, or invalid period value.

Warning

Changing the period does NOT automatically reset the elapsed time. If a timer has already accumulated elapsed time, it may expire immediately if the new period is less than the current elapsed time. Call xTimerReset() after changing the period if you want to start timing from zero with the new period.

Reducing the period on a running timer may cause immediate expiration if the elapsed time already exceeds the new period. Always check or reset the timer after reducing the period.

Note

xTimerChangePeriod() can be called whether the timer is running or stopped. The new period takes effect in both cases.

The timer's active state is preserved—if it was running, it continues running; if stopped, it remains stopped.

Changing to period value 0 is invalid and will return ReturnError. To disable a timer, use xTimerStop() instead.

See also

xTimerGetPeriod() - Query current timer period

xTimerReset() - Reset elapsed time after changing period

xTimerCreate() - Create timer with initial period

xTimerStart() - Start timer with its current period

xTimerStop() - Stop timer to prevent expiration

xTimerHasTimerExpired() - Check if timer has expired

4.2.4.93 xTimerCreate()

```
xReturn xTimerCreate (
        xTimer * timer_,
        const xTicks period_ )
```

Creates a software timer that can be used for measuring time intervals, implementing delays, or triggering periodic actions. Software timers are independent of task timers and provide general-purpose timing facilities for application use.

Software timers in HeliOS are NOT the same as task timers used for event-driven multitasking. Application timers are lightweight timing objects that can be:

- Started and stopped on demand
- Reset to restart counting from zero
- Queried to check if the period has elapsed
- Modified to change their period dynamically

Timer Operation: Timers count system ticks from when they are started. When the elapsed ticks exceed the configured period, the timer is considered "expired." Applications check timer expiration using xTimerHasTimerExpired() and can reset the timer to begin a new timing cycle.

Common Use Cases:

- Implementing timeout detection
- Creating delays without blocking
- Measuring elapsed time between events
- Implementing periodic operations (e.g., blinking LED)
- Rate limiting operations
- Watchdog functionality

Note

Software timers are passive timing objects. They do not automatically trigger callbacks or events when they expire. Applications must actively check timer status using xTimerHasTimerExpired().

Timer resolution is determined by the system tick rate, which depends on how frequently xTaskStartScheduler() executes (typically tied to hardware timer interrupts or main loop frequency).

Multiple timers can be created for different timing needs. Each timer maintains independent state and period configuration.

Example Usage:

```
 xTimer blinkTimer;
xTimer timeoutTimer;
// Create a timer with 1000 tick period (e.g., 1 second if 1ms ticks) if
(OK(xTimerCreate(&blinkTimer, 1000))) {
  // Start the timer xTimerStart(blinkTimer);
  // In your task or main loop:
  xBase expired;
  if (OK(xTimerHasTimerExpired(blinkTimer, &expired)) && expired) {
    // Timer has expired - toggle LED toggleLED();
    // Reset for next cycle xTimerReset(blinkTimer);
  }
}
// Create a timeout timer if (OK(xTimerCreate(&timeoutTimer, 5000))) {  //
5 second timeout xTimerStart(timeoutTimer);
  // Wait for operation with timeout while (!operationComplete) {
    xBase timedOut;
    if (OK(xTimerHasTimerExpired(timeoutTimer, &timedOut)) && timedOut) {
      // Timeout occurred handleTimeout();
      break;
    }
    // Continue waiting...
  }
  // Clean up xTimerDelete(timeoutTimer);
}
```

Parameters

out	*timer*↩_	Pointer to xTimer variable that will receive the timer handle. This handle is used in subsequent timer operations.
in	*period*↩_	Timer period in system ticks. The timer expires when elapsed ticks exceed this value. Must be greater than zero.

Returns

ReturnOK if the timer was successfully created, ReturnError if creation failed (out of memory, invalid parameters, or system error).

See also

xTimerDelete() - Delete a timer and free its resources

xTimerStart() - Start a timer counting

xTimerStop() - Stop a timer

xTimerReset() - Reset a timer to zero

xTimerHasTimerExpired() - Check if timer has expired

xTimerChangePeriod() - Change timer period

xTimerGetPeriod() - Get current timer period

xTimerIsTimerActive() - Check if timer is running

4.2.4.94 xTimerDelete() xReturn xTimerDelete (const xTimer *timer_*)

Permanently removes an application timer created with xTimerCreate() and releases all associated kernel resources. After deletion, the timer handle becomes invalid and must not be used in any subsequent timer operations. This operation is typically performed during cleanup, reconfiguration, or when a timer is no longer needed by the application.

xTimerDelete() immediately removes the timer from the kernel timer list regardless of its current state (stopped, running, or expired). If the timer was active, it is stopped automatically before deletion. No further timer events will occur, and all internal timer state is freed.

