HeliOS Developer's Guide

0.3.x Kernel

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

   QueueMessage_t

      Data structure for a message queue message **2**

2 File Index

2.1 File List

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

3 Data Structure Documentation

3.1 QueueMessage_t Struct Reference

Data structure for a message queue message.

```
#include <HeliOS.h>
```

Data Fields

- Base_t messageBytes
- char messageValue [CONFIG_MESSAGE_VALUE_BYTES]

3.1.1 Detailed Description

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

See also

    xQueueMessage

    xQueuePeek()

    xQueueReceive()

    xMemFree()

    CONFIG_MESSAGE_VALUE_BYTES

Warning

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

3.1.2 Field Documentation

3.1.2.1 messageBytes   Base_t QueueMessage_t::messageBytes

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

3.1.2.2 messageValue   char QueueMessage_t::messageValue[CONFIG_MESSAGE_VALUE_BYTES]

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

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

- HeliOS.h

3.2 SystemInfo_t Struct Reference

Data structure for system informaiton.

```
#include <HeliOS.h>
```

Data Fields

- char productName [OS_PRODUCT_NAME_SIZE]
- Base_t majorVersion
- Base_t minorVersion
- Base_t patchVersion
- Base_t numberOfTasks

3.2.1 Detailed Description

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

See also

> xSystemInfo
>
> xSystemGetSystemInfo()
>
> xMemFree()

Warning

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

3.2.2 Field Documentation

3.2.2.1 majorVersion Base_t SystemInfo_t::majorVersion

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

3.2.2.2 minorVersion Base_t SystemInfo_t::minorVersion

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

3.2.2.3 numberOfTasks Base_t SystemInfo_t::numberOfTasks

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

3.2.2.4 patchVersion Base_t SystemInfo_t::patchVersion

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

3.2.2.5 productName char SystemInfo_t::productName[OS_PRODUCT_NAME_SIZE]

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

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

- HeliOS.h

3.3 TaskInfo_t Struct Reference

Data structure for information about a task.

```
#include <HeliOS.h>
```

Data Fields

- Base_t id
- char name [CONFIG_TASK_NAME_BYTES]
- TaskState_t state
- Time_t lastRunTime
- Time_t totalRunTime

3.3.1 Detailed Description

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

See also

> xTaskInfo
>
> xTaskGetTaskInfo()
>
> xMemFree()
>
> CONFIG_TASK_NAME_BYTES

Warning

> The memory allocated for an instance of xTaskInfo must be freed using xMemFree().

3.3.2 Field Documentation

3.3.2.1 id `Base_t TaskInfo_t::id`

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

3.3.2.2 lastRunTime `Time_t TaskInfo_t::lastRunTime`

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

3.3.2.3 name `char TaskInfo_t::name[CONFIG_TASK_NAME_BYTES]`

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

3.3.2.4 state `TaskState_t TaskInfo_t::state`

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

3.3.2.5 totalRunTime `Time_t TaskInfo_t::totalRunTime`

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

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

- HeliOS.h

3.4 TaskNotification_t Struct Reference

Data structure for direct to task notifications.

```
#include <HeliOS.h>
```

Data Fields

- Base_t notificationBytes
- char notificationValue [CONFIG_NOTIFICATION_VALUE_BYTES]

3.4.1 Detailed Description

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

See also

> xTaskNotification
>
> xTaskNotifyTake()
>
> xMemFree()
>
> CONFIG_NOTIFICATION_VALUE_BYTES

Warning

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

3.4.2 Field Documentation

3.4.2.1 notificationBytes `Base_t TaskNotification_t::notificationBytes`

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

3.4.2.2 notificationValue `char TaskNotification_t::notificationValue[CONFIG_NOTIFICATION_VALUE_BYTES]`

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

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

- HeliOS.h

3.5 TaskRunTimeStats_t Struct Reference

Data structure for task runtime statistics.

```
#include <HeliOS.h>
```

Data Fields

- Base_t id
- Time_t lastRunTime
- Time_t totalRunTime

3.5.1 Detailed Description

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

See also

> xTaskRunTimeStats
>
> xTaskGetTaskRunTimeStats()
>
> xTaskGetAllRunTimeStats()
>
> xMemFree()

Warning

> The memory allocated for an instance of xTaskRunTimeStats must be freed using xMemFree().

