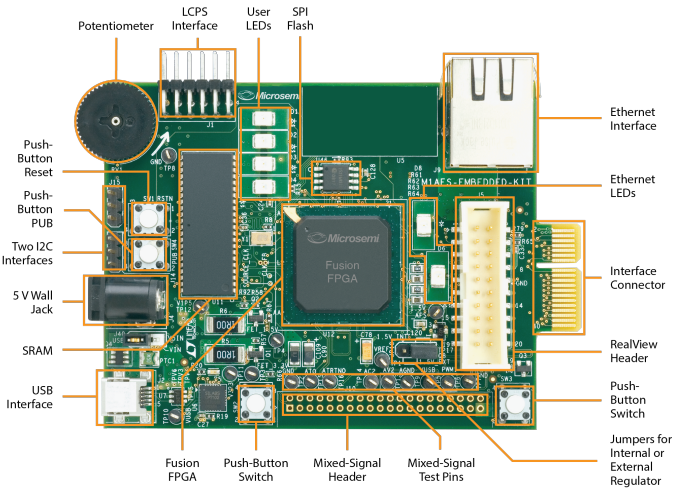

Introduction to FreeRTOS

Kizito NKURIKIYEYEZU, Ph.D.

What is FreeRTOS?

- FreeRTOS is an open source real-time operating system (RTOS) for embedded systems.
- FreeRTOS supports many different architectures and compiler toolchains, and is designed to be small, simple, and easy to use.
- It provides many features: neat and readable source code, it is portable, scalable, provides preemptive and co-operative scheduling, has multitasking and interrupt management

What is FreeRTOS?

- An open source, real-time operating system for small, low-power edge devices easy to program, deploy, secure, connect, and manage.
- Includes libraries for connecting to Amazon Web Services (AWS) web services and other edge devices running AWS IoT Greengrass.
- FreeRTOS is built with an emphasis on reliability and ease of use, and offers the predictability of long term support releases.

What is FreeRTOS?

- FreeRTOS is the most popular RTOS for small embedded MCU[1]
- Originally developed by Richard Barry around 2003
- Since 2017 it is developed and maintained by Amazon Web Services.
- FreeRTOS is a real-time kernel targeting at hard real-time applications.
- Primarily written in C with few assembler functions
- FreeRTOS has a very small footprint[2,3]
- It supports many compilers (CodeWarrior, GCC, IAR, etc.) as well as many processor architectures (ARM7, various PICs, 8051, x86, etc.).

[1] The most recent Embedded Markets Study shows that Embedded Linux and FreeRTOS continue to outpace other operating systems used in embedded development.

[2] The FreeRTOS scheduler requires 236 bytes of RAM and about 5 to 10 KBytes of ROM. Task creation require an additional 64 bytes of stack. See more at https://www.freertos.org/FAQMem.html

[3] FreeRTOS is a relatively small application. The minimum core of FreeRTOS is only three source (.c) files and a handful of header files, totalling just under 9000 lines of code, including comments and blank lines. A typical binary code image is less than 10KB.

