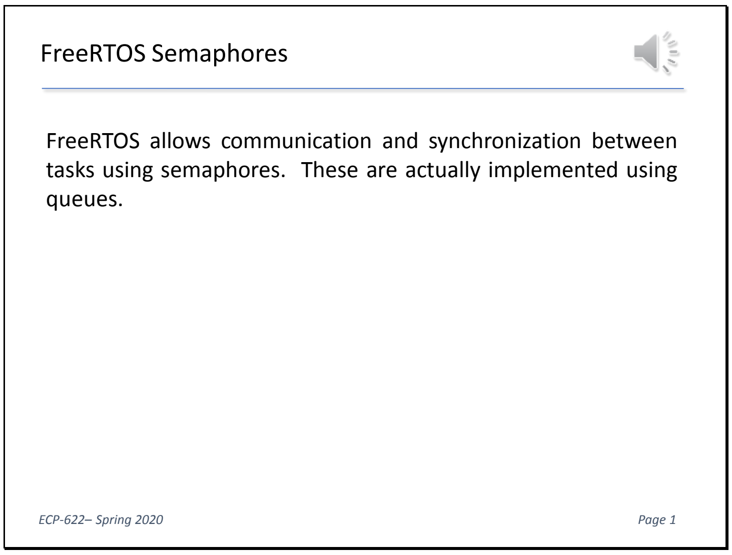
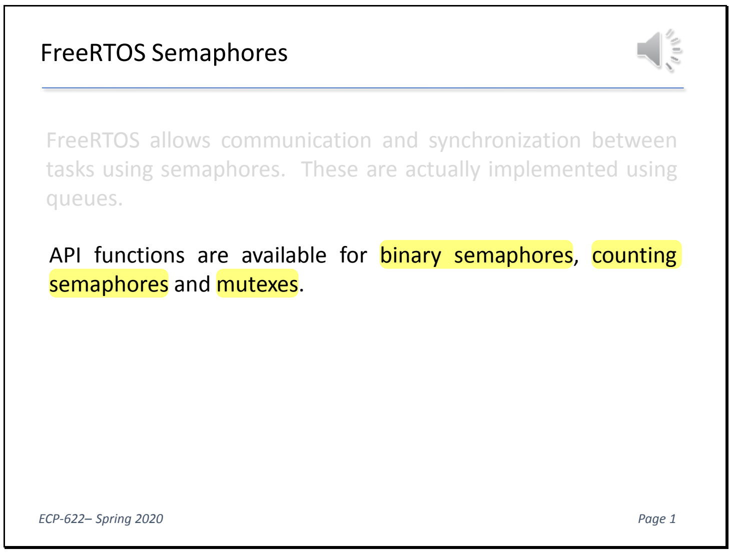

## FreeRTOS Semaphores

FreeRTOS allows communication and synchronization between tasks using semaphores. These are actually implemented using queues.

Slide 1

As all current operating systems, FreeRTOS allows tasks to use semaphores. Although these semaphores' behavior is the same as that we defined theoretically, they are internally implemented using queues. You can look at the details on the FreeRTOS web site. What is important here is to note that semaphores are treated in a manner completely different from normal integer variables.

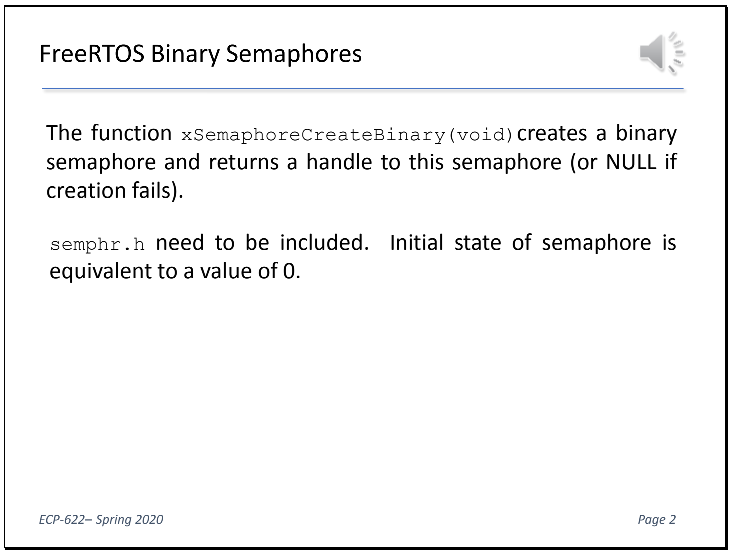

## FreeRTOS Semaphores

FreeRTOS allows communication and synchronization between tasks using semaphores. These are actually implemented using queues.

