**FreeRTOS**

**Introduction**

**INDEX:**

1. What is an OS?
   1. GPOS
   2. Why there is urge of embedded systems?
   3. Why OS in an embedded device?
2. RTOS
   1. Real time systems
   2. RTOS characterstics
   3. RTOS Fundamentals
      1. multitasking

2.3.1.1 single tasking systems

2.3.2 scheduling

2.3.3 context switching

2.3.3.1 the AVR context

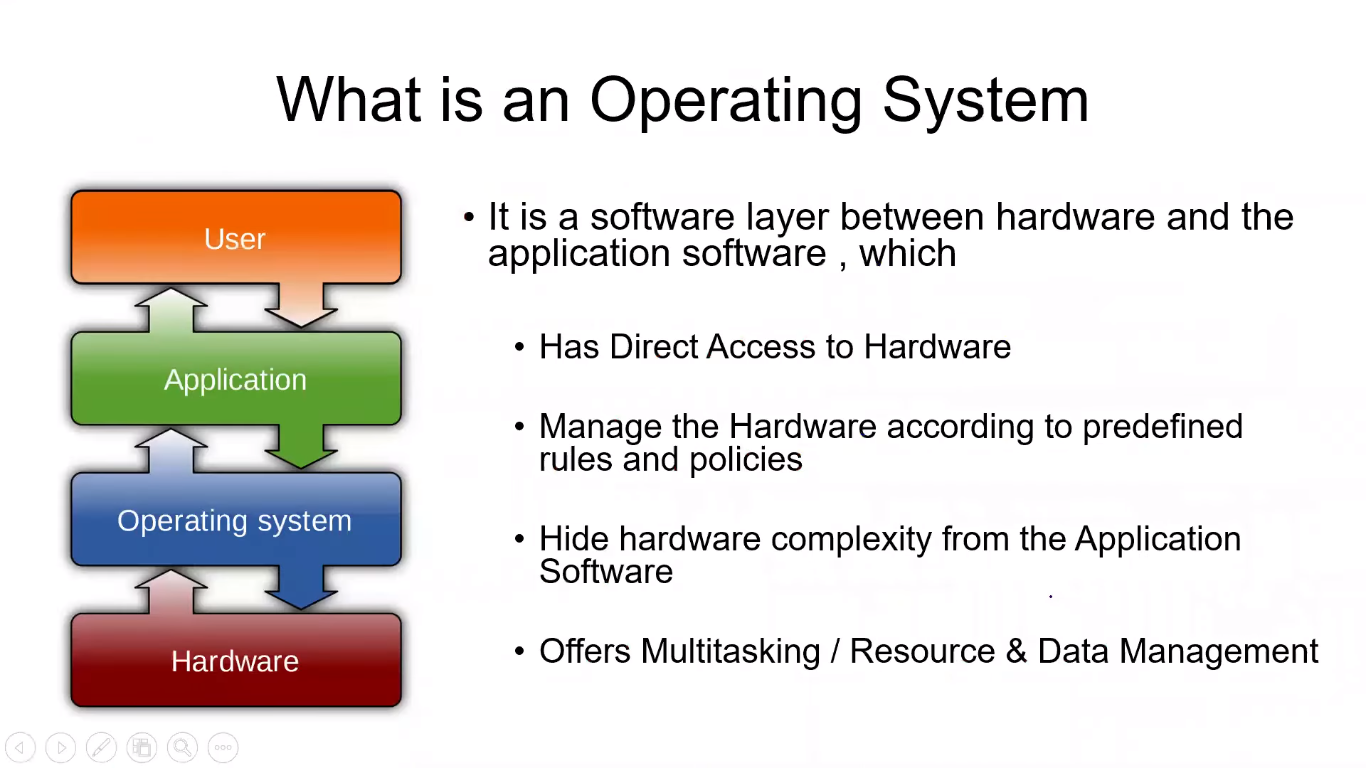
2.3.4 real time applications

2.3.5 real time scheduling

1. FreeRTOS
   1. main features
   2. FreeRTOS resources
2. FreeRTOS Distribution
   1. defination
   2. FreeRTOSConfig.h
   3. Top directories in FreeRTOS distribution
      1. source files common to all ports
      2. source files specific to port
      3. include paths
      4. header files
3. Creating a FreeRTOS project
   1. creating a new project from demo project
   2. creating new project from scratch
4. Coding style
   1. datatypes
   2. variable names
   3. function names
   4. formatting
   5. Macro names
5. Premilinary tips to start FreeRTOS
   1. configASSERT()
   2. malloc failed hook
   3. configCHECK\_FOR\_STACK\_OVERFLOW
      1. task states
      2. context switching
      3. AVR context
6. **What is an OS?**

Operating system is a piece of software that runs on a computer or microcontroller that accomplish number of important functions

1. Scheduling background tasks (it figures out how to give slices of time to each of these processes so everything happens concurrently) and users applications
2. It manages several virtual resources like files, libraries and folders allowing applications and processes to access them when needed
3. Provides device drivers for your system, these drivers allow the system to read and write from external disk respond to keyboard and mouse input
4. Examples: windows, mac os, Linux iOS, android
5. The scheduler is designed to prioritize such tasks, but often is non deterministic which means we cannot know exactly which task will execute when and how long.
6. At this time RTOS come into picture when you manage the timing deadlines and have the high-level device drivers



**GPOS(general purpose operating system) :**

* It is a (software) intermediate layer between application software and hardware
* It takes all the hardware complexities away from the hardware
* It provide APIs used by application software to control hardware
* Example:
* Windows

PC

* Linux
* Mac OS
* Android

Mobile

* IOS
* Embedded OS 🡪 embedded systems

**Definition :** an abstract layer between hardware and application software, that hides the hardware complexities from application software