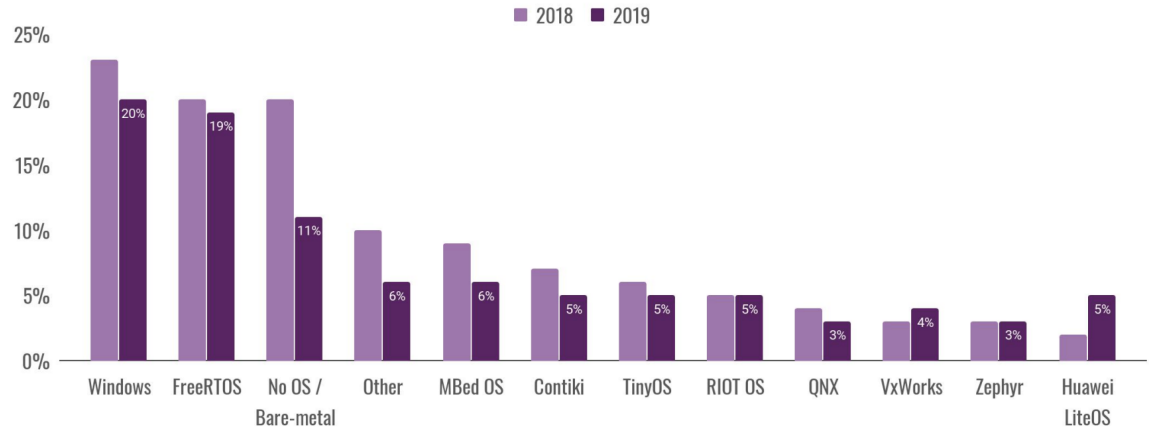

FIG 1. Most popular non-linux OS for IoT development[4], [5]

[4] https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf
[5] https://www.andplus.com/blog/which-operating-system-should-you-use-for-your-iot-solution

FreeRTOS Architecture overview

FreeRTOS's code breaks down into three main areas: tasks, communication, and hardware interfacing.

- tasks.c[6] and task.h[7] allow creating, scheduling, and maintaining tasks.
- about 40% of FreeRTOS's core code deals with communication. queue.c[8] and queue.h[9] handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes.
- The Hardware Whisperer—Much code in FreeRTOS kernel is hardware-independent. About 6% of FreeRTOS's core code acts a shim between the hardware-independent FreeRTOS core and the hardware-dependent code as shown in Figure 2.

[6] https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/tasks.c
[7] https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/include/task.h
[8] https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/queue.c
[9] https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/include/queue.h

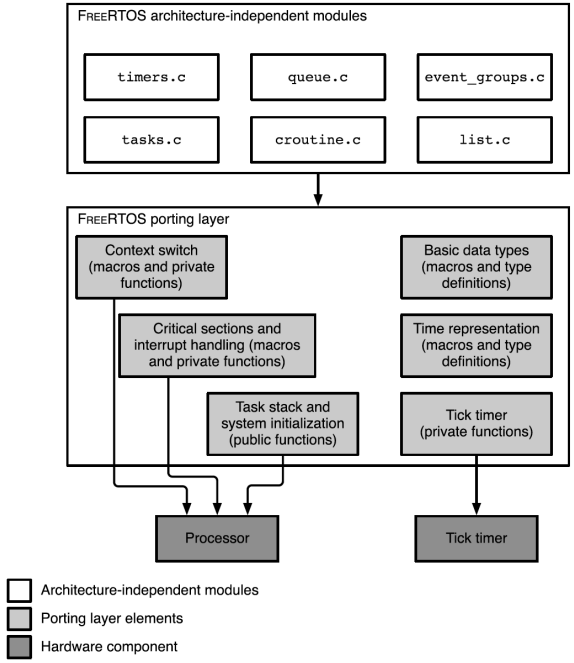

FIG 2. Summary of the FREERTOS porting layer

Task Management

Tasks in FreeRTOS

- With FreeRTOS, application can be structured as a set of autonomous tasks
- Each task executes within its own context (e.g., stack) with no coincidental dependency on other tasks
- The scheduler starts, stops, swaps in, and swaps out tasks as needed
- Each task has a user-assigned priority
  - low priority —tskIDLE_PRIORITY or idle task priority. It's equal to zero
  - configMAX_PRIORITIES-1 is the highest priority[10]
- It uses a "ready list" to keep track of all tasks that are currently ready to run.

```
1   static xList pxReadyTasksLists[configMAX_PRIORITIES];
```

  - pxReadyTasksLists[0] is a list of all ready priority 0 tasks
  - pxReadyTasksLists[1] is a list of all ready priority 1 tasks, and so on,
  - pxReadyTasksLists[configMAX_PRIORITIES-1] is the highest priority.