3.5.2 Field Documentation

3.5.2.1 id Base_t TaskRunTimeStats_t::id

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

3.5.2.2 lastRunTime Time_t TaskRunTimeStats_t::lastRunTime

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

3.5.2.3 totalRunTime Time_t TaskRunTimeStats_t::totalRunTime

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

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

- HeliOS.h

4 File Documentation

4.1 config.h File Reference

Kernel header file for user definable settings.

Macros

- #define CONFIG_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 8u

    Define the size in bytes of the message queue message value.
- #define CONFIG_NOTIFICATION_VALUE_BYTES 8u

    Define the size in bytes of the direct to task notification value.
- #define CONFIG_TASK_NAME_BYTES 8u

    Define the size in bytes of the ASCII task name.
- #define CONFIG_HEAP_SIZE_IN_BLOCKS 512u

    Define the number of blocks in the heap.
- #define CONFIG_HEAP_BLOCK_SIZE 32u

    Define the heap block size in bytes.
- #define CONFIG_QUEUE_MINIMUM_LIMIT 5u

    Define the minimum value for a message queue limit.

4.1.1 Detailed Description

Author

Manny Peterson (mannymsp@gmail.com)

Version

0.3.1

Date

2022-01-31

Copyright

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

4.1.2 Macro Definition Documentation

4.1.2.1 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.

Note

> Neither this setting nor the arduino.cpp file are needed if the end-user does not plan to use assertions on the Arduino platform/tool-chain.

4.1.2.2 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

4.1.2.3 CONFIG_HEAP_BLOCK_SIZE `#define CONFIG_HEAP_BLOCK_SIZE 32u`

Setting CONFIG_HEAP_BLOCK_SIZE allows the end-user to define the size of a heap block in bytes. The block size should be set to achieve the best possible utilization of the heap. A block size that is too large will waste the heap for smaller requests for heap. A block size that is too small will waste heap on entries. The default value is 32 bytes. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

> xMemAlloc()
>
> xMemFree()
>
> CONFIG_HEAP_SIZE_IN_BLOCKS

4.1.2.4 CONFIG_HEAP_SIZE_IN_BLOCKS `#define CONFIG_HEAP_SIZE_IN_BLOCKS 512u`

Setting CONFIG_HEAP_SIZE_IN_BLOCKS allows the end-user to define the size of the heap in blocks. The size of a block in the heap is determined by the CONFIG_HEAP_BLOCK_SIZE which is represented in bytes. The size of the heap needs to be adjusted to fit the memory requirements of the end-user's application. By default the CONFIG_HEAP_SIZE_IN_BLOCKS setting is not defined. The literal must be appended with "u" to maintain MISRA C:2012 compliance.

See also

> xMemAlloc()
>
> xMemFree()
>
> CONFIG_HEAP_BLOCK_SIZE

Note

> To use the platform and/or tool-chain defaults, leave CONFIG_HEAP_SIZE_IN_BLOCKS undefined.

4.1.2.5 CONFIG_MESSAGE_VALUE_BYTES `#define CONFIG_MESSAGE_VALUE_BYTES 8u`

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

See also

> xQueueMessage

4.1.2.6 CONFIG_NOTIFICATION_VALUE_BYTES `#define CONFIG_NOTIFICATION_VALUE_BYTES 8u`

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

See also

> xTaskNotification

4.1.2.7 CONFIG_QUEUE_MINIMUM_LIMIT `#define CONFIG_QUEUE_MINIMUM_LIMIT 5u`

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

See also

> xQueueIsQueueFull()
>
> xQueueSend()
>
> xQueueCreate()

4.1.2.8 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 )`

4.1.2.9 CONFIG_TASK_NAME_BYTES `#define CONFIG_TASK_NAME_BYTES 8u`

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

See also

> xTaskInfo

4.2 HeliOS.h File Reference

Header file for end-user application code.