Key characteristics:

- **Immediate deletion**: Timer removed regardless of active/stopped state
- **Resource cleanup**: All kernel memory associated with timer is freed
- **Handle invalidation**: Timer handle cannot be reused after deletion
- **Idempotent operation**: Safe to call even if timer was never started
- **No blocking**: Returns immediately after cleanup completes

Typical deletion scenarios:

- **Application cleanup**: Remove timers during task or module shutdown
- **Dynamic timer management**: Delete and recreate timers with different periods
- **Resource reclamation**: Free timers no longer needed to reduce memory usage
- **Error recovery**: Clean up timers after configuration or initialization failures

Example 1: Timer lifecycle management

```
 xTimer watchdogTimer;
// Create and use timer if (OK(xTimerCreate(&watchdogTimer, 1000))) {
  xTimerStart(watchdogTimer);
  // ... use timer for some period ...
  // Clean up when done xTimerStop(watchdogTimer);
  xTimerDelete(watchdogTimer);
  // watchdogTimer handle is now invalid
}
```

Example 2: Dynamic timer reconfiguration

```
 xTimer intervalTimer;
void setMonitoringInterval(xTicks newInterval) {
  // Delete existing timer if present if (intervalTimer != null) {
    xTimerDelete(intervalTimer);
  }
  // Create new timer with updated period if
(OK(xTimerCreate(&intervalTimer, newInterval))) {
    xTimerStart(intervalTimer);
  }
}
```

Example 3: Cleanup multiple timers

```
 xTimer timers[MAX_SENSORS];
xBase timerCount = 0;
void initSensorTimers(void) {
  for (xBase i = 0; i < MAX_SENSORS; i++) {
    if (OK(xTimerCreate(&timers[i], 100 * (i + 1)))) {
      xTimerStart(timers[i]);
      timerCount++;
    }
  }
}
void shutdownSensorTimers(void) {
  for (xBase i = 0; i < timerCount; i++) {
```

```
    xTimerDelete(timers[i]);
  }
  timerCount = 0;
}
```

Example 4: Conditional timer cleanup

```
 xTimer periodicTimer;
xBase timerActive = 0;
void disablePeriodicTask(void) {
  if (timerActive) {
    xTimerDelete(periodicTimer);
    timerActive = 0;
  }
}
void enablePeriodicTask(xTicks period) {
  // Clean up old timer if exists disablePeriodicTask();
  // Create new timer if (OK(xTimerCreate(&periodicTimer, period))) {
    xTimerStart(periodicTimer);
    timerActive = 1;
  }
}
```

Parameters

in	*timer*↩ _	Handle to the timer to delete. Must be a valid timer created with xTimerCreate(). After deletion, this handle becomes invalid.

Returns

ReturnOK if timer deleted successfully, ReturnError if deletion failed due to invalid timer handle or timer not found.

Warning

After xTimerDelete() returns successfully, the timer handle is invalid and must not be used in any subsequent operations. Using a deleted timer handle will result in ReturnError.

Deleting a timer that is referenced by multiple tasks can lead to errors if those tasks attempt to use the timer after deletion. Coordinate timer deletion across all tasks using the timer.

Note

xTimerDelete() automatically stops the timer before deletion if it was running. You do not need to explicitly call xTimerStop() first.

Unlike task and queue deletion, timer deletion does not require waiting for operations to complete. The timer is removed immediately.

Deletion is the only way to reclaim kernel memory allocated for a timer. Simply stopping a timer does not free its resources.

See also

xTimerCreate() - Create an application timer

xTimerStop() - Stop a running timer without deleting it

xTimerStart() - Start or restart a timer

xTimerReset() - Reset timer elapsed time to zero

xTimerIsTimerActive() - Check if timer is currently running

xTaskDelete() - Delete a task (similar resource cleanup pattern)

4.2.4.95 xTimerGetPeriod()

```
xReturn xTimerGetPeriod (
        const xTimer timer_,
        xTicks * period_ )
```

Retrieves the configured expiration period for an application timer in system ticks. This non-destructive query allows tasks to inspect timer configuration without affecting timer state or operation. The returned period value reflects the most recent setting from xTimerCreate() or xTimerChangePeriod().

xTimerGetPeriod() provides read-only access to the timer's period configuration. This is useful for validation, diagnostics, dynamic adjustments based on current settings, or coordination between multiple tasks that share timer resources. The query does not modify the timer state, elapsed time, or active/stopped status.

