Code Exercise-7: Implementing delay using vTAskDelayUntil()

In this article I will explain the code exercise -7 which demonstrates how to use the **vTAskDelayUntil()**.

The source code of this example is already included in the folder which you might have downloaded already.

From the previous lectures on **vTaskDelay()** and **vTAskDelayUntil()** you already know the differences between these two.

vTaskDelay() vs vTAskDelayUntil()

1. **vTaskDelay()** specifies a time at which the task wishes to unblock relative to the time at which **vTaskDelay()** is called

2. **vTAskDelayUntil()** specifies an absolute time at which the task wishes to unblock.[Time is calculated relative to the last wake up time of the task

3. **vTaskDelay()** doesn't ensure fixed execution frequency of a task whereas **vTAskDelayUntil()** ensures it

Alright now, go to Example 7 and open the **main.c**

Resources_RTOS\Source_codes\Atmel\Example007\Example007\Example007\src

In this example we are going to implement blocking delay using the API **vTaskDElayuntill().**

The main() function implements 2 tasks , Task 1 and Task 2 and both having the common task function **vTaskFunction**.

```
/* Create the first task at priority 1... */
xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

/* ... and the second task at priority 2. The priority is the second to
last parameter. */
xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );
```

The priority of Task 2 is higher than the priority of Task 1.

So, when you call, **vTaskStartScheduler();**, the Task 2 runs first.

Great!

now **vTaskFunction** will be executed by the Task 2 first.

In the **vTaskFunction** , we just extract the string which is passed to this task function.

```
/* The string to print out is passed in via the parameter. Cast this to a
character pointer. */
pcTaskName = ( char * ) pvParameters;
```

then, we get the current tick count value by using the below line of code .

```
/* The xLastWakeTime variable needs to be initialized with the current tick
count. Note that this is the only time we access this variable. From this
point on xLastWakeTime is managed automatically by the vTAskDelayUntil()
API function. */
xLastWakeTime = xTaskGetTickCount();
```

here **xTaskGetTickCount()** is a freeRTOS API defined in tasks.c , which returns the current kernel tick count value.

The kernel tick count value is stored in **xTickCount**, which is a global variable.

Suppose xTickCount = 1000, that means, 1000 times kernel tick interrupts have occurred after you call the function **vTaskStartScheduler().**

Alright, so **xLastWakeTime** will now have the current tick count value.

So, as soon as you come in to the task function, you first saved the current tick count value in to **xLastWakeTime.**

Great!

Now, the task function enters in to the infinite *for* loop and it tries to print out the string using **printf**.

Now, at this point let's assume that, our task gets pre-empted by some higher priority tasks like Task 3. Even though in this example i have not used Task 3 , but just assume that Task 3 preempts our task right at the below code.

```
/* Print out the name of this task. */
printf( pcTaskName );
```

Now, when the Task 2 resumes some time later, it will call the **vTaskDelayUntil** function right ?

```
/* We want this task to execute exactly every 250 milliseconds.  As per
the vTaskDelay() function, time is measured in ticks, and the
portTICK_RATE_MS constant is used to convert this to milliseconds.
xLastWakeTime is automatically updated within vTaskDelayUntil() so does not
have to be updated by this task code. */
```

```
vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
```

But the 250 ticks of delay is not calculated relative to the current tick count value of the kernel but instead calculated relative to the last wake up time of the task that is the value which is stored in the variable **xLastWakeTime** , which actually remembers the time prior to pre-emption of the task 2.

So, Task-2 now be blocked relative to the **xLastWakeTime**, which ensures the fixed periodicity.

If you want to check, go in to the function **vTAskDelayUntil()** and you will see the line below.

```
/* Generate the tick time at which the task wants to wake. */
xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;
```

the **xTimeToWake** is calculated by adding **pxPreviousWakeTime** and **xTimeIncrement**.

So, the task wakes up after **xTimeToWake**;

Great, so, that's the procedure to use **vTaskDelayuntil(),** only thing you need to keep in mind that , when you enter in to the task function before entering in to the infinite '*for*' loop, you have to take the snap shot of the current tick count value and save it in some variable.

The same applies for when Task 1 executes.

Assignment!
Great, now what you have to do is, compile the code and download it on to your hardware. Then analyse the output by getting the tracealyzer output. if you need any help then please feel free to ask me .

See you in the next lecture.

Thanks