1. FreeRTOS

O *FreeRTOS* é um kernel (gerenciador) utilizado em aplicações embarcadas que necessitam de aplicação em tempo real, sendo normalmente empregado em CubeSats. Esse kernel, desenvolvido e mantido pela Real Time Engineers Ltd, é distribuído gratuitamente sobre a licença *General Public License* (GPL) (BARRY,2018). O FreeRTOS foi desenvolvido para ser pequeno, portátil e escalável; de acordo com o site oficial­1 o kernel possui uma imagem típica de 6K a 12K bytes.

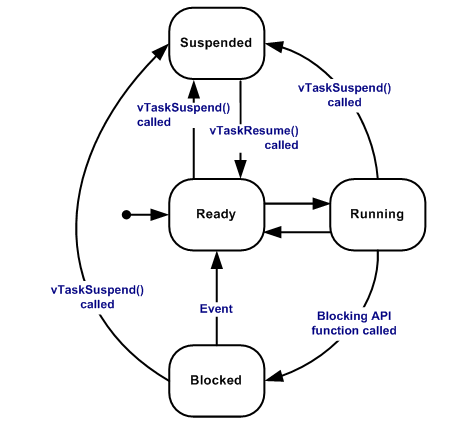
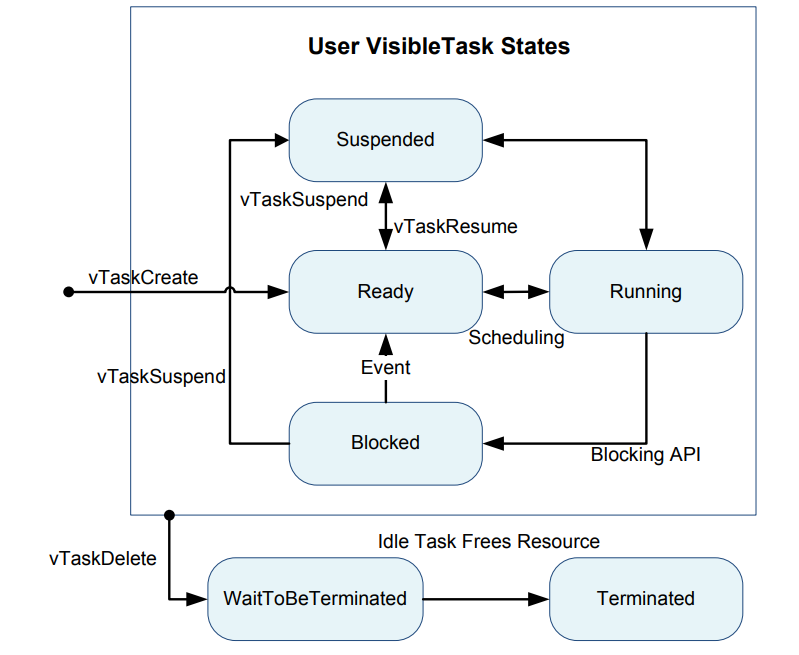
1. Disponível em<https://www.freertos.org/> Acesso em: 18/06/2018

O *FreeRTOS* é utilizado tanto em sistemas Não Críticos de Tempo Real (Soft-RTOS), quanto em sistemas Críticos de Tempo Real (Hard-RTOS). Aplicações Soft-RTOS possuem um tempo especifico para a realização de uma tarefa, mas o não cumprimento do prazo de execução não causa uma falha no sistema. Por exemplo, caso as luzes internas do carro não acenderem no instante em que a porta for aberta, o usuário não estaria em risco. Já as aplicações Hard-RTOS, o não cumprimento do prazo resulta em falha do sistema. Por exemplo, se o sensor de colisão do carro atrasar a sua resposta, o usuário estaria em grande risco.

No caso de aplicações aeroespaciais, o sistema deve ser projetado para ser um Hard-RTOS pois o travamento do *software* embarcado ocasionaria em falha total do sistema. Foi percebido que todas as missões pesquisadas, Apêndice B, possuíam algum tipo de sistema operacional para gerenciar as tarefas embarcadas.

No FreeRTOS, cada tarefa em execução é chamada de ‘*task’*. Para manter um padrão, será utilizada essa nomenclatura. No contexto do projeto, o uso das *tasks* é fundamental para criar um certo nível de abstração e garantir o requisito de Hard-RTOS. Por exemplo, a leitura da tensão da bateria é de extrema importância, então ela possuirá maior prioridade em relação a *task* de envio de dados; pois se o satélite consumir por completo a carga da bateria, ocasionando na perda do satélite.

Uma *task* só pode estar em quatro estados: *Ready*, *Running*, *Suspend* e *Blocked*. As APIs são encarregadas de alterar os estados de cada *task*. Os estados são melhores ilustrados na figura abaixo, onde há evidenciado as APIs responsáveis pela mudança.

* **Running**

When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

* **Ready**

Ready tasks are those 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.

* **Blocked**

A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.

Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

* **Suspended**

Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

* 1. FreeRTOS APIs

As APIs do FreeRTOS foram criadas para facilitar o desenvolvimento, fazendo uma interface entre o usuário e o kernel. De acordo com o site oficial há 145 APIs que são agrupadas nas seguintes categorias:

* *Task Creation,* permite que o usuário crie ou elimine uma *task* do sistema.
* *Task Control,* permite a troca dos estados das *tasks*.
* *Task Utilities,* possui APIs que APIs que permitem o usuário pegar alguma informação sobre as tasks, por exemplo, quais *tasks* estão em execução.
* *RTOS Kernel Control,* essa API oferece controle sobre o kernel, permitindo o usuário começar o *task scheduler,*  suspender as *tasks* em execução, entre outros.
* *Direct To Task Notifications ,* permite a troca de mensagens entre as *tasks.*
* *FreeRTOS-MPU Specific,* possui 3 APIs que permitem o usuario realizar o controle do kernel MPU.
* *Queues,* permite a comunicação entre as tarefas e interrupções a partir de uma fila.
* *Queue Sets,* permite o controle mais geral das filas, por exemplo, bloqueio de acesso das tarefas às filas.
* *Semaphore/Mutexes,* permite que usuário use semáforos binários para sincronização.
* *Software Timers,* permite a utilização de temporizadores, muito utilizado para contabilizar o tempo gasto entre as *tasks.*
* *Event Group,* essas APIs são flags que indicam se um evento ocorreu ou não, sendo muito utilizado para a gerar logs do sistema.
* *Co-routines*, permite a criação de rotinas, uma alternativa para tarefas com pouca prioridade.

Descrever cada API do kernel seria um tarefa cansativa e desnecessária, uma vez que há o manual de referência que explica cada API e mostra alguns exemplos. Com isso, abaixo há uma tabela que contem a descrição das funcionalidades das APIs das categorias *Task Creation* e *Task Control.* Essas categorias foram escolhidas para serem tratadas no texto, pois a manipulação das tarefas é a ação mais usual em um projeto com o FreeRTOS.

Abaixo há uma breve descrição sobre as APIs das categorias *Task Creation* e *Task Control.*

|  |  |  |
| --- | --- | --- |
| Categoria | Nome da API | Funcionalidade |
| Task Creation | xTaskCreate() | Criação de uma task com sua prioridade. Se a prioridade for maior que a task em execução, a task criada começa sua execução. A API retorna um handler para a tarefa. |
| xTaskCreateStatic() | Cria uma task, semelhante ao xTaskCreate. Entretanto, o tamanho destinado para a RAM é definido pelo usuário e é alocado estaticamente. A API retorna um handler para a tarefa. |
| vTaskDelete() | Realiza a remoção da tarefa. Se a task estiver em execução, o kernel busca uma tarefa de maior prioridade para ser executada. |
| Task Control | vTaskDelay() | Coloca a task, que está em execução, no modo Blocked e aciona o timer para despertar. |
| vTaskDelayUntil() | Coloca uma tarefa em execução no estado de Bloacked e configura o tempo de resumo. |
| uxTaskPriorityGet() | Retorna a prioridade da task a partir do seu handler. |
| vTaskPrioritySet() | Define uma nova prioridade para a task. |
| vTaskSuspend() | Coloca a task em modo Supended e localiza outra tarefa com prioridade maior para ser executada. |
| vTaskResume() | Coloca a tarefa suspensa para o estado de Ready. |
| xTaskResumeFromISR() | Coloca a tarefa suspensa no estado de Ready quando o *Scheduler* estiver parado. |
| xTaskAbortDelay() | Força uma tarefa a sair do modo *Blocked* e entrar no *Ready* |

BARRY, Richard. **Mastering the FreeRTOS™ Real Time Kernel**: A Hands-On Tutorial Guide. [S.l.]: Real Time Engineers Ltd., 2016. 399 p. Disponível em: <https://www.freertos.org/Documentation/161204\_Mastering\_the\_FreeRTOS\_Real\_Time\_Kernel-A\_Hands-On\_Tutorial\_Guide.pdf>. Acesso em: 17 maio 2018.

Visando ser um kernel pequeno, portátil e escalável, a parte principal do FreeRTOS consiste de apenas 3 arquivos em C. O núcleo do FreeRTOS está presente em apenas 2 arquivos, task.c e list.c, e estão presentes no diretório FreeRTOS/Source. Um adicional de mais quatro arquivos fazem parte do sistema, que são: queue.c, timers.c, event\_group.c e croutine.c (BARRY,2008). De acordo com o site do FreeRTOS1, o kernel possui uma imagem típica de 6K a 12K bytes.

Aplicações Soft-RTOS possuem requisitos que estabelecem um tempo especifico para a realização de uma tarefa, mas o não cumprimento do prazo de execução não causaria uma falha no sistema. Por exemplo, caso as luzes internas do carro não ligarem no instante que a porta for aberta não seria um risco ao usuário.