[10] configMAX_PRIORITIES is defined within FreeRTOSConfig.h. FreeRTOSConfig.h is application-specific and allows to customize the functionality of the RTOS kernel to the application being built. See https://www.freertos.org/a00110.html

The System Tick

- System tick—user-configurable heartbeat of a FreeRTOS system
- FreeRTOS configures the system to generate a periodic tick interrupt.
- User configurable but typically in the millisecond range.
- On each tick interrupt, the vTaskSwitchContext() function is called and selects the highest-priority ready task and puts it in the pxCurrentTCB variable

```
1 //Find the highest-priority queue that contains ready tasks.
2 while(listLIST_IS_EMPTY(&(pxReadyTasksLists[
     uxTopReadyPriority])))){
3   configASSERT( uxTopReadyPriority );
4   --uxTopReadyPriority;
5 }
6 //Make sure that tasks of the same  priority get an equal
     share of the processor time.
7 listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &(
     pxReadyTasksLists[ uxTopReadyPriority ]));
```

Tasks and task states

- A task is functions in RTOS
- A task can have the following states:
  - Running—the task is using microprocessor to execute instructions
  - Ready —the task has instructions for microprocessor to execute, but is not yet executing, perhaps because another higher priority task is running
  - Blocked —the task has nothing for microprocessor, waiting for external event (e.g., a button press) to occur for it to run again

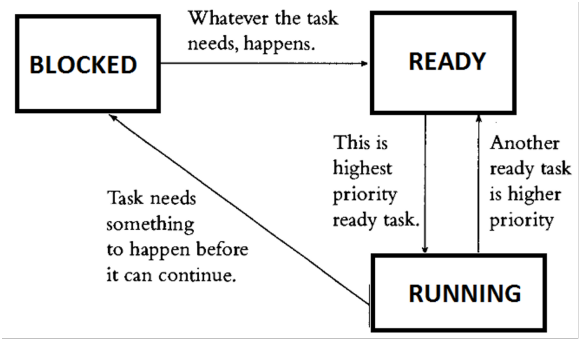

FIG 3. Common RTOS task states

Tasks and task states

- An application can consist of many tasks
- Only one task of the application can be executed at any given time on the microcontroller (single core)
- Thus, a task can exist in one of two states: Running or Not Running
- Only the scheduler can decide which task should enter the Running state
- A task is said to have been switched in or swapped in when transitioned from the Not Running to the Running state
- A task is said to have been switched out or swapped out when transitioned from the running state to the not-running state

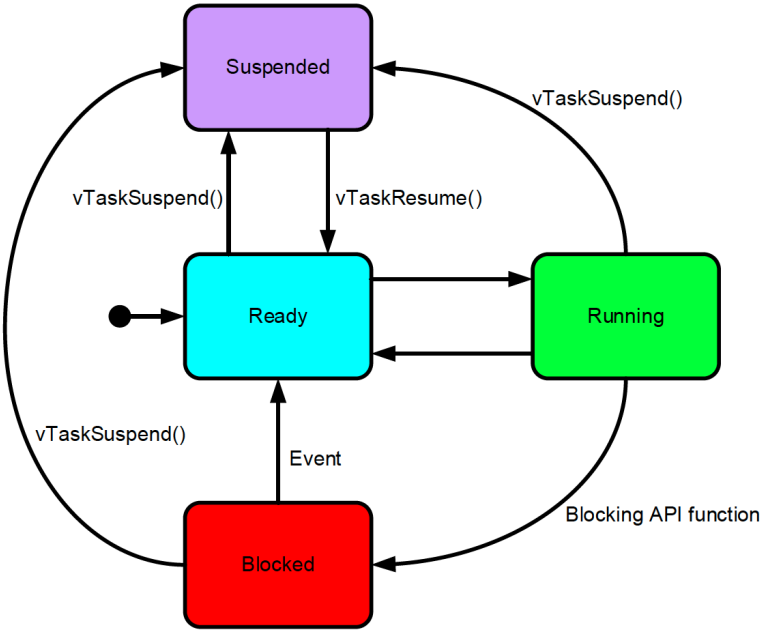

FIG 4. FreeRTOS tasks and task states

Tasks and task states

- New tasks are placed immediately in the Ready state
- The Ready list is arranged in order of priority with tasks of equal priority being serviced on a round-robin basis.
- The implementation of FreeRTOS actually uses multiple Ready lists – one at each priority level (Figure 5)