Data Structures

- struct TaskRunTimeStats_t

  Data structure for task runtime statistics.
- struct TaskInfo_t

  Data structure for information about a task.
- struct TaskNotification_t

  Data structure for direct to task notifications.
- struct QueueMessage_t

  Data structure for a message queue message.
- struct SystemInfo_t

  Data structure for system informaiton.

Macros

- #define DEREF_TASKPARM(t, p) *((t *)p)

  A C macro to simplify casting and dereferencing a task paramater.

Typedefs

- typedef uint8_t Base_t

  Type definition for the base data type.
- typedef TIME_T_TYPE Time_t

  Type definition for system time measured in microseconds.
- typedef void Task_t

  Stub type definition for the task type.
- typedef void TaskParm_t

  Type definition for the task parameter.
- typedef void Queue_t

  Stub type definition for the message queue type.
- typedef void Timer_t

  Stub type definition for the timer type.
- typedef Base_t xBase

  Type definition for the base data type.
- typedef Timer_t * xTimer

  Stub type definition for the timer type.
- typedef Queue_t * xQueue

  Stub type definition for the message queue type.
- typedef QueueMessage_t * xQueueMessage

  Data structure for a message queue message.
- typedef TaskNotification_t * xTaskNotification

  Data structure for direct to task notifications.
- typedef TaskInfo_t * xTaskInfo

  Data structure for information about a task.
- typedef TaskRunTimeStats_t * xTaskRunTimeStats

  Data structure for task runtime statistics.
- typedef Task_t * xTask

  Stub type definition for the task type.
- typedef TaskParm_t * xTaskParm

  Type definition for the task parameter.

- typedef Time_t xTime

    The type definition for time expressed in microseconds.
- typedef TaskState_t xTaskState

    Enumerated type for task states.
- typedef SchedulerState_t xSchedulerState

    Enumerated type for scheduler states.
- typedef SystemInfo_t ∗ xSystemInfo

    Data structure for system informaiton.

Enumerations

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

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

    Enumerated type for scheduler states.

Functions

- void SystemAssert (const char ∗file_, int line_)

    System call to handle assertions.
- void ∗ xMemAlloc (size_t size_)

    System call to allocate memory from the heap.
- void xMemFree (void ∗ptr_)

    System call to free memory allocated from the heap.
- size_t xMemGetUsed (void)

    System call to return the amount of allocated heap memory.
- size_t xMemGetSize (void ∗ptr_)

    System call to return the amount of heap memory allcoated for a pointer.
- xQueue xQueueCreate (xBase limit_)

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

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

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

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

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

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

    System call to send a message using a message queue.
- xQueueMessage xQueuePeek (xQueue queue_)

    System call to peek at the next message in a message queue.
- void xQueueDropMessage (xQueue queue_)

    System call to drop the next message in a message queue.
- xQueueMessage xQueueReceive (xQueue queue_)

    System call to receive the next message in the message queue.
- void xTaskStartScheduler (void)

System call to pass control to the HeliOS scheduler.

- void [xTaskResumeAll](void)

  System call to set scheduler state to running.

- void [xTaskSuspendAll](void)

  System call to set the scheduler state to suspended.

- [xSystemInfo xSystemGetSystemInfo](void)

  The [xSystemGetSystemInfo()](…) system call will return the type xSystemInfo containing information about the system including the OS (product) name, its version and how many tasks are currently in the running, suspended or waiting states.

- [xTask xTaskCreate](const char ∗name_, void(∗callback_)([xTask](…), [xTaskParm](…)), [xTaskParm](…) taskParameter_)

  System call to create a new task.

- void [xTaskDelete](…)([xTask](…) task_)

  System call to delete a task.

- [xTask xTaskGetHandleByName](const char ∗name_)

  System call to get a task's handle by its ASCII name.

- [xTask xTaskGetHandleById](…)([xBase](…) id_)

  System call to get a task's handle by its task identifier.

