INDEX

1. FreeRTOS Selection Criteria **2**

1.1. Technical Consideration 2

1.1.1. Portability 2

1.1.2. Scalability 2

1.1.3. Code and Data size requirements 2

1.1.4. Run time feature 3

1.1.5. Runtime Performance 3

1.1.6. Meeting Application Requirement 3

1.2. Commercial Consideration 3

1.3. FreeRTOS Features 4

2. FreeRTOS kernel functions **5**

2.1. Task Management 5

2.2. Memory Management 5

2.3. Interrupt Management 5

2.4. Resource Management 5

2.5. Low Power Support 6

2.5.1. Idle Hook function 6

2.5.2. Power-aware Task Delay 6

2.5.3. Tickless Idle Mode 6

3. FreeRTOS Architecture **8**

3.1. Source Code directory Structure 8

3.2. Creating a FreeRTOS Project 9

3.3. Coding Standards and Naming Conventions 12

3.3.1. Data Types 13

3.3.2. Naming Conventions 15

3.3.3. Function Names 16

3.3.4. Macro Names 17

FreeRTOS

1. FreeRTOS Selection Criteria

Selecting the right Real-Time Operating System(RTOS) for your project involves evaluating several criteria to choose RTOS that meets the specific needs of your application.

1.1. Technical Consideration

1.1.1. Portability

- Portability is supported by microcontroller architecture.
- In a real-world example, switching the microcontroller in a smart home device, portability significantly reduces development time and effort, allowing companies to be more adaptable and efficient.

1.1.2. Scalability

- Ability of the real-time operating system to handle varying levels of system complexity, workload, and resource requirements.
- It supports software standards such as USB, PS, TCP/IP
- FreeRTOS is designed to be scalable to accommodate a wide range of embedded applications, from simple, resource-constrained microcontrollers to more complex systems with multiple tasks, communication protocols, and peripherals

1.1.3. Code and Data size requirements

- In FreeRTOS, both stack and heap memory contribute to the overall data size and memory usage of the system. Task stacks are used for storing execution context and local variables, while the heap is used for dynamic memory allocation.
- By carefully managing stack and heap usage, developers can optimise memory usage and ensure efficient operation of embedded systems built with FreeRTOS.

1.1.4. Run time feature

- FreeRTOS provides a variety of runtime features that facilitate real-time task scheduling, inter-task communication, synchronisation, and memory management.
- These features are designed to meet the requirements of embedded systems and real-time applications.

1.1.5. Runtime Performance

- Runtime performance in FreeRTOS refers to the efficiency and responsiveness of the real-time operating system during the execution of tasks and management of system resources.

1.1.6. Meeting Application Requirement

- Meeting application requirements in FreeRTOS involves careful design, implementation, and testing of tasks, ISRs, and synchronization mechanisms. By organizing the system into well-defined tasks, utilizing queues and semaphores for synchronization, and keeping ISRs efficient

1.2. Commercial Consideration

- FreeRTOS is distributed under the MIT open-source licence, which allows for free commercial and non-commerical use. This means you can use FreeRTOS in your commercial projects without paying licensing fees.
- Cost and Supplier Stability.
- OS image : when we want to run an OS on a hardware we will run the required .c and .h files and get the OS image which is further flashed on our required hardware.
- RTOS SIZE should be < 100Mb.

1.3. FreeRTOS Features

1. Scheduler
2. Tickless Mode
3. Dynamic and Static Memory
4. Efficient Software Timers
5. Powerful Execution Trace Functionality
6. Stack Overflow detection Options
7. Free Source Code
8. Free development tools

2. FreeRTOS kernel functions

2.1. Task Management

- Tasks are the basic unit of execution in FreeRTOS. Each task is typically implemented as an infinite loop and is assigned a priority.
- To create, delete, control, and schedule tasks by using an API

2.2. Memory Management

- It should be allocated memory, free, and schema
- **Dynamic Memory Allocation :** FreeRTOS provides several heap management schemes.
- **Static Memory Allocation :** Allows for memory to be allocated at compile time.
- Task, queue and semaphore creation functions have static versions.