Common use cases:

- **Configuration validation**: Verify timer period matches expected values
- **Dynamic adjustments**: Calculate new periods based on current setting
- **Diagnostics and logging**: Report timer configuration for debugging
- **Coordinated timing**: Synchronize related timers based on period ratios
- **Settings persistence**: Save current timer configuration to non-volatile memory

Example 1: Validate timer configuration

```
 xTimer watchdogTimer;
void initWatchdog(void) {
  xTicks expectedPeriod = 5000;  // 5 seconds
  if (OK(xTimerCreate(&watchdogTimer, expectedPeriod))) {
    // Verify timer created with correct period xTicks actualPeriod;
    if (OK(xTimerGetPeriod(watchdogTimer, &actualPeriod))) {
      if (actualPeriod == expectedPeriod) {
        xTimerStart(watchdogTimer);
      } else {
        // Configuration mismatch - handle error logError("Watchdog period
mismatch");
      }
    }
  }
}
```

Example 2: Adaptive period adjustment

```
 xTimer pollTimer;
void speedUpPolling(void) {
  xTicks currentPeriod;
  if (OK(xTimerGetPeriod(pollTimer, &currentPeriod))) {
    // Halve the period (double the rate), but enforce minimum xTicks
newPeriod = currentPeriod / 2;
    if (newPeriod < 10) {
      newPeriod = 10;  // Minimum 10 ticks
    }
    xTimerChangePeriod(pollTimer, newPeriod);
    xTimerReset(pollTimer);
  }
}
void slowDownPolling(void) {
  xTicks currentPeriod;
  if (OK(xTimerGetPeriod(pollTimer, &currentPeriod))) {
    // Double the period (halve the rate), but enforce maximum xTicks
newPeriod = currentPeriod * 2;
    if (newPeriod > 10000) {
      newPeriod = 10000;  // Maximum 10 seconds
    }
    xTimerChangePeriod(pollTimer, newPeriod);
    xTimerReset(pollTimer);
  }
}
```

Example 3: Coordinated timer relationships

```
 xTimer fastTimer;
xTimer slowTimer;
void setupCoordinatedTimers(void) {
  // Fast timer runs at base rate xTimerCreate(&fastTimer, 100);
  xTimerStart(fastTimer);
```

```
  // Slow timer should run at 10x the fast timer period xTicks fastPeriod;
  if (OK(xTimerGetPeriod(fastTimer, &fastPeriod))) {
    xTimerCreate(&slowTimer, fastPeriod * 10);
    xTimerStart(slowTimer);
  }
}
```

Example 4: System diagnostics and reporting

```
 xTimer timers[MAX_TIMERS];
xBase timerCount = 0;
void reportTimerStatus(void) {
  for (xBase i = 0; i < timerCount; i++) {
    xTicks period;
    xBase isActive;
    xBase hasExpired;
    if (OK(xTimerGetPeriod(timers[i], &period))) {
      xTimerIsTimerActive(timers[i], &isActive);
      xTimerHasTimerExpired(timers[i], &hasExpired);
      // Log timer status printf("Timer %d: period=%lu, active=%d,
expired=%d\n", i, period, isActive, hasExpired);
    }
  }
}
```

Parameters

in	*timer*↩_	Handle to the timer to query. Must be a valid timer created with xTimerCreate().
out	*period*↩_	Pointer to variable receiving the timer period in system ticks. On success, contains the current period value.

Returns

ReturnOK if period retrieved successfully, ReturnError if query failed due to invalid timer handle or timer not found.

Warning

This function returns the configured period, not the elapsed time or remaining time. To determine how close a timer is to expiration, you need to track elapsed time or check expiration status with xTimerHasTimerExpired().

Note

xTimerGetPeriod() is a non-destructive query that does not modify timer state, elapsed time, or active/stopped status.

The returned period reflects the most recent value set by either xTimerCreate() (initial period) or xTimerChangePeriod() (updated period).

HeliOS does not provide direct access to elapsed time. To track progress toward expiration, implement your own elapsed time tracking using the scheduler's tick count.

See also

xTimerChangePeriod() - Modify timer period

xTimerCreate() - Create timer with initial period

xTimerIsTimerActive() - Check if timer is running

xTimerHasTimerExpired() - Check if timer has expired

xTimerReset() - Reset elapsed time to zero

xTimerStart() - Start timer with its current period

4.2.4.96 xTimerHasTimerExpired()

```
xReturn xTimerHasTimerExpired (
        const xTimer timer_,
        xBase * res_ )
```

Queries whether an application timer's elapsed time has reached or exceeded its configured period, indicating that the timer has expired. This query is the primary mechanism for detecting timer events in polling-based timing patterns. Once expired, the timer remains in the expired state until reset with xTimerReset() or stopped with xTimerStop().

A timer can only expire if it is active (started with xTimerStart()). Inactive timers never expire, even if sufficient time has passed. The expired state is "sticky"—once a timer expires, it remains expired until explicitly cleared, allowing multiple tasks to observe the expiration event without race conditions.