- [xTaskRunTimeStats xTaskGetAllRunTimeStats](…)([xBase](…) ∗tasks_)

  System call to return task runtime statistics for all tasks.

- [xTaskRunTimeStats xTaskGetTaskRunTimeStats](…)([xTask](…) task_)

  System call to return task runtime statistics for the specified task.

- [xBase xTaskGetNumberOfTasks](void)

  System call to return the number of tasks regardless of their state.

- [xTaskInfo xTaskGetTaskInfo](…)([xTask](…) task_)

  System call to return the details of a task.

- [xTaskInfo](…) ∗ [xTaskGetAllTaskInfo](…)([xBase](…) ∗tasks_)

  System call to return the details of all tasks.

- [xTaskState xTaskGetTaskState](…)([xTask](…) task_)

  System call to return the state of a task.

- char ∗ [xTaskGetName](…)([xTask](…) task_)

  System call to return the ASCII name of a task.

- [xBase xTaskGetId](…)([xTask](…) task_)

  System call to return the task identifier for a task.

- void [xTaskNotifyStateClear](…)([xTask](…) task_)

  System call to clear a waiting direct to task notification.

- [xBase xTaskNotificationIsWaiting](…)([xTask](…) task_)

  System call to check if a direct to task notification is waiting.

- [Base_t xTaskNotifyGive](…)([xTask](…) task_, [xBase](…) notificationBytes_, const char ∗notificationValue_)

  System call to give another task a direct to task notification.

- [xTaskNotification xTaskNotifyTake](…)([xTask](…) task_)

  System call to take a direct to task notification from another task.

- void [xTaskResume](…)([xTask](…) task_)

  System call to resume a task.

- void [xTaskSuspend](…)([xTask](…) task_)

  System call to suspend a task.

- void [xTaskWait](…)([xTask](…) task_)

  System call to place a task in a waiting state.

- void [xTaskChangePeriod](…)([xTask](…) task_, [xTime](…) timerPeriod_)

  System call to set the task timer period.

- [xTime xTaskGetPeriod](…)([xTask](…) task_)

  System call to get the task timer period.

- void xTaskResetTimer (xTask task_)

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

  System call to get the state of the scheduler.
- xTimer xTimerCreate (xTime timerPeriod_)

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

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

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

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

  System call to check if a timer is active.
- xBase xTimerHasTimerExpired (xTimer timer_)

  System call to check if a timer has expired.
- void xTimerReset (xTimer timer_)

  System call to reset a timer.
- void xTimerStart (xTimer timer_)

  System call to start a timer.
- void xTimerStop (xTimer timer_)

  The xTimerStop() system call will place the timer in the stopped state. Neither xTimerStart() nor xTimerStop() will reset the timer. Timers can only be reset with xTimerReset().
- void xSystemHalt (void)

  The xSystemHalt() system call will halt HeliOS.

4.2.1   Detailed Description

Author

Manny Peterson (mannymsp@gmail.com)

Version

0.3.1

Date

2022-01-31

Copyright

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

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

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

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

4.2.2 Macro Definition Documentation

4.2.2.1 DEREF_TASKPARM `#define DEREF_TASKPARM(`
 `t,`
 `p ) *((t *)p)`

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

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

Parameters

t	The data type to cast the task paramater to (e.g., int).
p	The task pointer, often named parm_.

4.2.3 Typedef Documentation

4.2.3.1 Base_t `typedef uint8_t Base_t`

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

See also

> xBase

4.2.3.2 Queue_t `typedef void Queue_t`

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

See also

> xQueue
>
> xQueueDelete()

Warning

> The memory allocated for an instance of xQueue must be freed using xQueueDelete().

4.2.3.3 Task_t `typedef void Task_t`

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

See also

> xTask
>
> xTaskDelete()

Warning

> The memory allocated for an instance of xTask must be freed by xTaskDelete()

4.2.3.4 TaskParm_t `typedef void TaskParm_t`

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

See also

> xTaskParm
>
> xMemAlloc()
>
> xMemFree()

Warning

> The memory allocated for an instance of xTaskParm must be freed using xMemFree().