2.3. Interrupt Management

- Interrupt management is a crucial aspect of real-time operating systems, Interrupt is the highest priority and we cannot pre-emptive
- **Interrupt Service Routines (ISRs) :**
    - ISRs are functions that get executed in response to hardware interrupts.
    - They should be kept short and fast to minimise the time spent with interrupts disabled and FreeRTOS provides specific API functions to handle tasks and communications within ISRs.

2.4. Resource Management

- Resource management in FreeRTOS involves efficiently handling tasks, memory, and inter-task communication to ensure smooth and reliable operation in resource-constrained embedded systems

2.5. Low Power Support

- FreeRTOS offers several features and techniques to support low power consumption in embedded systems. By leveraging tickless idle mode, power-aware task delay, and idle hook function.

2.5.1. Idle Hook function

- FreeRTOS provides an idle hook function, allowing developers to implement custom low-power strategies during idle periods. This hook function is called whenever the scheduler detects that there are no tasks to execute.
- **Usage :**
  - Implement a custom idle hook function to put the system into a low-power state during idle periods.

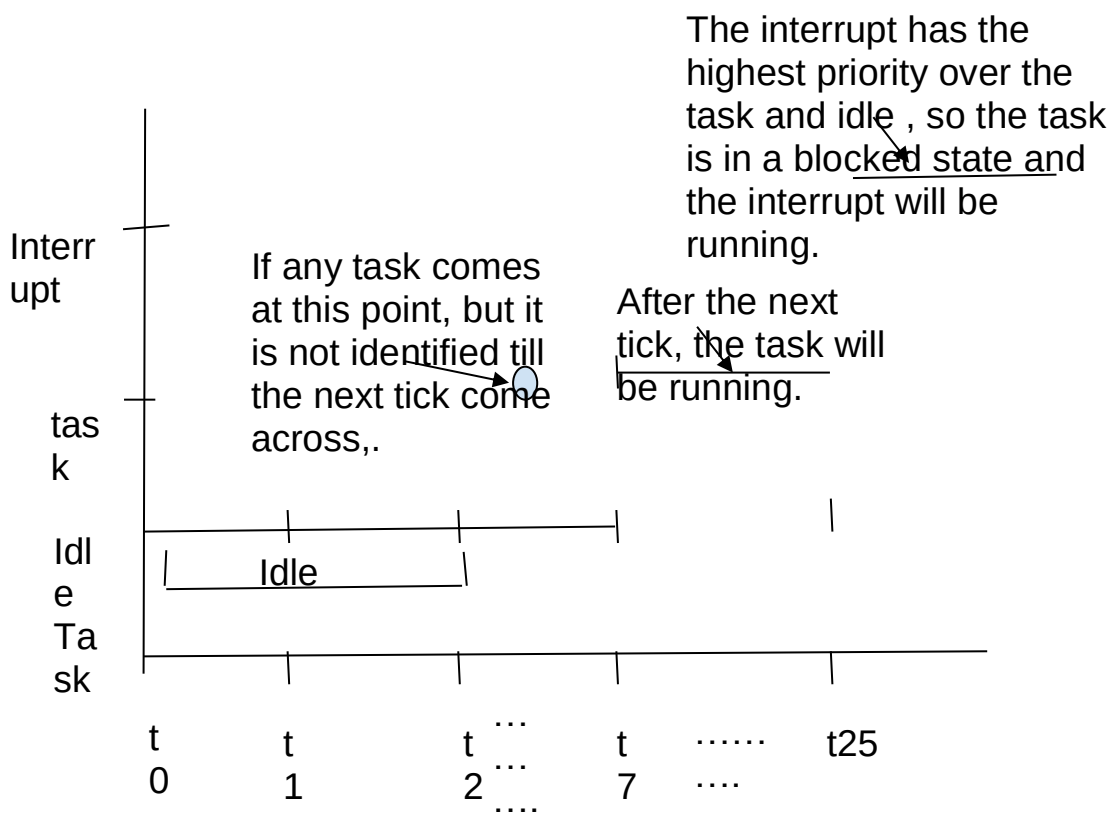
2.5.2. Power-aware Task Delay

- Tasks in FreeRTOS can use the vTaskDelayUntil() function to specify a time to wake up from a low-power state. This allows tasks to sleep until a specific time, conserving power when there is no immediate work to be done.