**1.1 Why there is urge of embedded systems:**

1. Trend of reducing cost of computers

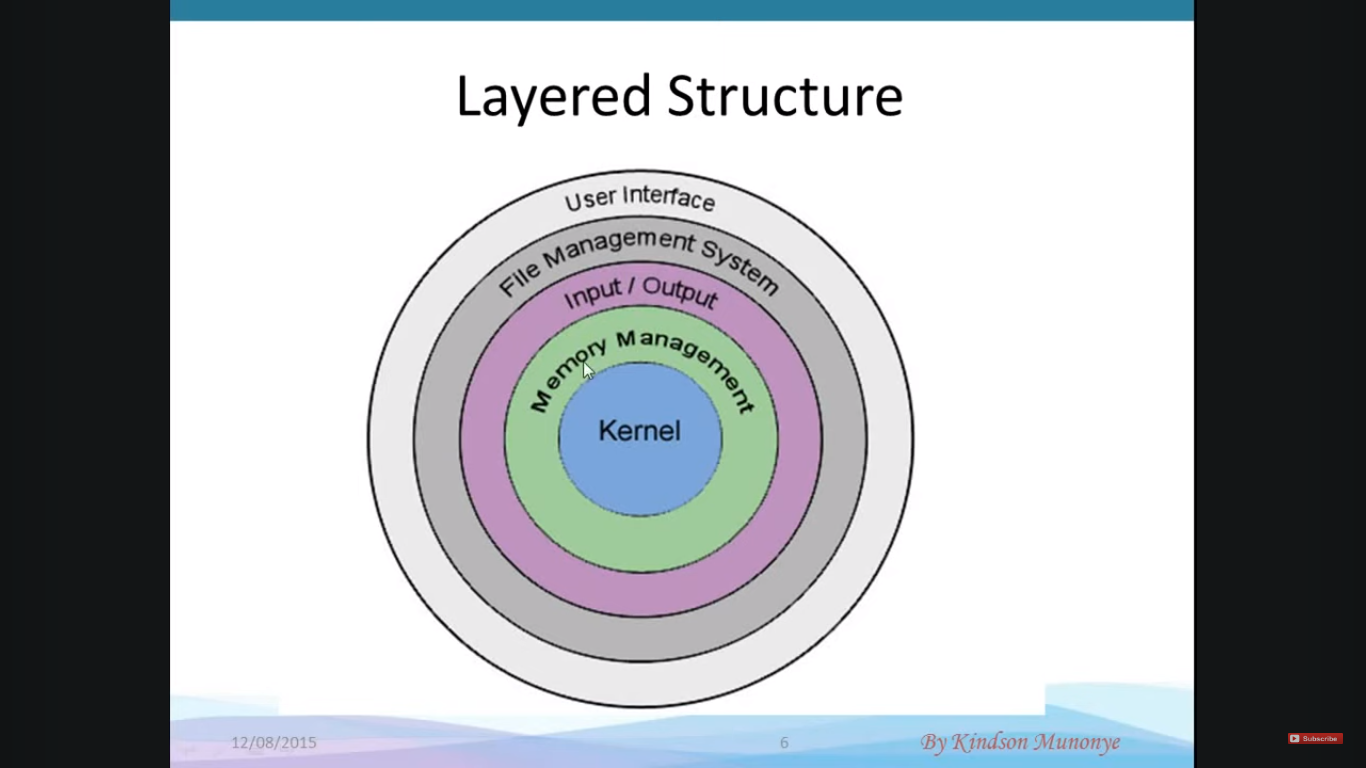
* Processors
* Memory

1. Flexibility due to internet
2. Reducing power consumption
3. Reducing size
4. Increasing processing power , hardware and software reliability

**1.2 Why OS in an embedded device?**

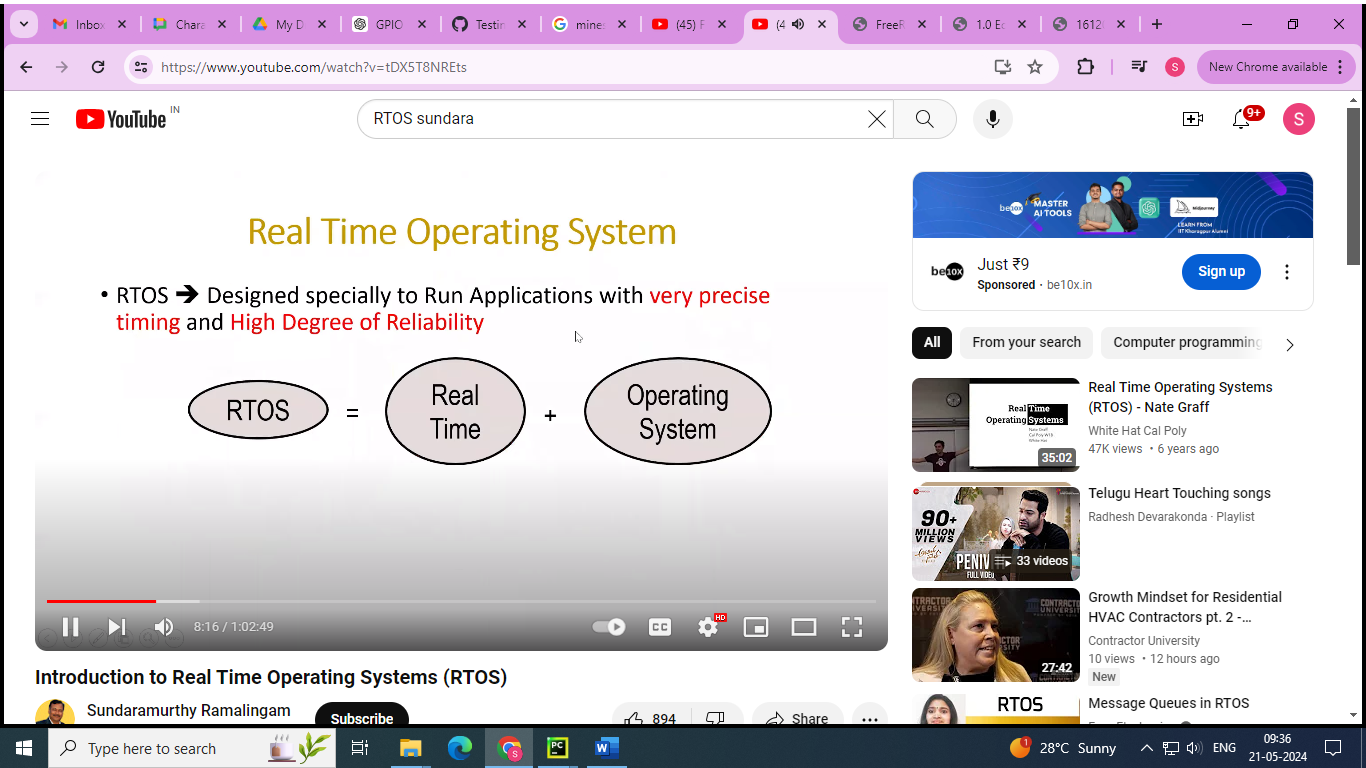
Supports for:

* Multi-tasking
* Scheduling
* Synchronization
* Timing aspects
* Memory management
* File systems
* Networking
* Graphics displays
* Interfacing wide range of I/O devices
* Scheduling and buffering of I/O operations
* Security and power management

****

1. **RTOS**

* It is designed to run the application with
* Very precise timing
* High degree of reliability
* a specialized operating system designed to manage hardware resources and execute tasks in real-time. Unlike general-purpose operating systems (like Windows or Linux), an RTOS is optimized for applications where precise timing and predictable responses are critical. Examples include embedded systems, industrial automation, robotics, automotive systems, and telecommunications.