4.2.3.5 Time_t `typedef TIME_T_TYPE Time_t`

The type definition for time expressed in microseconds.

The Time_t type is used to store system time which is measured in microseconds from system initialization. Despite its name, this type does not store real time clock (RTC) time or date information.

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

4.2.3.6 Timer_t `typedef void Timer_t`

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

See also

xTimer

xTimerDelete()

Warning

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

4.2.3.7 xBase `typedef Base_t xBase`

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

See also

Base_t

4.2.3.8 xQueue `typedef Queue_t* xQueue`

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

See also

Queue_t

xQueueDelete()

Warning

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

4.2.3.9 xQueueMessage `typedef QueueMessage_t* xQueueMessage`

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

See also

> QueueMessage_t
>
> xQueuePeek()
>
> xQueueReceive()
>
> xMemFree()
>
> CONFIG_MESSAGE_VALUE_BYTES

Warning

> The memory allocated for an instance of xQueueMessage must be freed using xMemFree().

4.2.3.10 xSchedulerState `typedef SchedulerState_t xSchedulerState`

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

See also

> xSchedulerState
>
> xTaskSuspendAll()
>
> xTaskResumeAll()
>
> xTaskGetSchedulerState()

4.2.3.11 xSystemInfo `typedef SystemInfo_t* xSystemInfo`

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

See also

> SystemInfo_t
>
> xSystemGetSystemInfo()
>
> xMemFree()

Warning

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

4.2.3.12 **xTask** `typedef Task_t* xTask`

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

See also

>   Task_t
>
>   xTaskCreate()
>
>   xTaskDelete()

Warning

>   The memory allocated for an instance of xTask must be freed by xTaskDelete()

4.2.3.13 **xTaskInfo** `typedef TaskInfo_t* xTaskInfo`

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

See also

>   TaskInfo_t
>
>   xTaskGetTaskInfo()
>
>   xMemFree()
>
>   CONFIG_TASK_NAME_BYTES

Warning

>   The memory allocated for an instance of xTaskInfo must be freed using xMemFree().

4.2.3.14 **xTaskNotification** `typedef TaskNotification_t* xTaskNotification`

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

See also

>   TaskNotification_t
>
>   xTaskNotifyTake()
>
>   xMemFree()
>
>   CONFIG_NOTIFICATION_VALUE_BYTES

Warning

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

4.2.3.15   xTaskParm `typedef TaskParm_t* xTaskParm`

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

See also

> TaskParm_t
>
> xMemAlloc()
>
> xMemFree()

Warning

> The memory allocated for an instance of xTaskParm must be freed using xMemFree().

4.2.3.16   xTaskRunTimeStats `typedef TaskRunTimeStats_t* xTaskRunTimeStats`

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

See also

> TaskRunTimeStats_t
>
> xTaskGetTaskRunTimeStats()
>
> xTaskGetAllRunTimeStats()
>
> xMemFree()

Warning

> The memory allocated for an instance of xTaskRunTimeStats must be freed using xMemFree().

4.2.3.17   xTaskState `typedef TaskState_t xTaskState`

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

See also

> TaskState_t
>
> xTaskResume()
>
> xTaskSuspend()
>
> xTaskWait()

4.2.3.18 xTime `typedef Time_t xTime`

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

See also

    Time_t

4.2.3.19 xTimer `typedef Timer_t* xTimer`

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

See also

    Timer_t

    xTimerDelete()

Warning

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

4.2.4 Enumeration Type Documentation

4.2.4.1 SchedulerState_t `enum SchedulerState_t`

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

See also

    xSchedulerState

    xTaskSuspendAll()

    xTaskResumeAll()

Enumerator

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

4.2.4.2 TaskState_t `enum TaskState_t`

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

See also

xTaskState

xTaskResume()

xTaskSuspend()

xTaskWait()

Enumerator

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

4.2.5 Function Documentation

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

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

See also

SYSASSERT

CONFIG_ENABLE_SYSTEM_ASSERT

CONFIG_SYSTEM_ASSERT_BEHAVIOR

Parameters

file←_	This is automatically defined by the compiler's definition of *FILE*
line←_	This is automatically defined by the compiler's definition of *LINE*

4.2.5.2 xMemAlloc() `void * xMemAlloc (`
            `size_t size_ )`

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

See also

> CONFIG_HEAP_SIZE_IN_BLOCKS
>
> CONFIG_HEAP_BLOCK_SIZE
>
> xMemFree()

Parameters

size↩ *_*	The amount (size) of the memory to be allocated from the heap in bytes.