2.5.3. Tickless Idle Mode

- FreeRTOS includes a tickless idle mode, which allows the CPU to enter a low-power state during idle periods instead of continuously running the tick interrupt. This feature reduces power consumption significantly, especially in battery-operated devices.
- Enable Tickless Idle Mode by setting configUSE_TICKLESS_IDLE to 1 in FreeRTOS configuration.

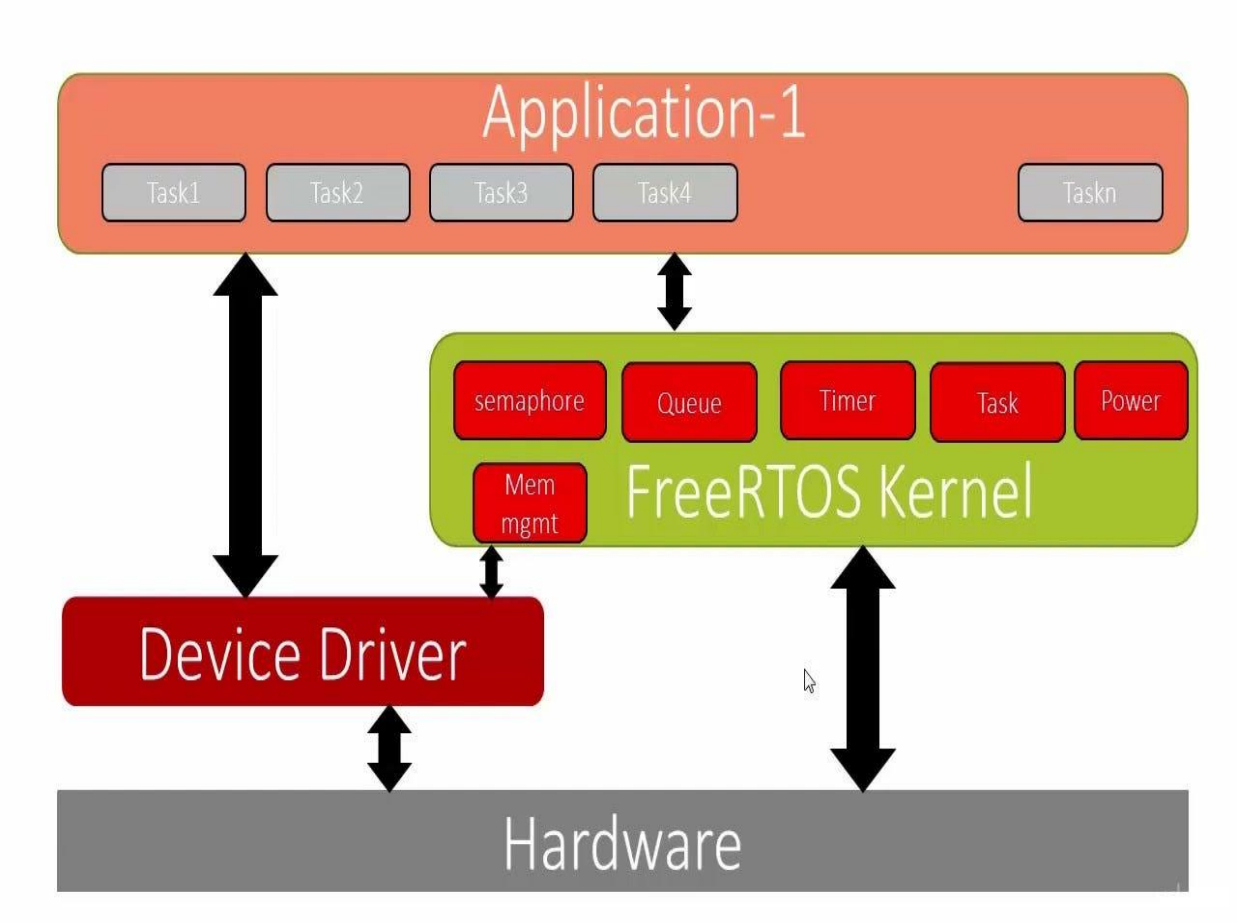
Example



In the above figure, since it has been idle for a long time, it may wake up to check at t1 instead of every tick.