* **Real time** : it should be operated with time constraint means it must have a deadline
* Responds to external events in a timely fashion
* The need to meet deadlines
* Does not mean faster performance

**2.1 Real-time systems:**

1. **Hard :** meets deadline all the time (100%) . missing deadline even single time cause fatal consequences

**Eg :** air bag systems

1. **Soft :** does not result fatal consequences. It may lower the qulity of service

**Eg** : DVD player

1. **Firm :** If a deadline is missed occasionally, the system dows not fail. Results produced by a task after deadline are ignored

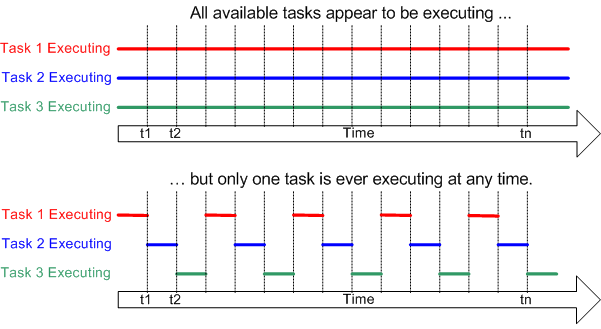
**2.2 RTOS characteristics:**

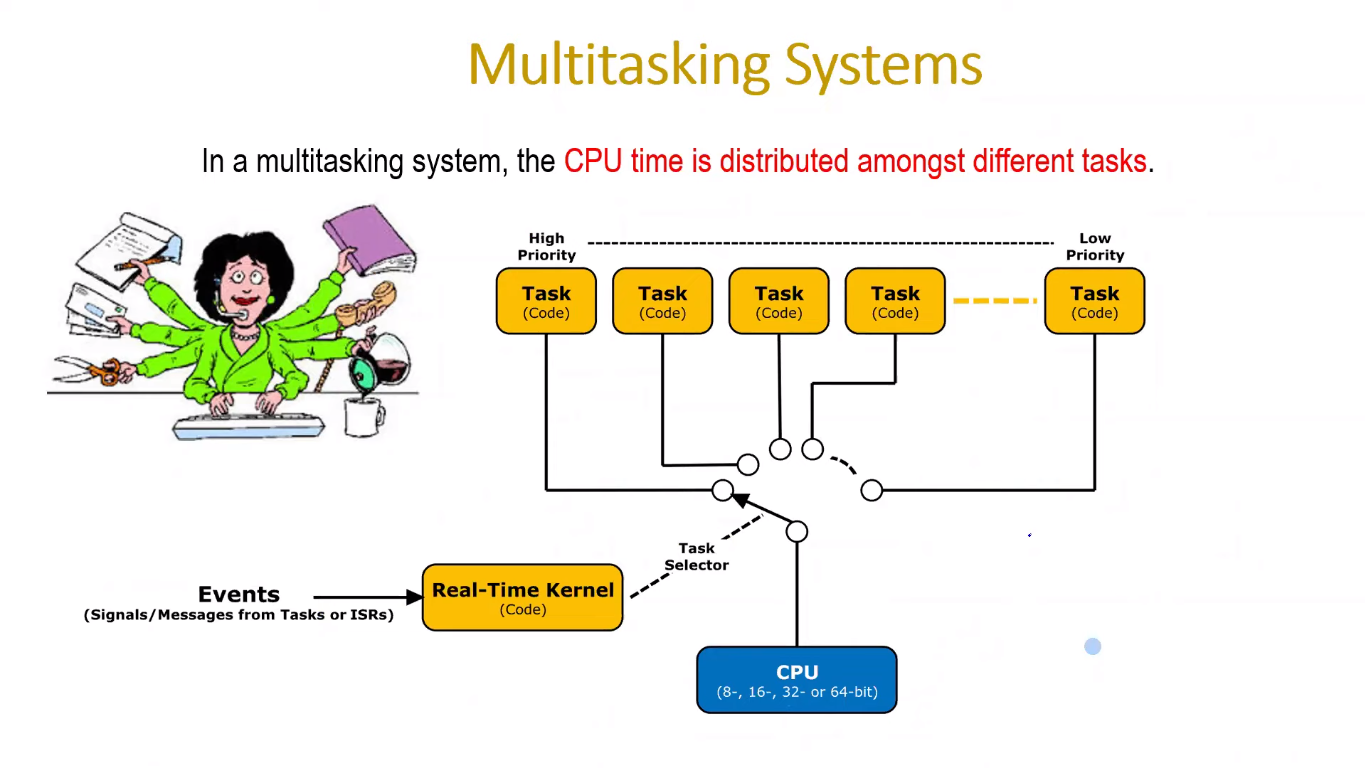
1. **Multi-tasking**

* Multiple super loops while(1) { }
* It allows execution of multiple tasks on single CPU
* All tasks execution as if they completely own entire CPU

1. **Kernel or scheduler**

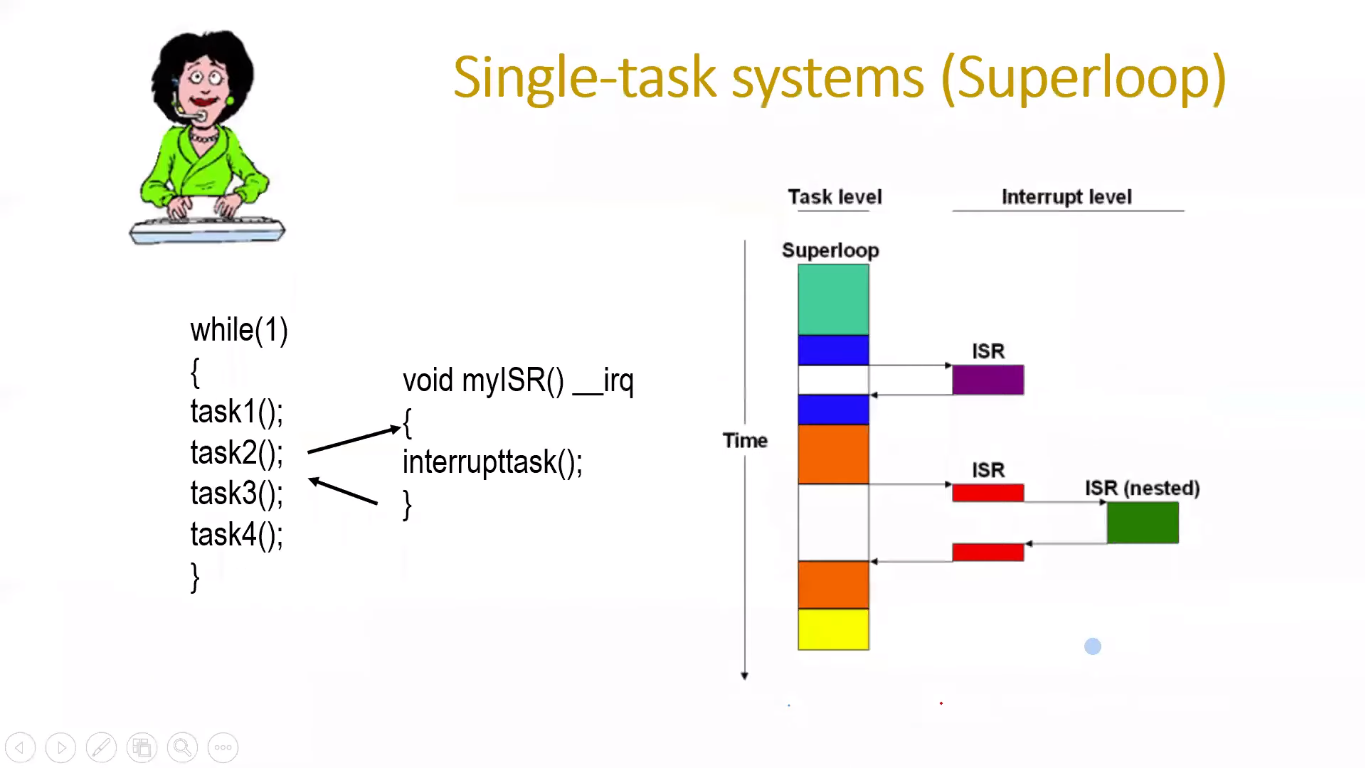
* Share processor time to tasks in predetermined manner / priority
* **Kernel** : It is central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time.
* Kernel loads first into memory when an operating system is loaded and remains into memory until operating system is shut down again. It is responsible for various tasks such as disk management, task management, and memory management.
* Kernel has a process table that keeps track of all active processes. Process table contains a per process region table whose entry points to entries in region table.
* Kernel loads an executable file into memory during ‘exec’ system call’.
* **Objectives of Kernel :**
* To establish communication between user level application and hardware.
* To decide state of incoming processes.
* To control disk management.
* To control memory management.
* To control task management.
* **Scheduler :** The scheduler is a crucial component responsible for managing the execution of processes by determining which process runs at any given time.
* **Process Scheduling**: The process of deciding which process gets CPU time and for how long. This involves managing the CPU's time among various processes to ensure fair and efficient utilization.
* **Context Switching**: When the scheduler switches the CPU from executing one process to executing another, it saves the state of the current process and loads the state of the next process. This is known as a context switch.
  1. **RTOS Fundamentals**
     1. **Multitasking**
* Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.
* 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**.
* A multitasking operating system can create the illusion of concurrent execution by rapidly switching between tasks, even though a single core processor can only execute one task at a time. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand. Time moves from left to right, with the coloured lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.