Typical expiration handling workflow:

1. Check if timer has expired with xTimerHasTimerExpired() 2. If expired, perform the timed action 3. Reset the timer with xTimerReset() to clear expiration and restart timing 4. Repeat the cycle

Key characteristics:

- **Sticky expiration**: Remains expired until xTimerReset() or xTimerStop()
- **Requires active timer**: Only active timers can expire
- **Non-destructive query**: Does not clear expiration or modify state
- **Event detection**: Primary method for implementing periodic actions

Example 1: Periodic sensor polling

```
 xTimer pollTimer;
void initSensor(void) {
  xTimerCreate(&pollTimer, 100);  // Poll every 100 ticks
xTimerStart(pollTimer);
}
void sensorTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(pollTimer, &expired)) && expired) {
    // Timer expired - read sensor xByte sensorValue = readSensor();
    processSensorData(sensorValue);
    // Reset for next period xTimerReset(pollTimer);
  }
}
```

Example 2: Watchdog timeout detection

```
 xTimer watchdogTimer;
void initWatchdog(void) {
  xTimerCreate(&watchdogTimer, 5000);  // 5 second timeout
xTimerStart(watchdogTimer);
}
void watchdogTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(watchdogTimer, &expired)) && expired) {
    // Watchdog expired - system not responding logError("Watchdog timeout
- system hung");
    performSystemReset();
  }
}

void petWatchdog(void) {
  // Called periodically by application to prevent timeout
xTimerReset(watchdogTimer);
}
```

Example 3: Multi-rate execution with multiple timers

```
 xTimer fastTimer, mediumTimer, slowTimer;
void initTimers(void) {
  xTimerCreate(&fastTimer, 10);    // Every 10 ticks
xTimerCreate(&mediumTimer, 100); // Every 100 ticks
xTimerCreate(&slowTimer, 1000);  // Every 1000 ticks
  xTimerStart(fastTimer);
```

```
  xTimerStart(mediumTimer);
  xTimerStart(slowTimer);
}
void controlTask(xTask task, xTaskParm parm) {
  xBase expired;
  // Fast update - 10 ticks if (OK(xTimerHasTimerExpired(fastTimer,
&expired)) && expired) {
    updateFastController();
    xTimerReset(fastTimer);
  }
  // Medium update - 100 ticks if (OK(xTimerHasTimerExpired(mediumTimer,
&expired)) && expired) {
    updateMediumController();
    xTimerReset(mediumTimer);
  }
  // Slow update - 1000 ticks if (OK(xTimerHasTimerExpired(slowTimer,
&expired)) && expired) {
    updateSlowController();
    xTimerReset(slowTimer);
  }
}
```

Example 4: Timeout for communication protocol

```
 xTimer responseTimer;
typedef enum {
  STATE_IDLE, STATE_WAITING_RESPONSE, STATE_COMPLETE
} CommState_t;
CommState_t commState = STATE_IDLE;
void sendRequest(void) {
  transmitData();
  commState = STATE_WAITING_RESPONSE;
  // Start timeout timer xTimerReset(responseTimer);
  xTimerStart(responseTimer);
}
void commTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (commState == STATE_WAITING_RESPONSE) {
    if (OK(xTimerHasTimerExpired(responseTimer, &expired)) && expired) {
      // Timeout - no response received logError("Communication timeout");
      xTimerStop(responseTimer);
      commState = STATE_IDLE;
      retryRequest();
    }
  }
}
void onResponseReceived(void) {
  // Stop timer on successful response xTimerStop(responseTimer);
  commState = STATE_COMPLETE;
  processResponse();
}
```

Parameters

in	*timer*↩ _	Handle to the timer to query. Must be a valid timer created with xTimerCreate().
out	*res*↩ _	Pointer to variable receiving the expiration status. Set to non-zero (true) if timer has expired, zero (false) if timer has not expired or is not active.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid timer handle or timer not found.

Warning

xTimerHasTimerExpired() does NOT automatically reset the timer. Once expired, the timer remains expired until you explicitly call xTimerReset() or xTimerStop(). Failing to reset an expired timer will cause it to appear expired on every subsequent check.

Inactive (stopped) timers never expire. Always verify the timer is active with xTimerStart() before expecting expiration events.

Note

The expiration state is "sticky"—it persists until cleared. This allows multiple tasks to safely check the same timer expiration without missing the event.

Expiration occurs when elapsed time >= period. At the moment the elapsed time equals the period, the timer transitions to expired state.

Unlike some RTOS timers that provide callback functions on expiration, HeliOS timers use polling with xTimerHasTimerExpired(). Tasks must explicitly check for expiration in their execution loops.

See also

xTimerReset() - Clear expiration and restart timing

xTimerStart() - Activate timer to enable expiration

xTimerStop() - Deactivate timer and clear expiration

xTimerIsTimerActive() - Check if timer is running

xTimerCreate() - Create timer with period

xTimerChangePeriod() - Modify timer period

4.2.4.97 xTimerIsTimerActive()

```
xReturn xTimerIsTimerActive (
    const xTimer timer_,
    xBase * res_ )
```

Queries whether an application timer is in the active (running) state. A timer is considered active if it has been started with xTimerStart() and has not been stopped with xTimerStop(). Active timers accumulate elapsed time on each scheduler tick and will eventually expire when elapsed time reaches the configured period.