Já as aplicações Hard-RTOS possuem requisitos que estabelecem um tempo especifico para a realização de uma tarefa e o não cumprimento do prazo resulta em falha do sistema. Por exemplo, se o sensor de colisão do carro atrasar a sua resposta, o usuário estaria em grande risco.

Esse kernel de tempo real é amplamente utilizado em aplicações embarcadas, tanto em Sistemas Não Críticos de Tempo Real (Soft-RTOS) quando em Sistemas Críticos de Tempo Real (Hard-RTOS)

No caso do software embarcado do OBDH, qualquer atraso causado por um travamento, ou execução de uma tarefa, pode ser catastrófico para a missão. Então para a aplicação o uso do FreeRTOS é aconselhada.

O FreeRTOS, desenvolvido e mantido pela Real Time Engineers Ltd, é distribuído gratuitamente sobre a licença *General Public License* (GPL). O *FreeRTOS* é um kernel, ou gerenciador, de tempo real amplamente utilizado em aplicações embarcadas que necessitem de aplicação em tempo real. Esse kernel é amplamente utilizado em aplicações embarcadas, tanto em aplicações de Sistemas Não Críticos de Tempo Real (Soft-RTOS) quando de Sistemas Críticos de Tempo Real (Hard-RTOS) (BARRY,2018).

An incremental modeling strategy is carried out in their project. They start with a basic model, and then take priority and mutex into account in the next two further incremental modeling. Mainly, there are 7 machines in this formal model, Config, Types, Queue, Scheduler, FreeRTOSBasic and FreeRTOS. More details of these machines are presented in Chapter 3.

The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

\begin{array}{ | l | l | l | l | l | l | l | l | l | l | l | l | l | l | l | l | l | }

\hline

Missão & & & & Microcontrolador & & & & & & & & & & & & \\ \hline

Informações Gerais & & & & Especificações Gerais & & & & Conversores & & Interface e Portas & & & Informações Adicionais & & & \\ \hline

Nome & Tipo de Missão & Unidade & TRL & Microcontrolador & Consumo & Barramento de Dados & Frequencia & AD & DA & I/O & Interface

Serial & I2C & Modos de Operação & Sistema Operacional & Watchdog Timer & Faixa de Temperatura \\ \hline

Hawksat-1 & Demostrador Tecnológico & 1U & 9.0 & MSP430F1612 & 12mW @ 8MHz & 16-bit & 8MHz & 8.0 & 2.0 & 48.0 & 2 USART, 2 SPI,

& 1.0 & 2.0 & - & sim & -40° a 80°C \\ \hline

SwissCube & Observação da Terra & 1U & 9.0 & AT91M55800A ARM & 140mW @ 33MHz & 32-bit & 33MHz & 8.0 & 2.0 & 58.0 & 3 USART, 2 SPI,

& 0.0 & 5.0 & x & sim & -40° a 85°C \\ \hline

Aalto-1 & Observação da Terra & 1U & 9.0 & AT91RM9200

ARM & 80mW@80MHz & 32-bit & 180MHz & - & - & 122.0 & 3 USART, 1 SPI,

& 0.0 & 2.0 & Linux & sim & -40 ºC a +85 ºC \\ \hline

GOMX-1 & Demonstrador Tecnológico & 2U & 9.0 & Nanomind A712D

- AT91M55800A & 293,7mW @ 40MHz & 32-bit & 40 MHz & 6.0 & 2.0 & 50.0 & USART,

SPI,

& 1.0 & 5.0 & Linux & sim & -40 ºC a +85 ºC \\ \hline

OpenOBC & x & x & 43195.0 & TMS570LS0432

ARM

®

Cortex

®

-R4 & 405mW @ 80MHz & 32-bit & 80MHz & 16.0 & - & 45.0 & 1 USART,

2 SPI, & 2.0 & 2.0 & FreeRTOS & sim & -40 ºC a +85 ºC \\ \hline

FloripaSat & Projeto Universitário & 1U & 43195.0 & MSP430F6659IPZ

RISC & 0.7mW @ 8MHz & 16-bit & 16 MHz & 16.0 & - & 74.0 & 3 USART,

3 SPI,

& 3.0 & 8.0 & FreeRTOS & sim & -40 ºC a +85 ºC \\ \hline

AAUSAT & Observação da Terra & 3U & 9.0 & C161PI & 150mW @ 25MHz & 16-bit & 25MHz & 4.0 & x & 76.0 & 2 USART, 2 SPI, & 2.0 & 2.0 & Keil RTX166 & sim & -40 ºC a +85 ºC \\ \hline

ATMOCUBE & Observação da Terra & 1U & 6.0 & PIC 16F877 & 1,6mW @ 4MHz & 8-bit & 20MHz & 8.0 & x & 14.0 & 1 USART, 1 MSSP & 0.0 & 2.0 & PIC 16F877 & sim & -55ºC a +125 ºC \\ \hline

\end{array}