**2.3.1.1 Single tasking systems:**

* ISR(interrupt service routine) are used for real time parts of the application (critical operations)
* Super loop design
* It becomes difficult to maintain if program beomes too large or user complex intercations
* Reaction time depend on execution time of entire requirements resulting in poor real time behaviour
* Used in simple/small systems

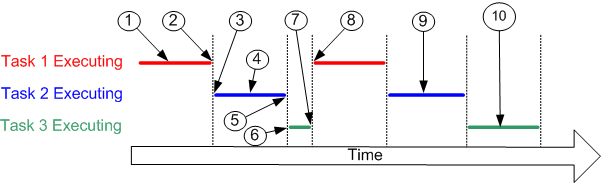


* + 1. **Scheduling**

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. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time..

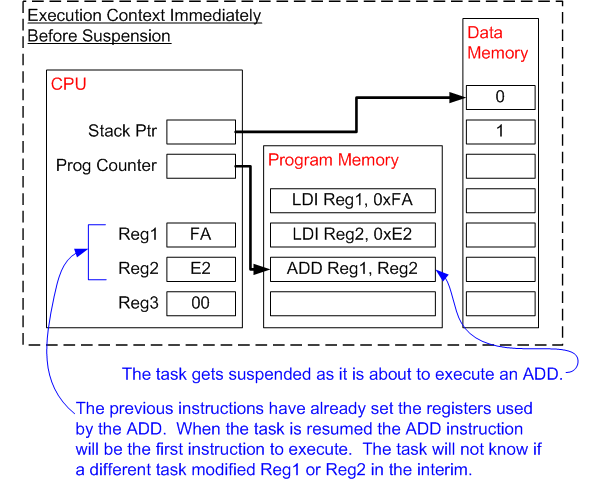
In addition to being suspended involuntarily by the kernel a task can choose to suspend itself. It will do this if it either wants to delay (**sleep**) for a fixed period, or wait (**block**) for a resource to become available (eg a serial port) or an event to occur (eg a key press). A blocked or sleeping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



Referring to the numbers in the diagram above:

* At (1) task 1 is executing.
* At (2) the kernel suspends (swaps out) task 1 ...
* ... and at (3) resumes task 2.
* While task 2 is executing (4), it locks a processor peripheral for its own exclusive access.
* At (5) the kernel suspends task 2 ...
* ... and at (6) resumes task 3.
* Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).
* At (8) the kernel resumes task 1.
* Etc.
* The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.
* The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

* + 1. **Context switching**
* As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution **context**.

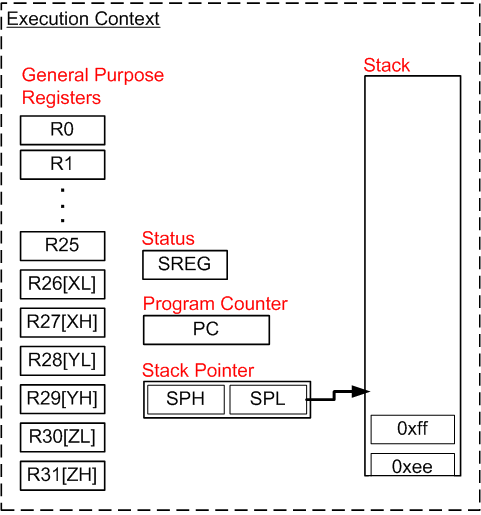


* A task is a sequential piece of code - it does not know when it is going to get suspended (swapped out or switched out) or resumed (swapped in or switched in) by the kernel and does not even know when this has happened.
* Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers. While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered – if it used the modified values the summation would result in an incorrect value.
* To prevent this type of error 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**.

**2.3.3.1 The AVR Context**  
[[RTOS Implementation Building Blocks](https://www.freertos.org/implementation/a00009.html)]

A context switch requires the entire execution context to be saved. On the AVR microcontroller the context consists of:

* 32 general purpose processor registers. The gcc development tools assume register R1 is set to zero.
* Status register. The value of the status register affects instruction execution, and must be be preserved across context switches.
* Program counter. Upon resumption, a task must continue execution from the instruction that was about to be executed immediately prior to its suspension.
* The two stack pointer registers.
* Saving the AVR context is one place where assembly code is unavoidable.
* portSAVE\_CONTEXT() is implemented as a macro



* + 1. **Real time applications**
* Real time / embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the real time / embedded system must respond and the RTOS scheduling policy must ensure these deadlines are met.

To achieve this objective the software engineer must first assign a priority to each task. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously

**2.3.5**

1. **FreeRTOS:**

* Marketing leading RTOS by real time engineers ltd.
* In STM32Cube firmware solution FreeRTOS is used as RTOS through the generic CMSIS-OS wrapping layer providing by ARM
* Examples and applications using the FreeRTOS can be directly ported on any other RTOS without modifying the high-level APIs , only CMSIS-OS wrapper must be changed
* CMSIS-OS (Cortex Microcontroller Software Interface Standard – Operating System) is a vendor-independent hardware abstraction layer for the Cortex-M processor series developed by ARM. It provides a standardized interface for real-time operating systems (RTOS) and software components to work across different microcontroller architectures.
  1. **Main features of FreeRTOS:**
* Pre-emptive or cooperative real time kernel
* Tiny memory footprint (less than 10kB ROM) and easy scalable
* Includes a tickles mode for low power applications
* Synchronization and inter-task communication using
* Message queues
* Binary and counting semaphores
* Mutexes
* Group events (flags)
* Software timers for tasks scheduling
* Execution trace functionality
* CMSIS\_RTOS API port
  1. **FreeRTOS-resources**

1. **Core resources**:

* **System timer (SysTick)** – generate system time

The SysTick timer is a hardware timer present in ARM Cortex-M processors, which provides a simple, programmable countdown timer. It’s commonly used in real-time operating systems (RTOS) like FreeRTOS for task scheduling and timekeeping.