API functions are available for binary semaphores, counting semaphores and mutexes.

Slide 2

There are three types of semaphores which differ in the method of declaration and in typical uses.


## FreeRTOS Binary Semaphores

The function `xSemaphoreCreateBinary(void)` creates a binary semaphore and returns a handle to this semaphore (or NULL if creation fails).

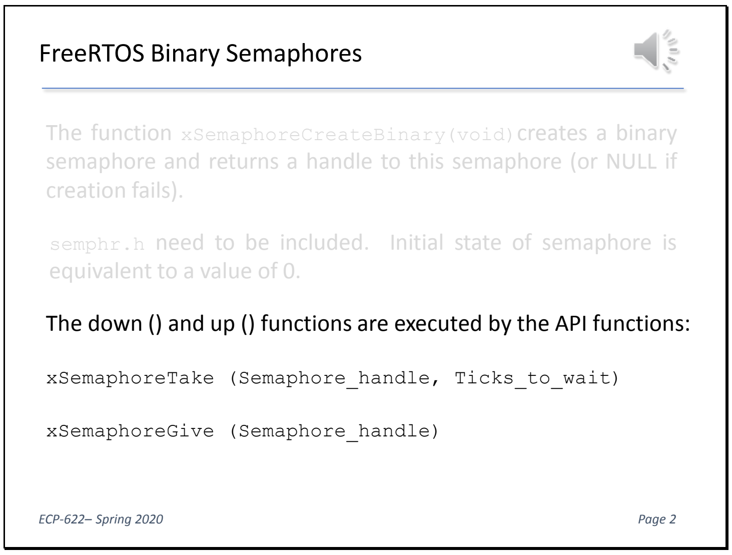
`semphr.h` need to be included. Initial state of semaphore is equivalent to a value of 0.

Slide 3

The binary semaphore - as its name implies- will only take values of 0 or 1. This type is typically used for signaling applications. Recall that the x in the start of the function name implies that it returns a word.

Slide 4

As it is mainly used for signaling, it is initially given a value of 0. Thus, it will block a task, waiting for a signal from another task (i.e. by the equivalent of the up function).


## FreeRTOS Binary Semaphores

The function `xSemaphoreCreateBinary(void)` creates a binary semaphore and returns a handle to this semaphore (or NULL if creation fails).

`semphr.h` need to be included. Initial state of semaphore is equivalent to a value of 0.

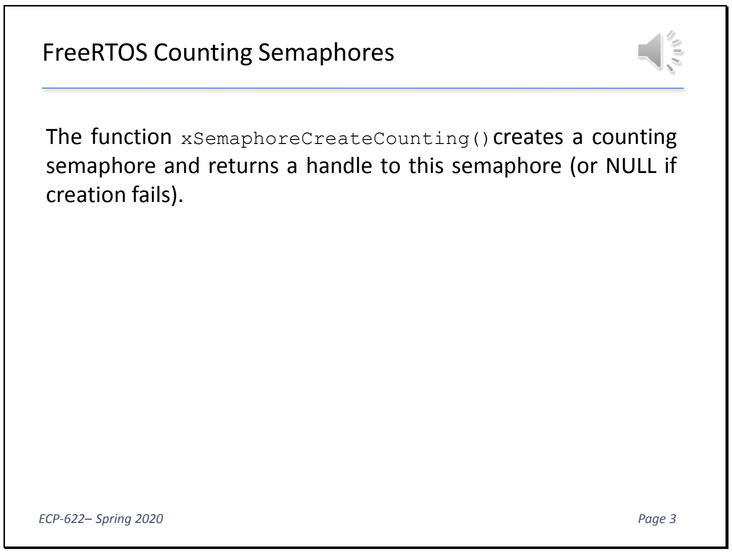
The down () and up () functions are executed by the API functions:

```
xSemaphoreTake (Semaphore_handle, Ticks_to_wait)

xSemaphoreGive (Semaphore_handle)
```

Slide 5

The functions xSemaphoreTake (equivalent to down) and xSemaphoreGive (equivalent to up) are the same for the three types of semaphores. Contrary to the theoretical definition of down, we can specify a maximum duration for blocking by xSemaphoreTake. After this duration, the functions times out and returns an error (0 value). For indefinite waiting, use portMAX_DELAY.