This non-destructive query allows tasks to check timer state before performing operations, coordinate timing between tasks, or implement conditional logic based on whether timing is currently in progress. The query does not affect timer operation or elapsed time accumulation.

Timer state transitions:

- **Inactive (stopped)**: Initial state after creation, or after xTimerStop()
- **Active (running)**: After xTimerStart(), accumulating elapsed time
- **Active and expired**: Timer remains active until xTimerReset() or xTimerStop()

Key characteristics:

- **Non-destructive query**: Does not modify timer state or elapsed time
- **State only**: Returns active/stopped status, not expiration status
- **Instant snapshot**: Reflects current state at time of call

Example 1: Conditional timer start

```
 xTimer processingTimer;
void startProcessing(void) {
  xBase isActive;
  // Only start if not already running if
(OK(xTimerIsTimerActive(processingTimer, &isActive))) {
    if (!isActive) {
      xTimerReset(processingTimer);
      xTimerStart(processingTimer);
      beginProcessing();
    } else {
      logWarning("Processing already in progress");
    }
  }
}
```

Example 2: Watchdog monitoring

```
 xTimer watchdogTimer;
void checkWatchdog(void) {
  xBase isActive;
  xBase hasExpired;
  if (OK(xTimerIsTimerActive(watchdogTimer, &isActive))) {
    if (!isActive) {
      // Watchdog not running - critical error handleWatchdogFailure();
    } else if (OK(xTimerHasTimerExpired(watchdogTimer, &hasExpired)) &&
hasExpired) {
      // Watchdog expired - system hung handleWatchdogTimeout();
    } else {
      // Watchdog running normally
    }
  }
}
void petWatchdog(void) {
  xBase isActive;
  if (OK(xTimerIsTimerActive(watchdogTimer, &isActive)) && isActive) {
    xTimerReset(watchdogTimer);  // Reset only if active
  }
}
```

Example 3: Performance measurement

```
 xTimer perfTimer;
void startMeasurement(void) {
  xBase isActive;
  if (OK(xTimerIsTimerActive(perfTimer, &isActive)) && isActive) {
    logWarning("Measurement already in progress");
    return;
  }
  xTimerReset(perfTimer);
  xTimerStart(perfTimer);
}
void endMeasurement(void) {
  xBase isActive;
  if (OK(xTimerIsTimerActive(perfTimer, &isActive)) && isActive) {
    xTimerStop(perfTimer);
    // Analyze performance metrics
  } else {
    logError("No measurement in progress");
  }
}
```

Example 4: System diagnostics

```
 xTimer systemTimers[MAX_TIMERS];
xBase timerCount = 0;
void reportSystemStatus(void) {
  xBase activeCount = 0;
  xBase expiredCount = 0;
  for (xBase i = 0; i < timerCount; i++) {
    xBase isActive, hasExpired;
    if (OK(xTimerIsTimerActive(systemTimers[i], &isActive)) && isActive) {
      activeCount++;
      if (OK(xTimerHasTimerExpired(systemTimers[i], &hasExpired)) &&
hasExpired) {
        expiredCount++;
      }
    }
  }
  printf("Timers: %d total, %d active, %d expired\n", timerCount,
activeCount, expiredCount);
}
```

Parameters

in	*timer_*	Handle to the timer to query. Must be a valid timer created with xTimerCreate().
out	*res_*	Pointer to variable receiving the active state. Set to non-zero (true) if timer is active/running, zero (false) if timer is stopped.

Returns

ReturnOK if query succeeded, ReturnError if query failed due to invalid timer handle or timer not found.

Warning

Active state does not indicate whether the timer has expired. A timer can be both active and expired simultaneously. Use xTimerHasTimerExpired() to check expiration status.

Note

xTimerIsTimerActive() returns the state at the moment of the call. In multitasking environments, the state may change immediately after if another task starts or stops the timer.

A timer that has expired remains active until explicitly stopped with xTimerStop() or reset with xTimerReset(). Expiration does not automatically stop the timer.

Newly created timers are inactive by default. They must be explicitly started with xTimerStart() to begin accumulating elapsed time.

See also

xTimerStart() - Start timer (make it active)

xTimerStop() - Stop timer (make it inactive)

xTimerHasTimerExpired() - Check if active timer has expired

xTimerReset() - Reset elapsed time while keeping timer active

xTimerCreate() - Create timer in inactive state

4.2.4.98 xTimerReset() `xReturn xTimerReset ( xTimer timer_ )`

Clears an application timer's accumulated elapsed time, returning it to the state immediately after creation or start. This operation is the primary method for acknowledging timer expiration and beginning a new timing cycle. The timer's configured period and active/stopped state are preserved—only the elapsed time is reset to zero.

xTimerReset() is typically called after detecting timer expiration with xTimerHasTimerExpired() to clear the expired state and restart the timing cycle for periodic operations. The reset does not affect whether the timer is active or stopped; an active timer continues running from zero elapsed time, while a stopped timer remains stopped with zero elapsed time.