Returns

> void∗ If successful, xMemAlloc() returns a pointer to the newly allocated memory. If unsuccessful, the system call will return null.

Note

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

4.2.5.3 xMemFree() `void xMemFree (`
            `void * ptr_ )`

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

See also

> xMemAlloc()

Parameters

ptr↩ *_*	The pointer to the allocated heap memory to be freed.

Warning

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

4.2.5.4 xMemGetSize() `size_t xMemGetSize (`
              `void * ptr_ )`

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

Parameters

ptr← *_*	The pointer to the allocated heap memory to obtain the size of the memory, in bytes, that is allocated.

Returns

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

Note

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

4.2.5.5 xMemGetUsed() `size_t xMemGetUsed (`
              `void )`

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

Returns

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

Note

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

4.2.5.6 xQueueCreate() `xQueue xQueueCreate (`
              `xBase limit_ )`

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

See also

> xQueue
> xQueueDelete()
> CONFIG_QUEUE_MINIMUM_LIMIT

Parameters

limit↩_	The message limit for the queue. When this number is reach, the queue is considered full and xQueueSend() will fail. The minimum limit for queues is dependent on the setting CONFIG_QUEUE_MINIMUM_LIMIT.

Returns

xQueue A queue is returned if successful, otherwise null is returned if unsuccessful.

Warning

The message queue memory can only be freed by xQueueDelete().

4.2.5.7 xQueueDelete() `void xQueueDelete (`
`xQueue queue_ )`

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

See also

xQueueCreate()

Parameters

queue↩_	The queue to be deleted.

4.2.5.8 xQueueDropMessage() `void xQueueDropMessage (`
`xQueue queue_ )`

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

Parameters

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

4.2.5.9 xQueueGetLength() `xBase xQueueGetLength (`
`xQueue queue_ )`

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

Parameters

queue↩	The queue to return the length of.
_	

Returns

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

4.2.5.10 xQueueIsQueueEmpty()   xBase xQueueIsQueueEmpty (
  xQueue *queue_*)

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

Parameters

queue↩	The queue to determine whether it is empty.
_	

Returns

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

4.2.5.11 xQueueIsQueueFull()   xBase xQueueIsQueueFull (
  xQueue *queue_*)

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

Parameters

queue↩	The queue to determine whether it is full.
_	

Returns

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

4.2.5.12 xQueueMessagesWaiting() xBase xQueueMessagesWaiting (

xQueue *queue_*)

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

Parameters

queue↩_	The queue to determine whether one or more messages are waiting.

Returns

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

4.2.5.13 xQueuePeek() xQueueMessage xQueuePeek (

xQueue *queue_*)

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

See also

xQueueMessage

xMemFree()

Parameters

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

Returns

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

Warning

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

4.2.5.14 xQueueReceive() xQueueMessage xQueueReceive (

xQueue *queue_*)

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

See also

      xQueueMessage

      xMemFree()

Parameters

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

Returns

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

Warning

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

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

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

See also

      CONFIG_MESSAGE_VALUE_BYTES

      xQueuePeek()

      xQueueReceive()

Parameters

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

Returns

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

4.2.5.16 xSystemGetSystemInfo() xSystemInfo xSystemGetSystemInfo (
      void)

Returns

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

See also

xSystemInfo

xMemFree()

Warning

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

4.2.5.17 xSystemHalt() void xSystemHalt (
      void)

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

4.2.5.18 xTaskChangePeriod() void xTaskChangePeriod (
      xTask *task_,*
      xTime *timerPeriod_*)

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

See also

xTaskWait()

xTaskGetPeriod()

xTaskResetTimer()

Parameters

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

4.2.5.19 xTaskCreate()   xTask xTaskCreate (

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

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

Parameters

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

Returns

> xTask A pointer to the newly created task.