* Two stack pointers : MSP, PSP

In FreeRTOS, MSP (Main Stack Pointer) and PSP (Process Stack Pointer) are two different stack pointers used for different purposes:

* **MSP (main stack pointer):**
  + The MSP is used to store the stack for the main thread of execution or the kernel itself.
  + It points to the main stack, which is typically used for interrupts and exceptions handling.
  + The main stack is shared among all threads in the system.
* **PSP (process stack pointer):**
  + The PSP is used to store the stack for the currently running task or process.
  + Each task has its own PSP, and the PSP is switched when a context switch occurs between tasks.
  + PSP allows each task to have its own stack, providing task isolation and preventing tasks from interfering with each other’s stack space.
  + Switching the PSP during a context switch allows FreeRTOS to seamlessly switch between tasks without having to save and restore the entire CPU context.

**interrupt vectors :**

* **SVC –** system service call : "svc" may refer to Supervisor Call. In ARM Cortex-M processors, Supervisor Calls are used to invoke operating system services or perform context switches between tasks. In FreeRTOS, SVC instructions are used for system calls, such as task creation, task deletion, or yielding the processor.
* **PendSV –** pended system call (switching context)

PendSV is a special interrupt used for context switching in real-time operating systems like FreeRTOS.

**3. Flash memory :** 6-10kB

**4. RAM memory** : ~0.5kB + task stacks

1. **FreeRTOS distribution :**

**4.1 Definition: FreeRTOS Port**

FreeRTOS can be built with approximately twenty different compilers, and can run on more than thirty different processor architectures. Each supported combination of compiler and processor is considered to be a separate FreeRTOS port.

FreeRTOS can be thought of as a library that provides multi-tasking capabilities to what would otherwise be a bare metal application.

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port. Build the source files as part of your project to make the FreeRTOS API available to your application.

**4.2 FreeRTOSConfig.h**

FreeRTOS is configured by a header file called FreeRTOSConfig.h.

FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example, FreeRTOSConfig.h contains constants such as **configUSE\_PREEMPTION**, the setting of which defines whether the co-operative or pre-emptive scheduling algorithm will be used

**4.3 The Top Directories in the FreeRTOS Distribution**

The first and second level directories of the FreeRTOS distribution are shown

FreeRTOS

│ │

│ ├─Source Directory containing the FreeRTOS source files

│ │

│ └─Demo Directory containing pre-configured and port specific FreeRTOS demo projects

│

FreeRTOS-Plus

│ ├─Source Directory containing source code for some FreeRTOS+ ecosystem components

│ └─Demo Directory containing demo projects for FreeRTOS+ ecosystem

**4.3.1 FreeRTOS Source Files Common to All Ports**

The core FreeRTOS source code is contained in just two C files that are common to all the FreeRTOS ports. These are called **tasks.c, and list.c,** and they are located directly in the FreeRTOS/Source directory.

In addition to these two files, the following source files are located in the same directory:

• **queue.c**

queue.c provides both queue and semaphore services

• **timers.c**

timers.c provides software timer functionality. It need only be included in the build if software timers are actually going to be used.

• **event\_groups.c**

event\_groups.c provides event group functionality. It need only be included in the build if event groups are actually going to be used.

• **croutine.c**

croutine.c implements the FreeRTOS co-routine functionality. It need only be included in the build if co-routines are actually going to be used. Co-routines were intended for use on very small microcontrollers, are rarely used now, and are therefore not maintained to the same level as other FreeRTOS feature

FreeRTOS

│

└─Source

│

├─tasks.c FreeRTOS source file - always required

├─list.c FreeRTOS source file - always required

├─queue.c FreeRTOS source file - nearly always required

├─timers.c FreeRTOS source file - optional

├─event\_groups.c FreeRTOS source file - optional

└─croutine.c FreeRTOS source file - optional

**4.3.2 FreeRTOS Source Files Specific to a Port**

Source files specific to a FreeRTOS port are contained within the FreeRTOS/Source/portable directory. The portable directory is arranged as a hierarchy, first by compiler, then by processor architecture

If you are running FreeRTOS on a processor with architecture ‘architecture’ using compiler ‘compiler’ then, in addition to the core FreeRTOS source files, you must also build the files located in FreeRTOS/Source/portable/[compiler]/[architecture] directory.

FreeRTOS provides five example heap allocation schemes. The five schemes are named heap\_1 to heap\_5, and are implemented by the source files heap\_1.c to heap\_5.c respectively. The example heap allocation schemes are contained in the FreeRTOS/Source/portable/MemMang directory. If you have configured FreeRTOS to use dynamic memory allocation then it is necessary to build one of these five source files in your project, unless your application provides an alternative implementation.

**FreeRTOS**

│

└─**Source**

│

└─**portable** Directory containing all port specific source files

│

├─**MemMang** Directory containing the 5 alternative heap allocation source files

│

├─ **[compiler 1]** Directory containing port files specific to compiler 1

│ │

│ ├─ **[architecture 1]** Contains files for the compiler 1 architecture 1 port

│ ├─ **[architecture 2]** Contains files for the compiler 1 architecture 2 port

│ └─ **[architecture 3]** Contains files for the compiler 1 architecture 3 port

│

└─ [compiler 2] Directory containing port files specific to compiler 2

│

├─ **[architecture 1]** Contains files for the compiler 2 architecture 1 port

├─ [**architecture 2**] Contains files for the compiler 2 architecture 2 port

└─ **[etc.]**

**4.3.3 Include Paths**

FreeRTOS requires three directories to be included in the compiler’s include path. These are:

1. The path to the core FreeRTOS header files, which is always FreeRTOS/Source/include.

2. The path to the source files that are specific to the FreeRTOS port in use. As described above, this is FreeRTOS/Source/portable/[compiler]/[architecture].

3. A path to the FreeRTOSConfig.h header file.

**4.3.4 Header Files**

A source file that uses the FreeRTOS API must include ‘FreeRTOS.h’, followed by the header file that contains the prototype for the API function being used—either ‘task.h’, ‘queue.h’, ‘semphr.h’, ‘timers.h’ or ‘event\_groups.h’.

**FreeRTOSonfig.h**

#ifndef FREERTOS\_CONFIG\_H

#define FREERTOS\_CONFIG\_H

/\*-----------------------------------------------------------

\* Application specific definitions.