Key characteristics:

- **Clears elapsed time**: Resets accumulated time to zero

- **Clears expiration**: Timer no longer reports as expired after reset
- **Preserves state**: Active/stopped state unchanged
- **Preserves period**: Configured timer period unchanged
- **Immediate effect**: Next expiration occurs one full period after reset

Common reset scenarios:

- **Periodic operations**: Reset after each expiration to start next cycle
- **Watchdog petting**: Reset watchdog timer to prevent timeout
- **Event acknowledgment**: Clear timer after handling timed event
- **Synchronization**: Align timer phases with system events
- **Period changes**: Reset after calling xTimerChangePeriod() to start fresh

Example 1: Periodic sensor polling pattern

```
 xTimer sensorTimer;
void initSensor(void) {
  xTimerCreate(&sensorTimer, 100);  // 100 tick period
xTimerStart(sensorTimer);
}
void sensorTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(sensorTimer, &expired)) && expired) {
    // Read and process sensor xByte value = readSensor();
    processSensorValue(value);
    // Reset for next cycle xTimerReset(sensorTimer);
  }
}
```

Example 2: Watchdog timer implementation

```
 xTimer watchdogTimer;
void initWatchdog(void) {
  xTimerCreate(&watchdogTimer, 5000);  // 5 second timeout
xTimerStart(watchdogTimer);
}
void petWatchdog(void) {
  // Called periodically by application to prevent timeout
xTimerReset(watchdogTimer);
}
void watchdogTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(watchdogTimer, &expired)) && expired) {
    // System hung - watchdog not reset in time logCritical("Watchdog
timeout");
    performSystemReset();
  }
}
```

Example 3: Debouncing button input

```
 xTimer debounceTimer;
xBase buttonPressed = 0;
void onButtonPress(void) {
  if (!buttonPressed) {
    // First press - start debounce timer buttonPressed = 1;
    xTimerReset(debounceTimer);
    xTimerStart(debounceTimer);
    handleButtonPress();
  }
}
void buttonTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (buttonPressed) {
    if (OK(xTimerHasTimerExpired(debounceTimer, &expired)) && expired) {
      // Debounce period elapsed - ready for next press buttonPressed = 0;
      xTimerStop(debounceTimer);
    }
  }
}
```

Example 4: Synchronized multi-timer operations

```
 xTimer timer1, timer2, timer3;
void synchronizeTimers(void) {
  // Reset all timers simultaneously to align phases xTimerReset(timer1);
  xTimerReset(timer2);
  xTimerReset(timer3);
  // All timers now start from zero elapsed time
}
void onSystemEvent(void) {
  // Synchronize timer phases on important system event
synchronizeTimers();
}
```

Parameters

in	*timer*↩ _	Handle to the timer to reset. Must be a valid timer created with xTimerCreate().

Returns

ReturnOK if timer reset successfully, ReturnError if operation failed due to invalid timer handle or timer not found.

Warning

xTimerReset() only clears elapsed time—it does NOT start a stopped timer. If the timer is not active, call xTimerStart() before or after xTimerReset() to begin timing.

After reset, the timer will not expire until a full period has elapsed. If you reset a timer that has partially elapsed, you lose that partial progress and must wait the full period again.

Note

xTimerReset() is typically called after detecting expiration to acknowledge the event and start a new timing cycle.

For periodic operations, the pattern is: check expired → perform action → reset timer. This ensures you don't miss expiration events.

Resetting does not change the timer's period. To change the period, use xTimerChangePeriod() and then optionally xTimerReset().

Calling xTimerReset() on an already-reset or newly-created timer is safe and has no adverse effects.

See also

xTimerHasTimerExpired() - Check if timer has expired before resetting

xTimerStart() - Activate timer after reset

xTimerStop() - Stop timer (also clears expiration)

xTimerChangePeriod() - Change period (usually followed by reset)

xTimerCreate() - Create timer (starts with zero elapsed time)

xTimerIsTimerActive() - Check if timer is running

4.2.4.99 xTimerStart() xReturn xTimerStart (xTimer *timer_*)

Activates an application timer, transitioning it from the stopped (inactive) state to the running (active) state. Once started, the timer begins accumulating elapsed time on each scheduler tick and will eventually expire when elapsed time reaches the configured period. This operation is required to enable timer expiration—only active timers can expire.

xTimerStart() preserves the timer's current elapsed time. If you start a timer that was previously stopped with xTimerStop(), it resumes from the elapsed time it had accumulated before stopping. To start timing from zero, call xTimerReset() before or after xTimerStart(). For newly created timers, elapsed time is already zero, so xTimerReset() is not necessary.

