FREE RTOS

# INTRO

The goal of this document is to present the e-paper display (EPD) driver and main setting. After recapping the main device characteristics, the MCU finite-state machine (FSM) in charge to manage the display is explained. Finally it is proposed a sketch of how the EPD is subdivided for this specific project and the meaning of each field.

# SETUP

1. Hardware : PXS board (including dsPIC33CH512MP508 MCU) has been used;
2. Software : MPLAB v5.45 + XC v1.61.
3. Demo projects : [code](https://github.com/microchip-pic-avr-examples/pic24-dspic33-freertos-demo) + [explanation](https://www.freertos.org/portpic24_dspic.html#DemoApp);
4. Upgrade : FreeRTOS+ contains add-on libraries for more advanced features (such as TCP-IP stack), whereas the original FreeRTOS contains the kernel and the core functionalities.

# PROJECT TIPS

* For new projects, remember to change MCU model in *Properties > Conf*;
* In case there is need to move/rename folders inside the main folder (e.g. Ex1) remember to report the changes also in file */nbproject/configurations.xml* (see especially *ItemPath*, *extra-include-directories* and *extra-include-directories-for-preprocessor* fields) and in *MPLAB > Properties > General > Source Folders*.
* Keep in mind compiler optimization levels > 0 can create issues to the expected code operation;
* In case of code misbehavior try to optimize the stack sizes of modules (e.g. MAX\_STACK\_SIZE in case of tasks);
* Each time you want to add a new library (e.g. Timers) to the project, remember to add the corresponding .h/.c files to the project (as "add existing item"), and to enable the dedicated macro in *FreeConfigRTOS.h* (e.g. *configUSE\_TIMERS*);
* For custom user-configuration, modify the file *FreeRTOSConfig.h* only (e.g. adding macros to enable features), and NEVER the file *FreeRTOS.h*;
* See useful DigiKey lessons about FreeRTOS [here](https://www.youtube.com/watch?v=Qske3yZRW5I&list=PLZIv8_XuNVRYQwCMXfBwaqnqVrN6OZViq&index=16&t=2s&ab_channel=Digi-Key);
* The tick-timing depends on both the MCU Fcy (90 MHz here) and the time-constraints of the tasks, however it is recommended to never use a tick-timing faster than 1 kHz;
* Typical FreeRTOS syntax :
  + "v" at the beginning of a functions name means the return type is void;
  + "pv" at the beginning of a functions name means the return type is void-pointer.

# INTRO

FreeRTOS offers several advantages :

* It is free (though a paid versions allow for some protection);
* It is portable (as it abstracts well from the hardware it runs on);
* It provides an easy way to separate sections of code (through tasks definition);
* It allows for control of when, and how often, tasks are run.

The core of the OS is the scheduler. As part of the operating system, the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program. Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. In FreeRTOS, a thread of execution is called a ***task***. FreeRTOS therefore provides the core real time scheduling functionality, inter-task communication, timing and synchronization primitives only. This means it is more accurately described as a real time kernel, or real time executive. Additional functionality, such as a command console interface, or networking stacks, can then be included with add-on components.

# MULTITASKING

Each executing program is a *task* (or *thread*) under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be *multitasking*. The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application :

* The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks;
* The partitioning can result in easier software testing, work breakdown within teams, and code reuse;
* Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system;
* A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently.

# SCHEDULING

The scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime. The scheduling policy is the algorithm used by the scheduler to decide which task to execute at any point in time. Typically priority preemptive mechanism is followed by the scheduler to decide which task will run during the new tick period (in case of tasks ready with the same priority a Round-Robin policy is followed to alternate their execution).

When sleeping, an RTOS task will specify a time after which it requires 'waking'. When blocking, an RTOS task can specify a maximum time it wishes to wait. The FreeRTOS real time kernel measures time using a tick count variable. A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy - allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency.

Each time the tick count is incremented the real time kernel must check to see if it is now time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly woken/unblocked task. A context switch occurring in this way is said to be "preemptive", as the interrupted task is preempted without suspending itself voluntarily.

In addition to being suspended involuntarily by the kernel (in particular, the scheduler) a task can choose to suspend itself. It will do this if it either wants to delay (sleep) for a fixed period, or wait (block) for a resource to become available (eg a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

Note a task can require multiple consecutive tick-periods to execute his operations (but depending on the scheduler decisions, these periods might not be consecutive).

**Note that during each tick period only up to 1 user-created task can run** :

* If a higher priority task becomes ready, the next tick will be reserved to this;
* If no tasks are ready when new tick period starts, the idle task will run for the entire tick period;
* If in the middle of a tick period a task suspends itself, the execution is passed immediately to the idle task until the next tick interrupt (see “IDLE TASK” section above), and NOT to the scheduler for starting other ready tasks immediately (since the scheduler is called ONLY when the tick-timeout expires). Here if all tasks are still suspended, the idle task keeps running until next tick ISR; otherwise the highest-priority ready task is executed;
* If multiple tasks with the same priority are ready when tick-timer expires, the scheduler alternates their execution among successive tick-slots following a Round-Robin policy. This can be disabled by setting *configure\_TIME\_SLICING* to 0: this way the first task will run for as many tick-periods it needs without being interrupted by other same-priority tasks and so on (but preemption for higher-priority tasks remain).

# CONTEXT-SWITCHING

While a task is in execution, it utilizes the MCU registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution context. A task is a sequential piece of code, and it does not know if/when it is going to get suspended (swapped out or switched out) or resumed (swapped in or switched in) by the kernel and does not even know if/when this has already happened. It is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

# QUEUES

Queues are the primary form of inter-task communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to its front.

Moreover, queue API functions permit a block time to be specified. When a task attempts to read from an empty queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either data becomes available on the queue, or the block time expires. On the other hand, when a task attempts to write to a full queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either space becomes available in the queue, or the block time expires.

# IDLE TASK

Whenever somewhen during a tick period a task suspends itself, the execution is passed immediately to the idle task until the next tick interrupt. The idle task is created automatically by FREERTOS inside the initial function *vTaskStartScheduler* and it is the only task running with no tick-timing. When no tasks are available for the CPU then this task will run, executing some garbage collector activities. For example, when a task is deleted, the memory used by it will not be free immediately, but within the next idle task cycle. The idle task can be customized by setting the macro *configUSE\_IDLE\_HOOK* to 1 and then adding your code in the function *vApplicationIdleHook*. Typical usage of this function is to add low-priority CPU monitoring/log functions or to place the MUC in some power-saving mode.

The fact the idle task is not subject to tick-timing (unlike any other user-created task) can be checked by placing the *Toggle\_GPIO* function inside the function *vApplicationIdleHook* : assuming to use a tick-timing of 1 ms, the signal toggling rate observed through oscilloscope is way faster (about 300 kHz) and especially remains constant even if the tick-timing is changes (e.g. to 4 ms).

# TO-DO

* Check if with level-2 optimization the project still works.