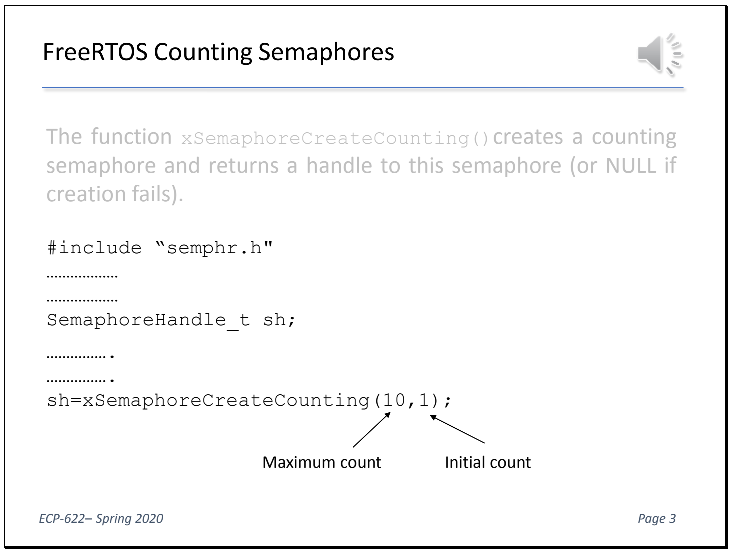
## FreeRTOS Counting Semaphores

The function `xSemaphoreCreateCounting()` creates a counting semaphore and returns a handle to this semaphore (or NULL if creation fails).

ECP-622– Spring 2020 Page 3

Slide 6

Counting semaphores are used in applications where semaphore can take positive integer values greater than one, as for example in the reader/writer problem.



## FreeRTOS Counting Semaphores

The function `xSemaphoreCreateCounting()` creates a counting semaphore and returns a handle to this semaphore (or NULL if creation fails).

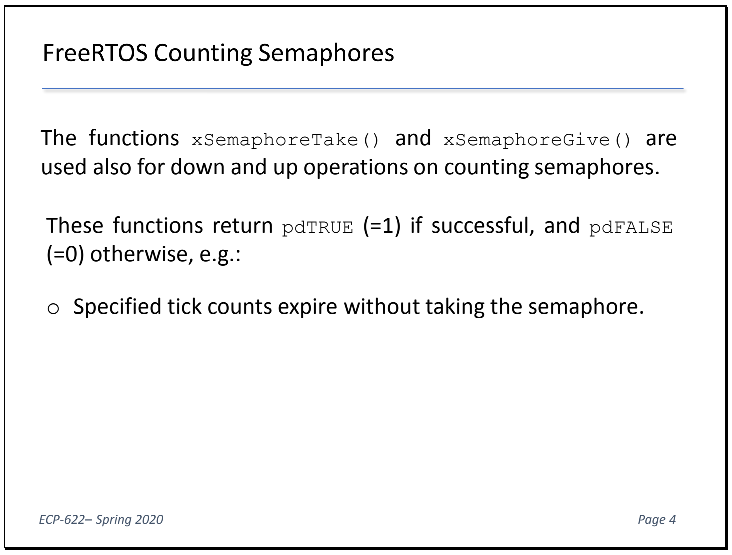
```
#include “semphr.h"
………………
………………
SemaphoreHandle_t sh;
…………….
…………….
sh=xSemaphoreCreateCounting(10,1);
```

Maximum count
Initial count

ECP-622– Spring 2020 Page 3

Slide 7

When creating a counting semaphore, we give it an arbitrary initial value as this will differ from one application to the other. One other difference from the theoretical definition is that we specify a maximum value that the semaphore cannot exceed. Note that the shown declaration of semaphore handle type is common to all types of semaphores.