Key characteristics:

- **Activates timer**: Enables elapsed time accumulation
- **Preserves elapsed time**: Continues from current elapsed time
- **Enables expiration**: Only active timers can expire
- **Idempotent**: Starting an already-running timer has no effect
- **Non-blocking**: Returns immediately after state change

Typical startup sequences:

- **New timer, start from zero**: xTimerCreate() → xTimerStart()
- **New timer, explicit reset**: xTimerCreate() → xTimerReset() → xTimerStart()
- **Resume stopped timer**: xTimerStart() (continues from stopped elapsed time)
- **Restart from zero**: xTimerReset() → xTimerStart()

Example 1: Basic timer startup

```
 xTimer periodicTimer;
void initTimer(void) {
  // Create timer with 100 tick period if (OK(xTimerCreate(&periodicTimer,
100))) {
    // Start timer - begins timing from zero xTimerStart(periodicTimer);
  }
}
void periodicTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(periodicTimer, &expired)) && expired) {
    performPeriodicAction();
    xTimerReset(periodicTimer);  // Reset for next cycle
  }
}
```

Example 2: Conditional start based on system state

```
 xTimer monitorTimer;
void enableMonitoring(void) {
  xBase isActive;
  if (OK(xTimerIsTimerActive(monitorTimer, &isActive)) && !isActive) {
    // Timer not running - start it xTimerReset(monitorTimer);  // Start
from zero xTimerStart(monitorTimer);
    logInfo("Monitoring enabled");
  }
}
void disableMonitoring(void) {
  xBase isActive;
  if (OK(xTimerIsTimerActive(monitorTimer, &isActive)) && isActive) {
    xTimerStop(monitorTimer);
    logInfo("Monitoring disabled");
  }
}
```

Example 3: Pause and resume pattern

```
 xTimer processTimer;
void pauseProcessing(void) {
  // Stop timer - preserves elapsed time xTimerStop(processTimer);
  logInfo("Processing paused");
}
void resumeProcessing(void) {
  // Start timer - continues from paused elapsed time
xTimerStart(processTimer);
  logInfo("Processing resumed");
}
void resetAndRestart(void) {
  // Restart from zero elapsed time xTimerReset(processTimer);
  xTimerStart(processTimer);
  logInfo("Processing restarted from zero");
}
```

Example 4: Multiple timer coordination

```
 xTimer timer1, timer2, timer3;
void startAllTimers(void) {
  // Reset all to zero xTimerReset(timer1);
  xTimerReset(timer2);
  xTimerReset(timer3);
  // Start all simultaneously xTimerStart(timer1);
  xTimerStart(timer2);
  xTimerStart(timer3);
  logInfo("All timers started synchronously");
}
void stopAllTimers(void) {
  xTimerStop(timer1);
  xTimerStop(timer2);
  xTimerStop(timer3);
  logInfo("All timers stopped");
}
```

Parameters

in	*timer*↩ _	Handle to the timer to start. Must be a valid timer created with xTimerCreate().

Returns

ReturnOK if timer started successfully, ReturnError if operation failed due to invalid timer handle or timer not found.

Warning

xTimerStart() does NOT reset elapsed time. If you stopped a timer partway through its period and then start it again, it will continue from where it left off. Call xTimerReset() before xTimerStart() if you want to begin timing from zero.

Only active (started) timers can expire. Calling xTimerHasTimerExpired() on a stopped timer will always return false, even if sufficient time has passed.

Note

xTimerStart() is idempotent—calling it multiple times on an already-running timer has no effect and is not an error.

For newly created timers, xTimerReset() is not necessary before xTimerStart() because elapsed time is already zero.

The timer begins accumulating elapsed time immediately on the next scheduler tick after xTimerStart() is called.

See also

xTimerStop() - Stop timer (opposite operation)

xTimerReset() - Reset elapsed time to zero

xTimerCreate() - Create timer (starts in stopped state)

xTimerIsTimerActive() - Check if timer is running

xTimerHasTimerExpired() - Check for expiration (requires active timer)

xTimerChangePeriod() - Change timer period

4.2.4.100 xTimerStop() `xReturn xTimerStop ( xTimer timer_ )`

Deactivates an application timer, transitioning it from the running (active) state to the stopped (inactive) state. Once stopped, the timer ceases accumulating elapsed time and cannot expire, even if sufficient scheduler ticks pass. The timer's current elapsed time is preserved, allowing for pause-and-resume scenarios if needed.

xTimerStop() is the primary method for temporarily disabling timing without destroying the timer. Unlike xTimerDelete(), which permanently removes the timer, xTimerStop() keeps the timer structure intact and allows restarting with xTimerStart(). If the timer was expired when stopped, the expiration state is cleared—xTimerHas↩ TimerExpired() will return false after stopping.