- If there is an external interrupt, then it may wake up in the middle of the tick time as well.
- If we turn off the tick for a long time, we can wait until that time to execute the task.

3. FreeRTOS Architecture



3.1. Source Code directory Structure

+-FreeRTOS-Plus → Contains FreeRTOS-Plus components and demo projects.

|

+-FreeRTOS → Contains the FreeRTOS real time kernel source files and demo projects

FreeRTOS

|

+-Demo → Contains the demo application projects.

|

+-Common → The demo application files that are used by all the demos.

+-Dir x → The demo application build files for port x

+-Dir y → The demo application build files for port y

|

+-Source → The core FreeRTOS kernel files

|

+-include → The core FreeRTOS kernel header files

|

+-Portable → Processor specific code.

|

+-Compiler x → All the ports supported for compiler x

+-Compiler y → All the ports supported for compiler y

+-MemMang → The sample heap implementations

3.2. Creating a FreeRTOS Project

1. **creating a new project from demo project：**To start a new application from an existing demo project:
    - Open the supplied demo project and ensure that it builds and executes as expected.

- Remove the source files that define the demo tasks. Any file that is located within the Demo/Common directory can be removed from the project.
- Delete all the function calls within main(), except **prvSetupHardware**() and **vTaskStartScheduleR()**,
- Check the project still builds.

2. Following these steps will create a project that includes the correct FreeRTOS source files, but does not define any functionality.

```
int main( void )
{
/* Perform any hardware setup necessary. */
prvSetupHardware();
/* --- APPLICATION TASKS CAN BE CREATED HERE --- */
 /* Start the created tasks running. */
 vTaskStartScheduler();
/* Execution will only reach here if there was insufficient heap to start the scheduler.
*/
for( ;; );
return 0;
 }
```

3. **Creating a New Project from Scratch :** As already mentioned, it is recommended that new projects are created from an existing demo project. If this is not desirable, then a new project can be created using the following procedure:
    - Using your chosen tool chain, create a new project that does not yet include any FreeRTOS source files.
    - Ensure the new project can be built, downloaded to your target hardware, and executed.
    - Only when you are sure you already have a working project, add the FreeRTOS source files detailed in Table 1 to the project.
    - Copy the FreeRTOSConfig.h header file used by the demo project provided for the port in use into the project directory.
    - Add the following directories to the path the project will search to locate header files:

4. FreeRTOS/Source/include
5. FreeRTOS/Source/portable/[compiler]/[architecture] (where [compiler] and [architecture] are correct for your chosen port)
6. The directory containing the FreeRTOSConfig.h header file
7. Copy the compiler settings from the relevant demo project.
8. Install any FreeRTOS interrupt handlers that might be necessary. Use the web page that describes the port in use, and the demo project provided for the port in use, as a reference

file	location
Tasks.c	FreeRTOS/Source
Queue.c	FreeRTOS/Source
List.c	FreeRTOS/Source
Timers.c	FreeRTOS/Source
Event_groups.c	FreeRTOS/Source
All c and assembler files	FreeRTOS/Source/portable/[compiler]/[architecture]
Heap_n.c	FreeRTOS/Source/portable/MemMang/heap_n.c ,where n =1/2/3/4/5

3.3. Coding Standards and Naming Conventions

- Variables of type *uint32_t* are prefixed with ul, where the 'u' denotes 'unsigned' and the 'l' denotes 'long'.
- Variables of type *uint16_t* are prefixed with us, where the 'u' denotes 'unsigned' and the 's' denotes 'short'.
- Variables of type *uint8_t* are prefixed *uc*, where the 'u' denotes 'unsigned' and the 'c' denotes 'char'.
- Variables of non-stdint types are prefixed *x*. Examples include BaseType_t and TickType_t, which are portable layer defined typedefs for the type that is the natural or most efficient type for the architecture, and the type used to hold the RTOS tick count, respectively.
- Unsigned variables of non-stdint types have an additional prefix *u*. For example, variables of type UBaseType_t (unsigned BaseType_t) are prefixed *ux*.
- Variables of type *size_t* are also prefixed *x*.
- Enumerated variables are prefixed *e*
- Pointers have an additional *p* prefixed, for example, a pointer to a uint16_t will have a prefix *pus*.