See also

> xTask
>
> xTaskParm
>
> xTaskDelete()
>
> xTaskState
>
> CONFIG_TASK_NAME_BYTES

Warning

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

4.2.5.20 xTaskDelete()   void xTaskDelete (

        xTask *task_*)

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

Parameters

task↩ _	A pointer to the task to be deleted.

Warning

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

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

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

See also

[xTaskRunTimeStats](#)

[xMemFree()](#)

Parameters

*tasks←*_	A variable of type xBase passed by reference which will contain the number of tasks upon return. If no tasks currently exist, this variable will not be modified.

Returns

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

Warning

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

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

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

See also

[xTaskInfo](#)

Parameters

*tasks←*_	A variable of type xBase passed by reference which will contain the number of tasks upon return. If no tasks currently exist, this variable will not be modified.

Returns

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

Warning

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

4.2.5.23 xTaskGetHandleById() `xTask xTaskGetHandleById (`
`xBase id_ )`

The xTaskGetHandleById() system call will return a pointer to the task handle specified by its identifier.

See also

xBase

Parameters

id↩_↩	The identifier of the task to return the handle pointer for.

Returns

xTask A pointer to the task handle. xTaskGetHandleById() returns null if the the task identifier cannot be found.

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

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

See also

CONFIG_TASK_NAME_BYTES

Parameters

name↩_	The ASCII name of the task to return the handle pointer for. The task name is NOT a null terminated string.

Returns

xTask A pointer to the task handle. xTaskGetHandleByName() returns null if the name cannot be found.

4.2.5.25 xTaskGetId() `xBase xTaskGetId (`
　　　　　`xTask task_ )`

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

Parameters

task↩	The task to return the identifier of.
_	

Returns

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

4.2.5.26 xTaskGetName() `char * xTaskGetName (`
　　　　　`xTask task_ )`

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

See also

CONFIG_TASK_NAME_BYTES

xMemFree()

Parameters

task↩	The task to return the name of.
_	

Returns

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

Warning

The memory allocated by xTaskGetName() must be free by xMemFree().

4.2.5.27   xTaskGetNumberOfTasks()   `xBase xTaskGetNumberOfTasks (`

`           void  )`

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

Returns

xBase The number of tasks.

4.2.5.28   xTaskGetPeriod()   `xTime xTaskGetPeriod (`

`           xTask task_ )`

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

See also

xTaskWait()

xTaskChangePeriod()

xTaskResetTimer()

Parameters

task←_	The task to return the timer period for.

Returns

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

4.2.5.29   xTaskGetSchedulerState()   `xSchedulerState xTaskGetSchedulerState (`

`           void  )`

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

See also

xSchedulerState

xTaskSuspendAll()

xTaskResumeAll()

Returns

xSchedulerState The state of the scheduler.

4.2.5.30 xTaskGetTaskInfo() xTaskInfo xTaskGetTaskInfo (
        xTask *task_*)

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

See also

> xTaskInfo

Parameters

task← _	The task to return the details of.

Returns

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

Warning

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

4.2.5.31 xTaskGetTaskRunTimeStats() xTaskRunTimeStats xTaskGetTaskRunTimeStats (
        xTask *task_*)

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

See also

> xTaskRunTimeStats
>
> xMemFree()

Parameters

task← _	The task to get the runtime statistics for.

Returns

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

Warning

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

4.2.5.32 xTaskGetTaskState() `xTaskState xTaskGetTaskState (`
> `xTask task_ )`

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

See also

> xTaskState

Parameters

task←_	The task to return the state of.

Returns

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

4.2.5.33 xTaskNotificationIsWaiting() `xBase xTaskNotificationIsWaiting (`
> `xTask task_ )`

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

Parameters

task←_	The task to check for a waiting task notification.