\*

\* These definitions should be adjusted for your particular hardware and

\* application requirements.

\*

\* THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE

\* FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.

\*

\* See http://www.freertos.org/a00110.html.

\*----------------------------------------------------------\*/

#include <stdint.h>

extern uint32\_t SystemCoreClock;

#define configCPU\_CLOCK\_HZ (SystemCoreClock)

#define configTICK\_RATE\_HZ ((TickType\_t)1000)

#define configTOTAL\_HEAP\_SIZE ((size\_t)(4096))

#define configMINIMAL\_STACK\_SIZE ((unsigned short)130)

#define configCHECK\_FOR\_STACK\_OVERFLOW 0

#define configMAX\_PRIORITIES (5)

#define configUSE\_PREEMPTION 1

#define configUSE\_TIME\_SLICING 0

#define configIDLE\_SHOULD\_YIELD 1

#define configMAX\_TASK\_NAME\_LEN (10)

/\* Software timer definitions. \*/

#define configUSE\_TIMERS 1

#define configTIMER\_TASK\_PRIORITY (2)

#define configTIMER\_QUEUE\_LENGTH 5

#define configTIMER\_TASK\_STACK\_DEPTH (configMINIMAL\_STACK\_SIZE \* 2)

#define configUSE\_MUTEXES 1

#define configUSE\_RECURSIVE\_MUTEXES 1

#define configUSE\_COUNTING\_SEMAPHORES 1

#define configUSE\_QUEUE\_SETS 1

#define configUSE\_IDLE\_HOOK 1

#define configUSE\_TICK\_HOOK 0

#define configUSE\_MALLOC\_FAILED\_HOOK 0

#define configUSE\_16\_BIT\_TICKS 0

/\* Set the following definitions to 1 to include the API function, or zero

to exclude the API function. \*/

#define INCLUDE\_vTaskPrioritySet 1

#define INCLUDE\_uxTaskPriorityGet 1

#define INCLUDE\_vTaskDelete 1

#define INCLUDE\_vTaskSuspend 1

#define INCLUDE\_vTaskDelayUntil 1

#define INCLUDE\_vTaskDelay 1

#define INCLUDE\_eTaskGetState 1

/\* Cortex-M specific definitions. \*/

#ifdef \_\_NVIC\_PRIO\_BITS

/\* \_\_BVIC\_PRIO\_BITS will be specified when CMSIS is being used. \*/

#define configPRIO\_BITS \_\_NVIC\_PRIO\_BITS

#else

/\* 15 priority levels \*/

#define configPRIO\_BITS 4

#endif

/\* The lowest interrupt priority that can be used in a call to a "set priority"

function. \*/

#define configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY 0xff

/\* The highest interrupt priority that can be used by any interrupt service

routine that makes calls to interrupt safe FreeRTOS API functions. DO NOT CALL

INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER

PRIORITY THAN THIS! (higher priorities are lower numeric values. \*/

#define configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY 10

/\* Interrupt priorities used by the kernel port layer itself. These are generic

to all Cortex-M ports, and do not rely on any particular library functions. \*/

#define configKERNEL\_INTERRUPT\_PRIORITY (configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY << (8 - configPRIO\_BITS))

/\* !!!! configMAX\_SYSCALL\_INTERRUPT\_PRIORITY must not be set to zero !!!!

See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. \*/

#define configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY << (8 - configPRIO\_BITS))

/\* Normal assert() semantics without relying on the provision of an assert.h

header file. \*/

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE\_INTERRUPTS(); for( ;; ); }

/\* Map the FreeRTOS port interrupt handlers to their CMSIS standard names. \*/

#define xPortPendSVHandler PendSV\_Handler

#define vPortSVCHandler SVC\_Handler

#define xPortSysTickHandler OS\_SysTick\_Handler

/\* Include debug event definitions \*/

#include "freertos\_evr.h"

#endif /\* FREERTOS\_CONFIG\_H \*/

1. **Creating a FreeRTOS Project**

**5.1 creating a new project from demo project**

To start a new application from an existing demo project:

1. Open the supplied demo project and ensure that it builds and executes as expected.

2. 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.

3. Delete all the function calls within main(), except prvSetupHardware() and vTaskStartScheduler(),

4. Check the project still builds.

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;**

**}**

**5.2 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:

1. Using your chosen tool chain, create a new project that does not yet include any FreeRTOS source files.

2. Ensure the new project can be built, downloaded to your target hardware, and executed.

3. Only when you are sure you already have a working project, add the FreeRTOS source files detailed in Table 1 to the project.

4. Copy the FreeRTOSConfig.h header file used by the demo project provided for the port in use into the project directory.

5. Add the following directories to the path the project will search to locate header files:

* FreeRTOS/Source/include
* FreeRTOS/Source/portable/[compiler]/[architecture] (where [compiler] and [architecture] are correct for your chosen port)
* The directory containing the FreeRTOSConfig.h header file