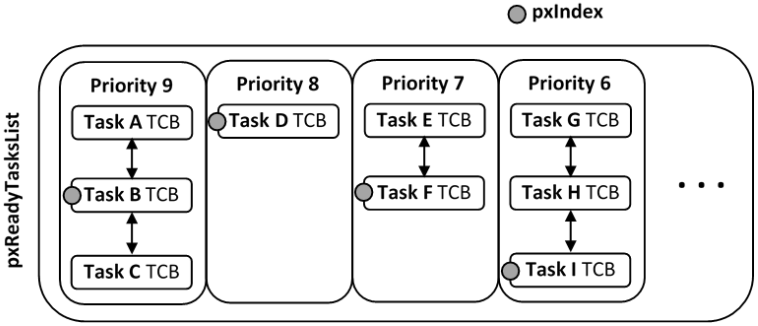

FIG 5. Illustration of FreeRTOS Ready Task List Data Structure

Tasks Functions

- Tasks are implemented as C functions
- The prototype of a task function must return void and take a void pointer parameter
- A task will typically execute indefinitely in an infinite loop: must never terminate by attempting to return to its caller
- If required, a task can delete itself prior to reaching the function end
- A single task function definition can be used to create any number of tasks
  - Each created task is a separate execution instance
  - Each created task has its own stack
  - Each created task has its own copy of any automatic variables defined within the task itself

Tasks Functions

- Tasks are defined as a functions that return void and takes a void pointer as its only parameter.
- The parameter can be used to pass information of any type into the task
- Task functions should never return so are typically implemented as a continuous loop.
- Attempting to do so will result in an configASSERT() being called if it is defined.
- Call vTaskDelete(NULL) to ensure its exit is clean.

```
void vTaskFunction(void *pvParameters ){
  while (true)
  {
    -- Task application code here. --
  }
  //In normal conditions, this code should never execute.
    vTaskDelete( NULL );
  }
```

LISTING 1: Skeleton of a FreeRTOS task

Create a Task

```
BaseType_t xTaskCreate(
  /* Function that implements the task. */
  TaskFunction_t pvTaskCode,
  /* Text name for the task. */
  const char * const pcName,
  /* Stack size in words, not bytes. */
  configSTACK_DEPTH_TYPE usStackDepth,
  /* Parameter passed into the task. */
  void *pvParameters,
  /* Priority at which the task is created. */
  UBaseType_t uxPriority,
  /* Used to pass out the created task's handle. */
  TaskHandle_t *pxCreatedTask
);
```

LISTING 2: Excerpt of the FreeRTOS task control block

TAB 1. Parameters of the FreeRTOS's xTaskCreate function

pvTaskCode	Pointer to the task entry function (just the name of the function that implements the task).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle.The maximum length of a task's name is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
usStackDepth	The number of words (not bytes!) to allocate for use as the task's stack. For example, if the stack is 16-bits wide and usStackDepth is 100, then 200 bytes will be allocated for use as the task's stack.
pvParameters	A value that is passed as the paramater to the created task. If pvParameters is set to the address of a variable then the variable must still exist when the created task executes - so it is not valid to pass the address of a stack variable.
uxPriority	The priority at which the created task will execute. Priorities are asserted to be less than configMAX_PRIORITIES. If configASSERT is undefined, priorities are silently capped at (configMAX_PRIORITIES - 1).
pxCreatedTask	Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.