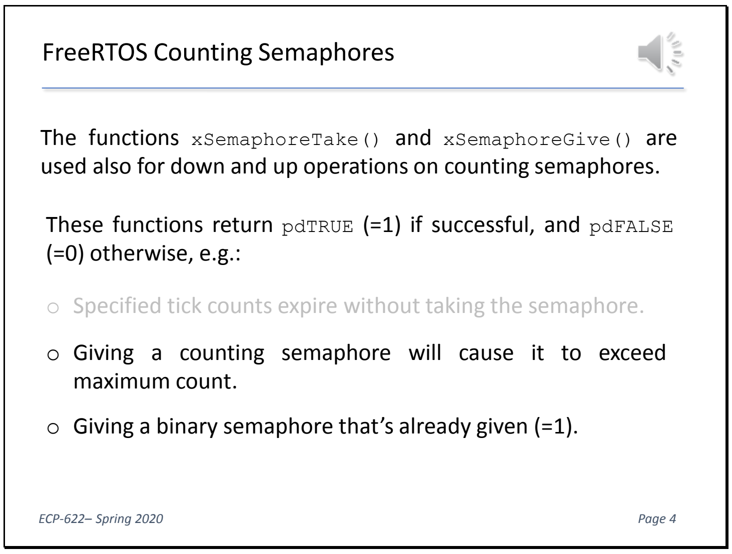
## FreeRTOS Counting Semaphores

The functions `xSemaphoreTake()` and `xSemaphoreGive()` are used also for down and up operations on counting semaphores.

These functions return `pdTRUE` (=1) if successful, and `pdFALSE` (=0) otherwise, e.g.:

- Specified tick counts expire without taking the semaphore.

ECP-622– Spring 2020 Page 4



## FreeRTOS Counting Semaphores

The functions `xSemaphoreTake()` and `xSemaphoreGive()` are used also for down and up operations on counting semaphores.

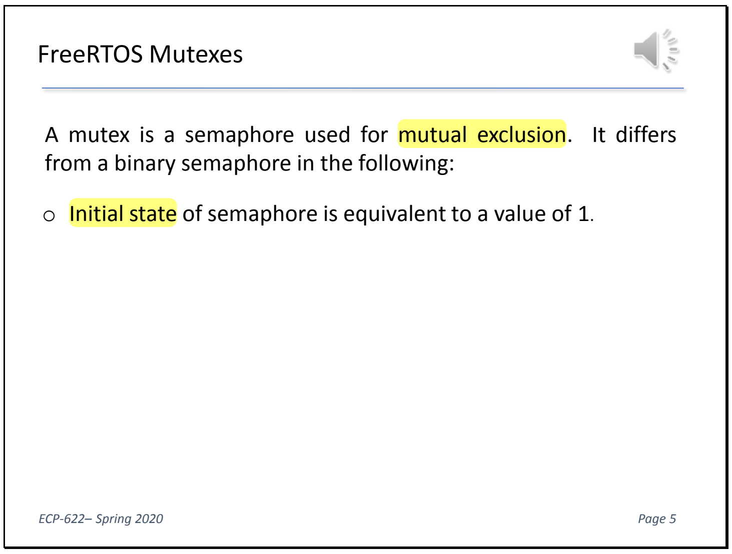
These functions return `pdTRUE` (=1) if successful, and `pdFALSE` (=0) otherwise, e.g.:

- Specified tick counts expire without taking the semaphore.
- Giving a counting semaphore will cause it to exceed maximum count.
- Giving a binary semaphore that's already given (=1).

ECP-622– Spring 2020 Page 4

Slide 10

Thus, if we try to perform the equivalent of an up operation that will cause the counting semaphore to exceed the specified maximum value, the operation fails and an error is returned. Note that the same occurs if we try to increase a binary semaphore which is already equal to 1.



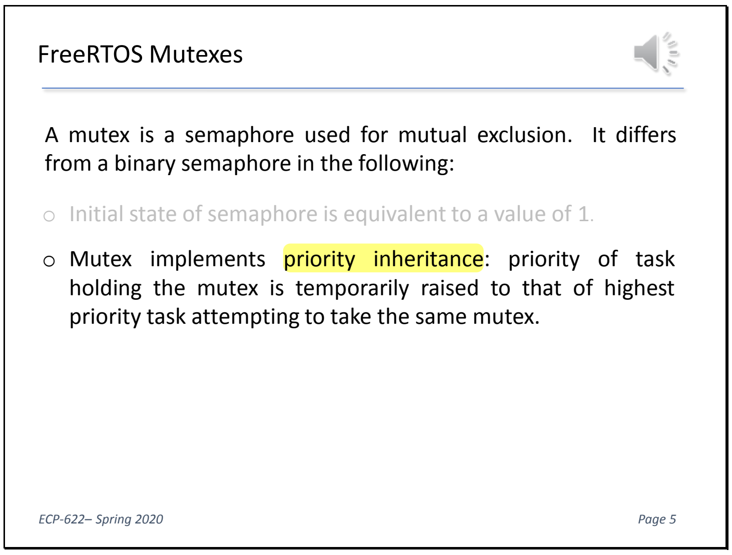
## FreeRTOS Mutexes

A mutex is a semaphore used for mutual exclusion. It differs from a binary semaphore in the following:

- Initial state of semaphore is equivalent to a value of 1.