6. Copy the compiler settings from the relevant demo project.

7. 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 |

1. **Coding Style**

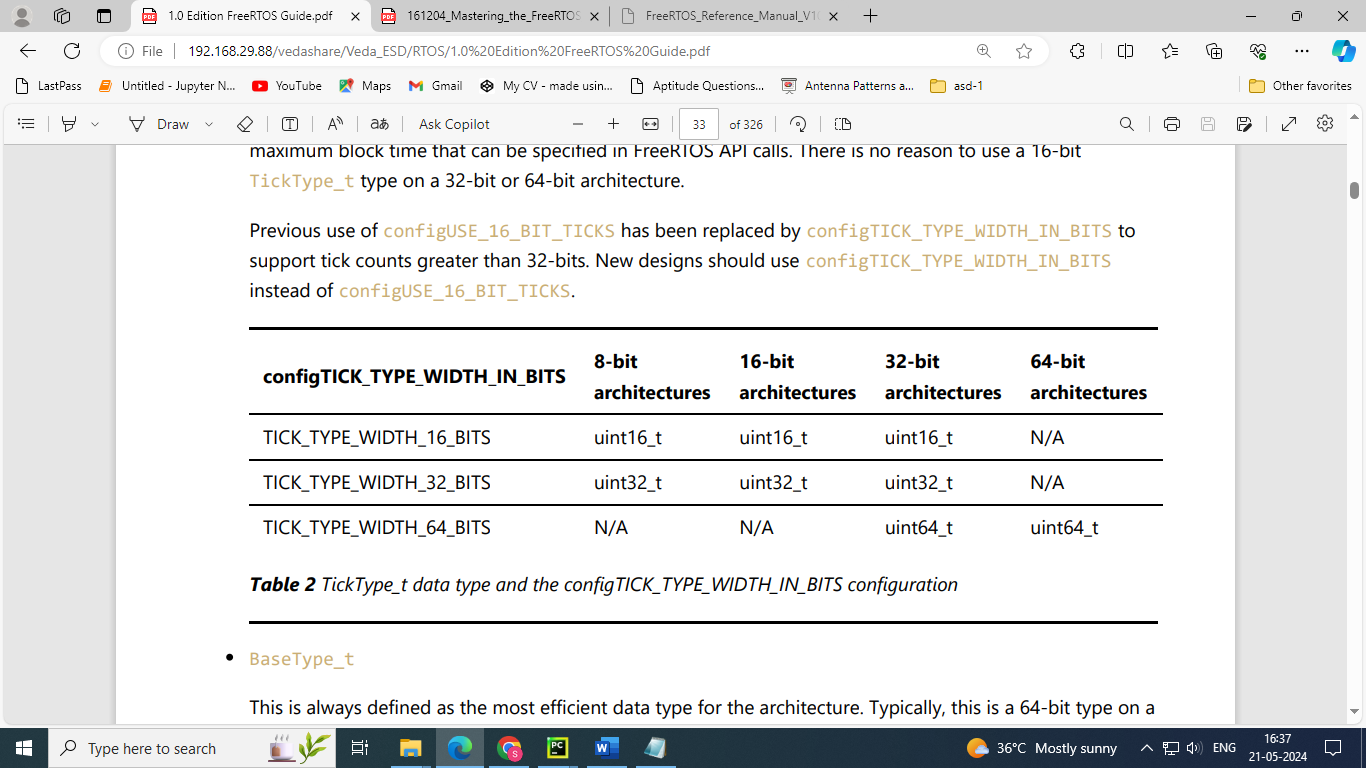
**6.1 Data Types**

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.



**// To use 16-bit tick counts**

**#define configUSE\_16\_BIT\_TICKS 1**

**// To use 32-bit tick counts**

**#define configUSE\_16\_BIT\_TICKS 0**

### Typical Definition of BaseType\_t in portmacro.h

Here is a typical definition of **BaseType\_t** for a 32-bit ARM Cortex-M microcontroller:

/\* Type definitions. \*/

#define portCHAR char

#define portFLOAT float

#define portDOUBLE double

#define portLONG long

#define portSHORT short

#define portSTACK\_TYPE uint32\_t

#define portBASE\_TYPE long

typedef portSTACK\_TYPE StackType\_t;

typedef long BaseType\_t;

typedef unsigned long UBaseType\_t;

In this example:

* **BaseType\_t** is defined as **long**, which is a 32-bit signed integer on most ARM Cortex-M microcontrollers.
* **UBaseType\_t** is the unsigned counterpart, used where unsigned integers are needed.

the FreeRTOS source code explicitly qualifies every use of char with either ‘signed’ or ‘unsigned’, unless the char is used to hold an ASCII character, or a pointer to char is used to point to a string.

Plain int types are never used

**Variable Names**

Variables are prefixed with their type: ‘c’ for char, ‘s’ for int16\_t (short), ‘l’ int32\_t (long), and ‘x’ for BaseType\_t and any other non-standard types (structures, task handles, queue handles, etc.). If a variable is unsigned, it is also prefixed with a ‘u’.

If a variable is a pointer, it is also prefixed with a ‘p’. For example, a variable of type uint8\_t will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.

**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.**

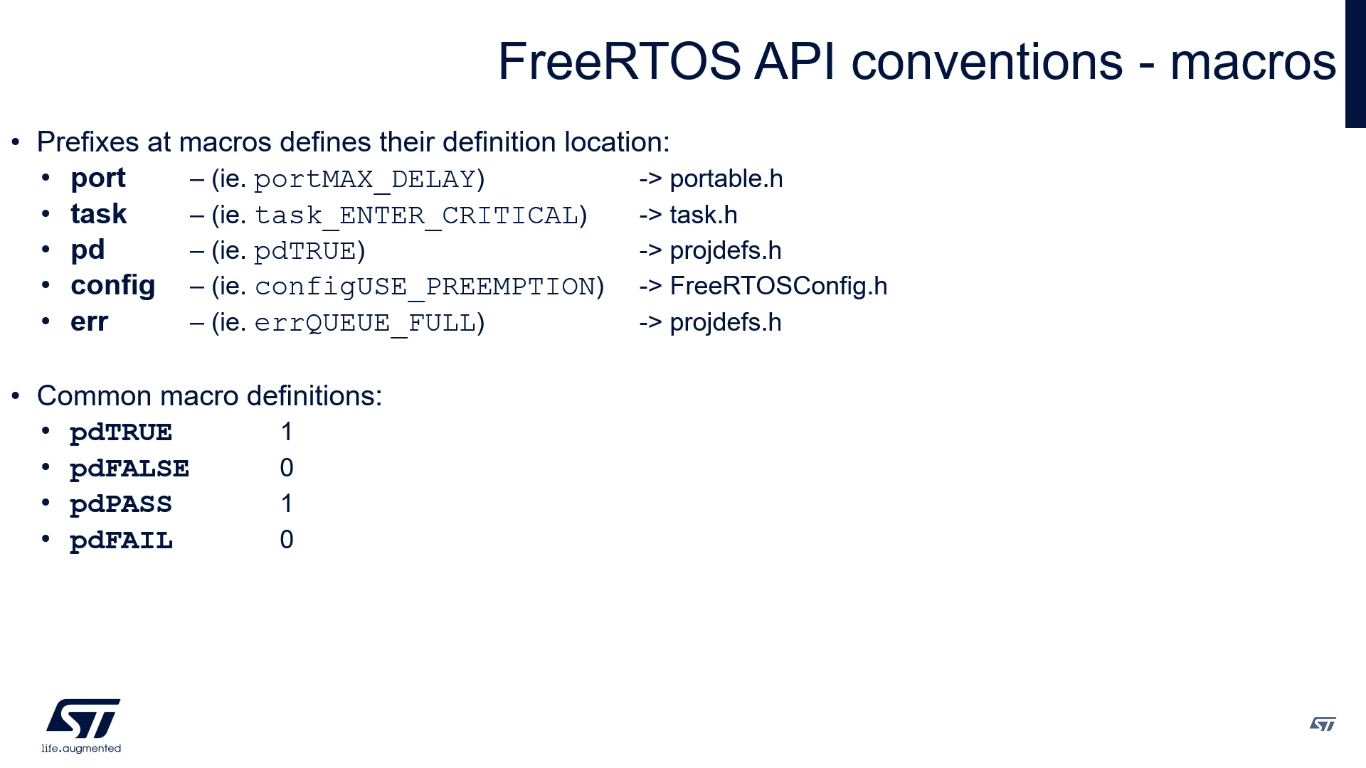
File scope (private) functions are prefixed with ‘prv’.

**Formatting**

One tab is always set to equal four spaces.

**Macro Names**

Most macros are written in upper case, 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.

**Preliminary tips to start FreeRTOS**

Whether you are new to FreeRTOS or an experienced developer, it is always advised to start new developments with [**configASSERT()**](https://www.freertos.org/a00110.html#configASSERT) defined, a [**malloc failed hook**](https://www.freertos.org/a00016.html) implemented, and [**configCHECK\_FOR\_STACK\_OVERFLOW**](https://www.freertos.org/Stacks-and-stack-overflow-checking.html) set to 2.