Return values

- pdTRUE—when the task was created successfully
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY—the task could not be created because there was insufficient heap memory

Not Running State

- Blocked state—A task that is waiting for an event
- Tasks can enter the blocked state to wait for two different types of event
  - Time related events where a delay period expiring or an absolute time being reached. For example, when the vTaskDelay()[11] function is called
  - Synchronization events where the events originate from another task or interrupt. For example, wait for data to arrive on a queue
- It is possible for a task to block on a synchronization event with a timeout .For example, wait for a maximum of 10 ms for data to arrive on a queue
- The Suspended state: tasks in the Suspended state are not available to the scheduler.
  - A task can be suspended through a call to the vTaskSuspend()[12]API function
  - A suspended task can be resumed via a call to the vTaskResume()[13] or xTaskResumeFromISR()[14] API functions

[11] https://www.freertos.org/a00127.html
[12] https://www.freertos.org/a00130.html
[13] https://www.freertos.org/a00131.html
[14] https://www.freertos.org/taskresumefromisr.html

Not Running State

- Ready State—tasks that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state. Tasks are able to run, and therefore “ready” to run, but not currently in the Running state
- When a task is first created and made ready to run, the kernel puts it into the ready state. In this state, the task actively competes with all other ready tasks for the processor’s execution time.
- Ready tasks can only move to the running state. Because many tasks might be in the ready state, the kernel’s scheduler uses the priority of each task to determine which task to move to the running state.
- Tasks in the ready state cannot move directly to the blocked state. They first need to run so they can make a blocking call
- Because many tasks might be in the ready state, the kernel’s scheduler uses the priority of each task to determine which task to move to the running state.

Time & delay

- Functions like the Arduino delay()[15] generate delay using a null loop and poll a counter until it reaches a fixed value. This waste processor cycles

```
1  void delay(unsigned long ms){
2    uint16_t start = (uint16_t)micros();
3    while (ms > 0){
4      yield();
5      if (((uint16_t)micros() - start) >= 1000){
6        ms--;
7        start += 1000;
8      }
9    }
10 }
```

- Such functions are called blocking delays because they block code execution until they returns—which means nothing else can run while they are running.
- Non-blocking delays provide a way to use time delays without blocking the processor, so it can do other things while the functions create the delay.

[15] https://github.com/arduino/ArduinoCore-samd/blob/master/cores/arduino/delay.c

Time & delay

- FreeRTOS's vTaskDelay()[16] API function is non-blocking
  - vTaskDelay() places the calling task into the Blocked state for a fixed number of tick interrupts
  - The task in the Blocked state will not use any processing time at all

```
1 void vTaskDelay( const TickType_t xTicksToDelay );
```

  - xTicksToDelay is the number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state.
  - The constant portTICK_RATE_MS stores the time in milliseconds of a tick, which can be used to convert milliseconds into ticks.

```
1 // delay for 500ms.
2 vTaskDelay(500/portTICK_RATE_MS);
```

- An alternative method is to use the vTaskDelayUntil()[17] API function

[16] https://www.freertos.org/a00127.html
[17] https://www.freertos.org/vtaskdelayuntil.html

Time & delay

- The length of a tick depends on the operating system configuration and, to some extent, on hardware capabilities.
- The configuration macro configTICK_RATE_HZ represents the configured tick frequency in Hertz and may be useful to convert back and forth between the usual time measurement units and clock ticks.
- For instance, the quantity 1000000/configTICK_RATE_HZ is the tick length, approximated as an integer number of microsecond.
- The function TickType_t xTaskGetTickCount(void) returns the current time as the number of ticks elapsed since the operating system scheduler was started.

TAB 2. Time-Related Primitives of FREERTOS

Function	Purpose	Optional
`xTaskGetTickCount`	Return current time, in ticks	-
`xTaskGetTickCountFromISR`	Return current time, in ticks, to an ISR	-
`vTaskDelay`	Relative time delay	*
`vTaskDelayUntil`	Absolute time delay	*

vTaskDelayUntil() API[18]

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,const
   TickType_t xTimeIncrement );
```

- pxPreviousWakeTime—Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil().
- xTimeIncrement—The cycle time period. The task will be unblocked at time (*pxPreviousWakeTime + xTimeIncrement). Calling vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interval period.

[18] INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the RTOS Configuration documentation for more information.

vTaskDelayUntil() API