Returns

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

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

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

See also

CONFIG_NOTIFICATION_VALUE_BYTES

xTaskNotifyTake()

Parameters

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

Returns

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

4.2.5.35 xTaskNotifyStateClear() `void xTaskNotifyStateClear (`
      `xTask task_ )`

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

Parameters

task↩ _	The task to clear the notification for.

4.2.5.36 xTaskNotifyTake() `xTaskNotification xTaskNotifyTake (`
      `xTask task_ )`

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

See also

xTaskNotification

xTaskNotifyGive()

xMemFree()

CONFIG_NOTIFICATION_VALUE_BYTES

Parameters

task←_	The task to return a waiting task notification.

Returns

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

Warning

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

4.2.5.37 xTaskResetTimer() `void xTaskResetTimer (`
        `xTask task_ )`

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

See also

xTaskWait()

xTaskChangePeriod()

xTaskGetPeriod()

Parameters

task←_	The task to reset the task timer for.

4.2.5.38 xTaskResume() `void xTaskResume (`
        `xTask task_ )`

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

See also

xTaskState

xTaskSuspend()

xTaskWait()

Parameters

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

4.2.5.39 xTaskResumeAll() `void xTaskResumeAll (`
           `void  )`

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

See also

    xTaskSuspendAll()

4.2.5.40 xTaskStartScheduler() `void xTaskStartScheduler (`
           `void  )`

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

4.2.5.41 xTaskSuspend() `void xTaskSuspend (`
           `xTask task_ )`

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

See also

    xTaskState
    xTaskResume()
    xTaskWait()

Parameters

task←_	The task to suspend.

4.2.5.42 xTaskSuspendAll() `void xTaskSuspendAll (`
           `void  )`

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

See also

xTaskResumeAll()

4.2.5.43   xTaskWait()   `void xTaskWait (`
            `xTask task_ )`

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

See also

xTaskState

xTaskResume()

xTaskSuspend()

Parameters

task←_	The task to place in the waiting state.

4.2.5.44   xTimerChangePeriod()   `void xTimerChangePeriod (`
            `xTimer timer_,`
            `xTime timerPeriod_ )`

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

See also

xTimerHasTimerExpired()

Parameters

*timer_*	The timer to change the period for.
*timer← Period_*	The timer period in is microseconds. Timer period must be zero or greater.

4.2.5.45   xTimerCreate()   `xTimer xTimerCreate (`

xTime *timerPeriod_*)

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

See also

> xTimer
>
> xTimerDelete()

Parameters

*timer↩ Period_*	The number of microseconds before the timer expires.

Returns

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

Warning

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

4.2.5.46 xTimerDelete() `void xTimerDelete (`
   `xTimer timer_ )`

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

See also

> xTimerCreate()

Parameters

timer↩ _	The timer to be deleted.

4.2.5.47 xTimerGetPeriod() `xTime xTimerGetPeriod (`
   `xTimer timer_ )`

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

Parameters

timer↩_	The timer to get the timer period for.

Returns

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

4.2.5.48   xTimerHasTimerExpired()   xBase xTimerHasTimerExpired (
          xTimer timer_)

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

See also

xTimerReset()

Parameters

timer↩_	The timer to determine if the period has expired.

Returns

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

4.2.5.49   xTimerIsTimerActive()   xBase xTimerIsTimerActive (
          xTimer timer_)

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

See also

xTimerStart()

Parameters

timer↩_	The timer to check if active.

Returns

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

4.2.5.50 xTimerReset() `void xTimerReset (`
`        xTimer timer_ )`

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

Parameters

timer↩ *_*	The timer to be reset.

4.2.5.51 xTimerStart() `void xTimerStart (`
`        xTimer timer_ )`

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

See also

xTimerStop()

xTimerReset()

Parameters

timer↩ *_*	The timer to be started.

4.2.5.52 xTimerStop() `void xTimerStop (`
`        xTimer timer_ )`

See also

xTimerStart()

xTimerReset()

Parameters

timer↩ *_*	The timer to be stopped.

Index