software timers:

hello welcome back in the section we

shall look at software timers software

timers are used to schedule the

execution of a function at a set time in

the future or periodically with a fixed

frequency the function executed by the

software timer is called the software

timers callback function software timers

are implemented and are completely under

the control of the free artists kano

they do not require any hardware support

and are not related to Hardware timers

that come with a microcontroller these

are different these are made by the

Econo in the section we shall see the

characteristics of the software timer

compared to the normal tasks that we've

spoken about we shall also see the timer

command queue we shall also see the

difference between a one-shot software

timer and a periodic software timer and

of course we shall learn how to create

start reset and change the period of

software timers so the software timer

callback functions are implemented as

normalcy functions the only thing

special about them is they are prototype

which must return a void and take a

handle to a software timer as the only

argument that's what it looks like very

simple very straightforward and the

software time our callback functions

execute from start to finish

they must not contain infinite loops

like the tasks they have to exit in a

normal way and they should be kept very

short and must not enter into things

like the blocked state or wait on other

resources the software timers period is

the time between the software timer

being started and the software timers

callback function executing and like we

mentioned earlier there are two types of

software timers we've got the one-shot

timer and we've got the auto reload

timer once started the one-shot timer

will execute it callback function only

once one short time I can be restarted

manually but it does not restart itself

for the auto reload timers once started

an auto reload timer will restart itself

each time it expires resulting in a

periodic execution of its core

back function a software time I can

either be in a dormant state or a Runyan

state a dormant software timer exists

and can be referenced by its handle but

it's not running so its callback

functions will not execute a running

software timer will execute it callback

function after a time equal to its

period has elapsed since the software

timer entered the running State or since

the software timer was last reset this

illustration over here shows the autorai

load software timer state and transition

as we can see when the timer expires in

the auto reload timer the timer goes

back is still in the running state it

keeps on going the only way to take the

auto reload timer to the dormant state

is to use the ex timer stop API and when

the auto reload time is in the dormant

state we can use either the x time I

start X time I restart or the X time I

change period to move it from the

dormant state to the running state and

from this diagram again we can see that

when we first creates the timer it is in

the dormant state when it's only when we

use these other three api's the start

the reset and change period is only when

we do this then we get the time has

moved from the dormant to the run in the

state now let's take a look at the one

short timers diagram so from here we can

see the one short timer as soon as the

timer has expired it moves to the

dormant state it doesn't go to the

running state contrast what we just saw

with this one when the timer expires

after it's expired it still goes on to

the running state executing periodically

grass in the in the one shot over here

when the timer expires it goes to the

dormant state and we can manually put it

in the dormant state by still using the

x timer stop API and over here just like

what we saw previously when we create

the timer it's in the dormant state and

we move it from the dormant state to the

running States by using the either start

reset or change period functions but a

key difference between the two is to

notice that over here when the timer

expires it goes

the Dumont state whereas in the auto

reload setting when the timer expires is

still remains in the running state and

continues its execution periodically

so this order is for this lesson if you

have any questions send me a message or

leave in the questions area and I'll see

you

Semaphore

Hello welcome back. In this section we shall look at semaphores. In this very first lesson we shall

take a look at binary semaphores and binary semaphores can be used for different things but most often

at times they are used for synchronization.

So in this particular section what we're going to be talking about is how to use binary semaphores

for synchronizing.

Especially we can use a binary semaphore to synchronize an interrupt service routine in a task. A binary

semaphore can be used to unblock a task each time a particular interrupt occurs effectively synchronizing

the task with the interrupt.

This allows the majority of the interrupt event processing to be implemented within the synchronized task

with only a very fast and short portion remaining directly in the interrupt service routine.

And this is what is known as defered interrupts process ing. The processing is defered to a task because when

we are using interrupt sercice routines we want them to be very short and fast as possible because if we

spend so much time in the interrupt service routine it would affect our real time kernel

So what we want is just receive the interrupt and send a quick notification to a task to perform the

processing

But we mostly use binary semaphores to achieve defered interrupt processing also we can use binary

semaphores to synchronize tasks.

We can make any two tasks or even three tasks synchronize and run at the same time that we can also use

them to make just a portion of the code to execute at the same time.

We can place semaphores at specific portions of different parts of the code and make them all meet at

the same point and execute at the same time.

We shall see all of this in an actual example rather than talk about it.

So what I'm going to do is rather than describe all the things the semaphores can be used for.

We'll quickly go through the semaphor APIs and go straight to coding to write some actual semaphore code.

So let's get to it.

Let's start with his very first one. To create a semahpre we use the xsemaphorecreatebinary.

And this is what we use to create a binary semaphore. Later on we shall talk about another semaphore

known as the counting semaphore, that is why this function is called

xSemaphorescreatebinary because we use this to create a binary semaphore only. This function takes

no argument,

however it returns a semaphore handle and if it returns a semaphore handle we can

use this handle to reference the semaphore that we created. It can also return a null value. When

a null is returned

this means there isn't enough heap memory available for freeRTOS to allocate the semaphore data

structures. The next semaphore API we shall look at is the xsemaphoretake and this is basically used

for taing a semaphore and to take a semaphore means to obtain or receive a semaphore. And a semaphore can be taken only

if it's available.

So you can have two tasks with a single semaphore.

If one task takes the semaphores.

There's no longer there is no semaphore available for other tasks to take.

So a semaphores can only be taken if it's available.

And this function takes two arguments.

The first argument is the handle of the semaphores, the handle of the semaphore we wish to take and the

second argument,

By now you are familiar with that argument, it is the X ticks to wait, and this argument basically specifies

the maximum amount of time the task should remain in the block state to wait for the semaphore.

If it's not available to to wait for it to become available. We can also give a semaphore. Once a particular

task no longer needs a semaphore, we can give it back and we use the xsemaphoregive function and this

function takes just one argument and this argument.

is the handle of the semaphore that is being given back. So you can think of it this way.

One task takes a semaphore. The semaphore is no longer available but after that task gives back

the semaphore, the semaphore becomes available for another task to take.

So this is the one we use for tasks.

But if we are dealing with interrupt service routines we have to use the function

xsemaphoresgivegivefromISR

,

And this one here takes two arguments.

The first argument is similar to the first one we saw which is the handle of the semaphores and the

second argument over here is called the HigherPriorityTaskWoken