```
void vTaskBlinkLed(void* pvParameters){
  // set pin PB0 of PORTB for output
  DDRB |= (1<<PB0);
  TickType_t xLastWakeTime = xTaskGetTickCount();
  while (true) {
    // Toggle the 1st bit on PORT B (i.e. PB0)
    PORTB ^= (1<<PB0);
    vTaskDelayUntil(&xLastWakeTime, (1000/ portTICK_PERIOD_MS));
  }
  vTaskDelete(NULL);
}
```

LISTING 3: Example of using the vTaskDelayUntil() and blink an LED every 500ms

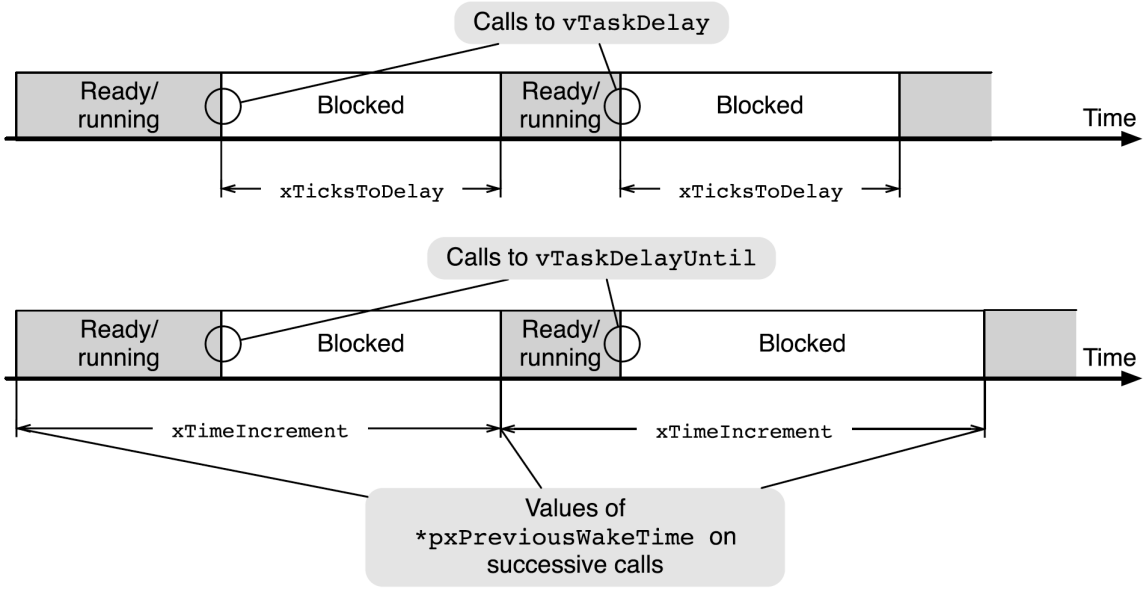

FIG 6. Relative versus absolute delays

Task Priority

- Higher priorities task run before lower priority task
- Tasks with the same priority share CPU time (in time slices)
- The scheduler runs at the end of each time slice to select the next task to run
- The length of the time slice is set by the SysTick interrupt frequency by configuring the configTICK_RATE_HZ while setting the compile configuration in FreeRTOSConfig.h
- The initial priority of a task is assigned when created by using xTaskCreate()
- The task priority can be queried by using the xTaskPriorityGet()
- The task priority can be changed by using vTaskPrioritySet()
- Low numeric values denote low priority tasks (0 is the lowest priority)

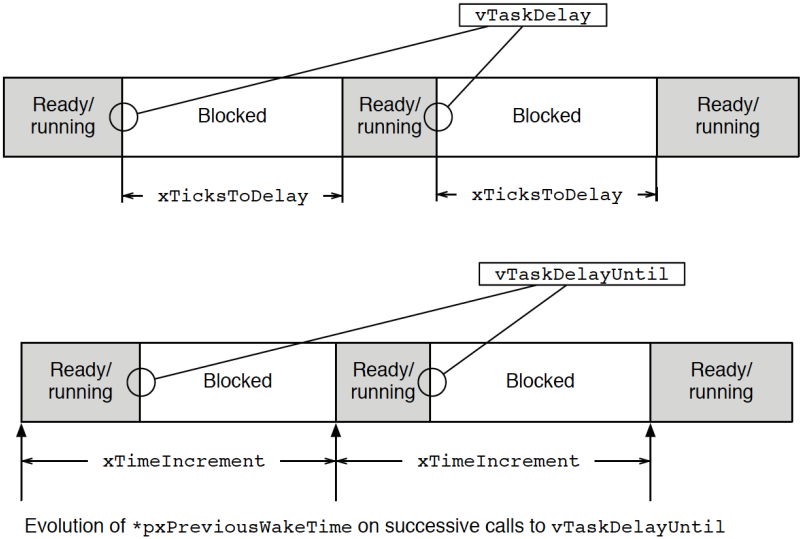

FIG 7. Comparison between relative and absolute time delays, as implemented by vTaskDelay and vTaskDelayUntil.—An absolute delay is better when a task has to carry out an operation periodically because it guarantees that the period will stay constant even if the response time of the task—the grey rectangles in Figure—varies from one instance to another

Idle Task and Idle Task Hook Functions

- The idle task is executed when no application tasks are in running state[19]
- The idle task is automatically created by the scheduler when vTaskStartScheduler() is called
- The idle task has the lowest possible priority(priority 0) to ensure it never prevents a higher priority application task
- Application specific functionality can be directly added into the idle task via an idle hook (or call-back) function
  - An idle hook function is automatically called per iteration of the idle task loop
  - Can be utilized to execute low priority, background or continuous processing