Slide 11

The third type of semaphores, called the mutex is specifically intended for mutual exclusion. Thus, its normal initial value is 1 instead of 0 as binary semaphores.



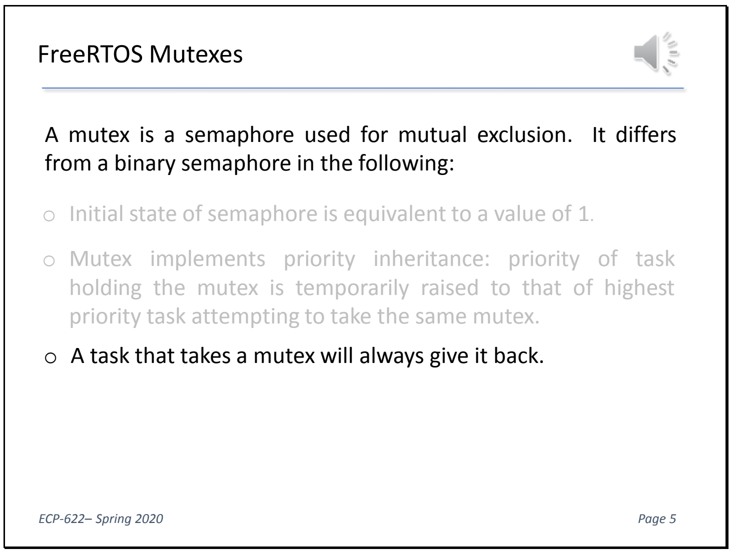
## FreeRTOS Mutexes

A mutex is a semaphore used for mutual exclusion. It differs from a binary semaphore in the following:

- Initial state of semaphore is equivalent to a value of 1.
- Mutex implements priority inheritance: priority of task holding the mutex is temporarily raised to that of highest priority task attempting to take the same mutex.

Slide 12

As we have seen before, priority inheritance is an essential technique for real-time kernels. Priority inheritance does not apply for binary or counting semaphores.



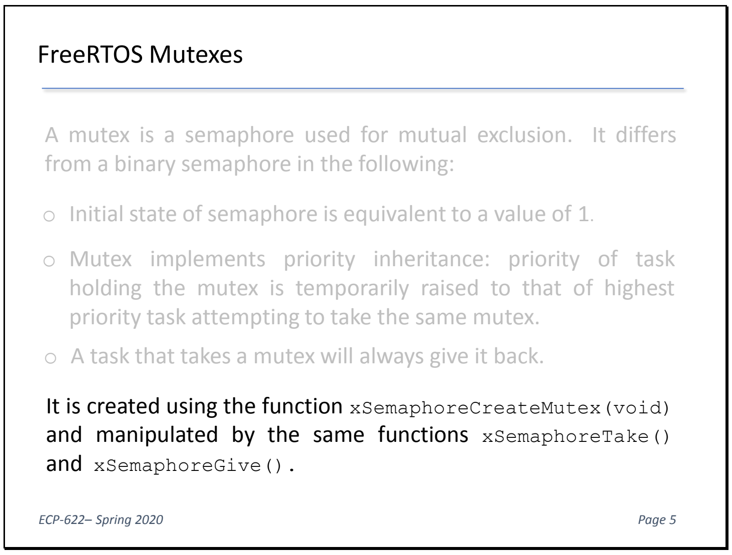
## FreeRTOS Mutexes

A mutex is a semaphore used for mutual exclusion. It differs from a binary semaphore in the following:

- Initial state of semaphore is equivalent to a value of 1.
- Mutex implements priority inheritance: priority of task holding the mutex is temporarily raised to that of highest priority task attempting to take the same mutex.
- A task that takes a mutex will always give it back.

Slide 13

Otherwise, an error will occur. So we should never use a mutex take without a mutex give.



## FreeRTOS Mutexes

A mutex is a semaphore used for mutual exclusion. It differs from a binary semaphore in the following:

- Initial state of semaphore is equivalent to a value of 1.
- Mutex implements priority inheritance: priority of task holding the mutex is temporarily raised to that of highest priority task attempting to take the same mutex.
- A task that takes a mutex will always give it back.

It is created using the function `xSemaphoreCreateMutex(void)` and manipulated by the same functions `xSemaphoreTake()` and `xSemaphoreGive()`.