Key characteristics:

- **Deactivates timer**: Halts elapsed time accumulation
- **Preserves elapsed time**: Current elapsed time is maintained
- **Clears expiration**: Expired state is reset to non-expired
- **Prevents expiration**: Stopped timers cannot expire
- **Idempotent**: Stopping an already-stopped timer has no effect
- **Non-blocking**: Returns immediately after state change

Common stop scenarios:

- **Conditional timing**: Stop timer when condition is met
- **Pause-resume patterns**: Stop to pause, xTimerStart() to resume
- **Event completion**: Stop timer after timed event finishes early
- **Power management**: Stop non-essential timers to reduce overhead
- **Timeout cancellation**: Stop timer when expected response arrives

Example 1: Communication timeout with early completion

```
 xTimer timeoutTimer;
void sendRequest(void) {
  transmitData();
  // Start timeout timer xTimerReset(timeoutTimer);
  xTimerStart(timeoutTimer);
}
void onResponseReceived(void) {
  // Response arrived - cancel timeout xTimerStop(timeoutTimer);
  processResponse();
}
void commTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(timeoutTimer, &expired)) && expired) {
    // Timeout - no response received xTimerStop(timeoutTimer);
    handleTimeout();
  }
}
```

Example 2: Conditional timer management

```
 xTimer activityTimer;
void onActivityDetected(void) {
  xBase isActive;
  // Start timer on activity if (OK(xTimerIsTimerActive(activityTimer,
&isActive)) && !isActive) {
    xTimerReset(activityTimer);
    xTimerStart(activityTimer);
  }
}
void onIdleCondition(void) {
  xBase isActive;
  // Stop timer when system goes idle if
(OK(xTimerIsTimerActive(activityTimer, &isActive)) && isActive) {
    xTimerStop(activityTimer);
  }
}
```

Example 3: Pause and resume timing

```
 xTimer processTimer;
void pauseProcessing(void) {
  // Stop timer - preserves elapsed time for resume
xTimerStop(processTimer);
  suspendProcessing();
}
void resumeProcessing(void) {
  // Start timer - continues from paused elapsed time
xTimerStart(processTimer);
  continueProcessing();
}
void cancelProcessing(void) {
  // Stop and reset - clears elapsed time xTimerStop(processTimer);
  xTimerReset(processTimer);
  cleanupProcessing();
}
```

Example 4: Power-aware timer management

```
 xTimer backgroundTimers[MAX_TIMERS];
xBase timerCount = 0;
void enterLowPowerMode(void) {
  // Stop all non-critical timers for (xBase i = 0; i < timerCount; i++) {
    xTimerStop(backgroundTimers[i]);
  }
  enableSleepMode();
}
void exitLowPowerMode(void) {
  // Resume all timers disableSleepMode();
  for (xBase i = 0; i < timerCount; i++) {
    xTimerStart(backgroundTimers[i]);
  }
}
```

Example 5: One-shot timer pattern

```
 xTimer oneShotTimer;
void startOneShotTimer(xTicks delay) {
  // Configure and start one-shot timer xTimerChangePeriod(oneShotTimer,
delay);
  xTimerReset(oneShotTimer);
  xTimerStart(oneShotTimer);
}
void timerTask(xTask task, xTaskParm parm) {
  xBase expired;
  if (OK(xTimerHasTimerExpired(oneShotTimer, &expired)) && expired) {
    // Timer expired - stop to prevent re-triggering
```

```
xTimerStop(oneShotTimer);
    performOneShotAction();
  }
}
```

Parameters

in	*timer*↩ _	Handle to the timer to stop. Must be a valid timer created with xTimerCreate().

Returns

ReturnOK if timer stopped successfully, ReturnError if operation failed due to invalid timer handle or timer not found.

Warning

xTimerStop() preserves the current elapsed time. If you later call xTimerStart() without calling xTimerReset(), the timer will continue from its stopped elapsed time, not from zero. Call xTimerReset() after xTimerStop() if you want to clear elapsed time.

Stopped timers cannot expire. xTimerHasTimerExpired() will return false for stopped timers regardless of elapsed time.

Note

xTimerStop() clears the expiration flag. Even if the timer was expired when stopped, xTimerHasTimerExpired() will return false after the stop.

xTimerStop() is idempotent—calling it multiple times on an already-stopped timer has no effect and is not an error.

Stopping a timer does not free its resources. The timer remains allocated and can be restarted with xTimerStart(). To free timer resources, use xTimerDelete().

Newly created timers are already in the stopped state. Calling xTimerStop() on a newly created timer has no effect but is not an error.

See also

xTimerStart() - Start timer (opposite operation)

xTimerReset() - Reset elapsed time (often called after stop)

xTimerDelete() - Permanently remove timer and free resources

xTimerIsTimerActive() - Check if timer is running

xTimerHasTimerExpired() - Check for expiration (always false for stopped timers)

xTimerCreate() - Create timer (starts in stopped state)

Index