1. **configASSERT()**

* The semantics of the configASSERT() macro are the same as the standard C **assert()** macro. An assertion is triggered if the parameter passed into configASSERT() is zero
* configASSERT() is called throughout the FreeRTOS source files to check how the application is using FreeRTOS. It is highly recommended to develop FreeRTOS applications with **configASSERT()** defined.
* The example definition (shown at the top of the file and replicated below) calls vAssertCalled(), passing in the file name and line number of the triggering configASSERT() call ( **\_\_FILE\_\_ and \_\_LINE\_\_** are standard macros provided by most compilers). This is just for demonstration as **vAssertCalled()** is not a FreeRTOS function, configASSERT() can be defined to take whatever action the application writer deems appropriate.
* It is normal to define configASSERT() in such a way that it will prevent the application from executing any further. This is for two reasons; stopping the application at the point of the assertion allows the cause of the assertion to be debugged, and executing past a triggered assertion will probably result in a crash anyway.
* Note defining configASSERT() will increase both the application code size and execution time. When the application is stable the additional overhead can be removed by simply commenting out the configASSERT() definition in FreeRTOSConfig.h.

**/\* Define configASSERT() to call vAssertCalled() if the assertion fails. The assertion**

**has failed if the value of the parameter passed into configASSERT() equals zero. \*/**

**#define configASSERT ( x ) if( ( x ) == 0 )** **vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )**

**void vAssertCalled(const char \*pcFile, unsigned long ulLine) {**

**/\* Handle assertion failure \*/**

**printf("ASSERTION FAILED: File %s, Line %lu\n", pcFile, ulLine);**

**/\* Optionally halt the system or take other corrective actions \*/**

**for (;;); // Halting the system in this example**

**}**

* **pcFile**: A pointer to a string containing the name of the source file where the assertion failed.
* **ulLine**: The line number within the source file where the assertion failed.
* If running FreeRTOS under the control of a debugger, then configASSERT() can be defined to just disable interrupts and sit in a loop, as demonstrated below. That will have the effect of stopping the code on the line that failed the assert test - pausing the debugger will then immediately take you to the offending line so you can see why it failed.

**/\* Define configASSERT() to disable interrupts and sit in a loop. \*/**

**#define configASSERT ( x ) if( ( x ) == 0 ) { taskDISABLE\_INTERRUPTS(); for( ;; ); }**

* The purpose of **taskDISABLE\_INTERRUPTS()** is to ensure that a block of code is executed atomically, without interruption by interrupts. This is essential in scenarios where data consistency or synchronization between tasks is critical. By disabling interrupts, you can prevent context switches and interrupt service routines (ISRs) from running during the execution of the protected code.

1. **Malloc Failed Hook Function**

* The memory allocation schemes implemented by [heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c and heap\_5.c](https://www.freertos.org/a00111.html) can optionally include a malloc() failure hook (or callback) function that can be configured to get called if pvPortMalloc() ever returns NULL.
* Defining the malloc() failure hook will help identify problems caused by lack of heap memory - especially when a call to pvPortMalloc() fails within an API function.
* **pvPortMalloc()** is a memory allocation function provided by FreeRTOS that allows dynamic allocation of memory from the FreeRTOS heap. It is similar in purpose to the standard **malloc()** function provided by the C runtime library, but it is designed to work within the FreeRTOS environment and is typically used for allocating memory for tasks, queues, and other RTOS objects.

Example usage of pvPortMalloc():

**#include "FreeRTOS.h"**

**void \*pvMemory;**

**size\_t xSize = 100; // Size of the memory block in bytes**

**pvMemory = pvPortMalloc(xSize);**

**if (pvMemory != NULL) {**

**// Memory allocation successful, use the memory**

**} else {**

**// Memory allocation failed, handle the error**

**}**

* The malloc failed hook will only get called if

configUSE\_MALLOC\_FAILED\_HOOK is set to 1 within FreeRTOSConfig.h. When this is set the application must provide the hook function with the following prototype:

**#define configUSE\_MALLOC\_FAILED\_HOOK 1**

**#define configTOTAL\_HEAP\_SIZE ( ( size\_t ) ( 10 \* 1024 ) )**

**// Example: 10 KB heap**

**void vApplicationMallocFailedHook( void ) {**

**/\* Handle memory allocation failure \*/**

**printf("Memory allocation failed!\n");**

**taskDISABLE\_INTERRUPTS();**

**for (;;) {**

**// Optionally log the error, reset the system, or take other actions**

**}**

**}**

* + The **vApplicationMallocFailedHook()** function is a hook provided by FreeRTOS that allows you to handle situations where memory allocation fails, typically due to insufficient heap memory. This function is called by FreeRTOS when a call to **pvPortMalloc()** fails to allocate memory.

1. [**configCHECK\_FOR\_STACK\_OVERFLOW**](https://www.freertos.org/Stacks-and-stack-overflow-checking.html)
   * Each task maintains its own stack. If a task is created using **[xTaskCreate()](https://www.freertos.org/a00125.html)** then the memory used as the task's stack is allocated automatically from the [FreeRTOS heap](https://www.freertos.org/a00111.html), and dimensioned by a parameter passed to the xTaskCreate() API function. If a task is created using **[xTaskCreateStatic()](https://www.freertos.org/xTaskCreateStatic.html)** then the memory used as the task's stack is pre-allocated by the application writer. Stack overflow is a very common cause of application instability

* The application must provide a stack overflow hook function if **configCHECK\_FOR\_STACK\_OVERFLOW** is not set to 0. The hook function must be called **vApplicationStackOverflowHook(),** and have the prototype below:

**void vApplicationStackOverflowHook(TaskHandle\_t xTask, char \*pcTaskName) {**

**// Handle stack overflow here**

**printf("Stack overflow detected in task %s\n", pcTaskName);**

**taskDISABLE\_INTERRUPTS();**

**for(;;); // Halt the system**

**}**

* The xTask and pcTaskName parameters pass to the hook function the handle and name of the offending task respectively. Note however, depending on the severity of the overflow, these parameters could themselves be corrupted, in which case the pxCurrentTCB variable can be inspected directly.
* **pxCurrentTCB** is a pointer used internally by FreeRTOS to keep track of the currently executing task. TCB stands for Task Control Block, which is a data structure used by FreeRTOS to store information about a task. Each task in FreeRTOS has its own TCB, and **pxCurrentTCB** points to the TCB of the task that is currently running.
  + To further know about this macro we need to know about the context switching and states of the tasks

**Stack Overflow Detection - Method 1**

* It is likely that the stack will reach its greatest (deepest) value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.
* This method is quick but not guaranteed to catch all stack overflows. Set configCHECK\_FOR\_STACK\_OVERFLOW to 1 to use this method.

**Stack Overflow Detection - Method 2**

* When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the RTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.
* This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.
* Set configCHECK\_FOR\_STACK\_OVERFLOW to 2 to use this method.

**Stack Overflow Detection - Method 3**

* Set configCHECK\_FOR\_STACK\_OVERFLOW to 3 to use this method.
* This method is available only for selected ports. When available, this method enables ISR stack checking.

.