- In line with MISRA guides, unqualified standard *char* types are only permitted to hold ASCII characters and are prefixed *c*.
- In line with MISRA guides, variables of type *char ** are only permitted to hold pointers to ASCII strings and are prefixed *pc*.

3.3.1. Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two port specific data types: **TickType_t** and **BaseType_t**.

Macro or typedef used	description
TickType_t	● FreeRTOS configures a periodic interrupt called the tick interrupt. ● The number of tick interrupts that have occurred since the FreeRTOS application started is called the tick count. The tick count is used as a measure of time. ● The time between two tick interrupts is called the tick period. Times are specified as multiples of tick periods. ● TickType_t is the data type used to hold the tick count value, and to specify times. ● TickType_t can be either an unsigned 16-bit type, or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is set to 1, then TickType_t is defined as uint16_t. If configUSE_16_BIT_TICKS is set to 0 then TickType_t is defined as uint32_t.

BaseType_t	● This is always defined as the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture. ● **BaseType_t** is typically defined as the most efficient signed integer type for the target architecture. For most 32-bit processors, it is defined as a **long** or **int**. For 8-bit or 16-bit processors, it might be defined as a smaller integer type to match the natural word size of the processor. ● **BaseType_t** is generally used for return types that can take only a very limited range of values, and for **pdTRUE/pdFALSE** type Booleans.
UBaseType_t	● This is an unsigned BaseType_t.
StackType_t	● Defined to the type used by the architecture for items stored on the stack. Normally this would be a 16-bit type on 16-bit architectures and a 32-bit type on 32-bit architectures, although there are some exceptions. Used internally by FreeRTOS.

TickType_t

Previous use of configUSE_16_BIT_TICKS has been replaced by configTICK_TYPE_WIDTH_IN_BITS to support tick counts greater than 32-bits. New designs should use configTICK_TYPE_WIDTH_IN_BITS instead of configUSE_16_BIT_TICKS.

configTICK_TYPE_WIDTH_IN_BITS	8-bit architectures	16-bit architectures	32-bit architectures	64-bit architectures
TICK_TYPE_WIDTH_16_BITS	uint16_t	uint16_t	uint16_t	N/A
TICK_TYPE_WIDTH_32_BITS	uint32_t	uint32_t	uint32_t	N/A
TICK_TYPE_WIDTH_64_BITS	N/A	N/A	uint64_t	uint64_t

Table 2 *TickType_t data type and the configTICK_TYPE_WIDTH_IN_BITS configuration*

// To use 16-bit tick counts

#define configUSE_16_BIT_TICKS 1

// To use 32-bit tick counts

#define configUSE_16_BIT_TICKS 0

3.3.2. Naming Conventions

- FreeRTOS Source files conform to the MISRA(Motor Industry Software Reliability Association) coding standard guidelines.
- Variables of type ***uint32_t*** are prefixed with ul, where the **'u'** denotes **'unsigned'** and the **'l'** denotes **'long'**.
- Variables of type ***uint16_t*** are prefixed with us, where the **'u'** denotes **'unsigned'** and the **'s'** denotes **'short'**.
- Variables of type ***uint8_t*** are prefixed *uc*, where the 'u' denotes **'unsigned'** and the **'c'** denotes **'char'**.
- Variables of non-stdint types are prefixed ***x***. Examples include **BaseType_t** and **TickType_t**, which are portable layer defined typedefs for the type that is the natural or most efficient type for the architecture, and the type used to hold the RTOS tick count, respectively.