```
1    void vApplicationIdleHook(){}
2    void vApplicationTickHook(){}
3    void vApplicationAssertHook(){}
4    void vApplicationMallocFailedHook(){}
```

[19] https://www.freertos.org/a00016.html

TAB 3. Summary of the task-related primitives of FreeRTOS[20]

Function	Purpose	Optional
`vTaskStartScheduler`	Start the scheduler	-
`vTaskEndScheduler`	Stop the scheduler	-
`xTaskCreate`	Create a new task	-
`vTaskDelete`	Delete a task given its handle	*
`uxTaskPriorityGet`	Get the priority of a task	*
`vTaskPrioritySet`	Set the priority of a task	*
`vTaskSuspend`	Suspend a specific task	*
`vTaskResume`	Resume a specific task	*
`xTaskResumeFromISR`	Resume a specific task from an ISR	*
`xTaskIsTaskSuspended`	Check whether a task is suspended	*
`vTaskSuspendAll`	Suspend all tasks but the running one	-
`xTaskResumeAll`	Resume all tasks	-
`uxTaskGetNumberOfTasks`	Return current number of tasks	-

[20]The elements marked with * are optional; they may or may not be present, depending on the FreeRTOS configuration

Remarks

- When main() executes, the FreeRTOS's scheduler is not yet active. It can be explicitly started and stopped by means of the following function calls

```
1 void vTaskStartScheduler(void);
2 void vTaskEndScheduler(void);
```

- The function vTaskStartScheduler() reports errors back to the caller in an unusual way:
  - When it is able to start the scheduler successfully, vTaskStartScheduler does not return to the caller at all
  - Otherwise, vTaskStartScheduler() may return to the caller for two distinct reasons:
    1. The scheduler was successfully started, but one of the tasks then stopped it by invoking vTaskEndScheduler().
    2. The scheduler was not started at all because an error occurred.
- After creation, it is possible to change the priority of a task or retrieve it by means of the functions

```
1 void vTaskPrioritySet(TaskHandle_t task,UBaseType_t priority);
2 UBaseType_t uxTaskPriorityGet(TaskHandle_t xTask);
```

The end