- Unsigned variables of non-stdint types have an additional prefix ***u***. For example, variables of type **UBaseType_t** (unsigned **BaseType_t**) are prefixed ***ux***.
- Variables of type ***size_t*** are also prefixed ***x***.
- Enumerated variables are prefixed ***e***
- Pointers have an additional ***p*** prefixed, for example, a pointer to a **uint16_t** will have a prefix ***pus***.
- In line with MISRA guides, unqualified standard ***char*** types are only permitted to hold ASCII characters and are prefixed ***c***.
- In line with MISRA guides, variables of type ***char**** are only permitted to hold pointers to ASCII strings and are prefixed **pc**.
- Macros are prefixed with the file in which they are defined. The prefix is lowercase. For example, **configUSE_PREEMPTION** is defined in **FreeRTOSConfig.h.**

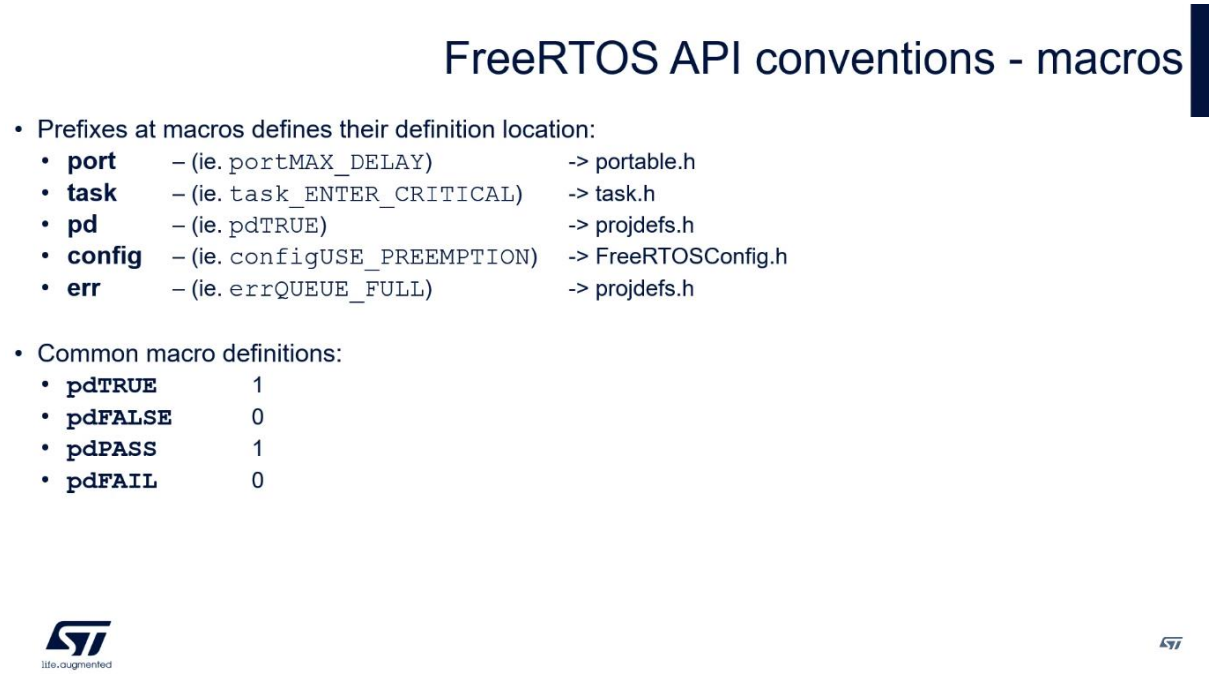
3.3.3. Function Names

Functions are prefixed with both the type they return, and the file they are defined within. For example:

- v**Task**PrioritySet() returns a void and is defined within **task.c.**
- x**Queue**Receive() returns a variable of type BaseType_t and is defined within **queue.c.**
- pv**Timer**GetTimerID() returns a pointer to void and is defined within **timers.c.**
- File scope (private) functions are prefixed with ‘prv’.

3.3.4. Macro Names

Most macros are written in uppercase, and prefixed with lower case letters that indicate where the macro is defined..



Note that the semaphore API is written almost entirely as a set of macros, but follows the function naming convention, rather than the macro naming convention.