ARDUINO FREE RTOS

https://www.youtube.com/watch?v=F321087yYy4&list=PLEBQazB0HUyQ4hAPU1cJED6t3DU0h34bz

An embedded system is a computer system that is integrated into a device or a product for performing specific functions. These systems are used in various applications, including automotive, medical devices, aerospace, and consumer electronics. In embedded systems, real-time tasks are critical, and the timing of these tasks plays an important role.

Real-time tasks are those tasks that require a guaranteed response time to an event. These tasks need to be completed within a specific time limit; otherwise, the system may fail or produce incorrect results. In an embedded system, real-time tasks can be periodic, such as sampling a sensor value at a fixed rate, or aperiodic, such as responding to an external event.

An RTOS (Real-Time Operating System) is an operating system that is specifically designed to handle real-time tasks in embedded systems. An RTOS is different from a general-purpose operating system, such as Windows or Linux, which are designed to run on desktop or server computers. An RTOS is optimized for running applications with precise timing and a high degree of reliability.

One of the critical features of an RTOS is its ability to provide time determinism. **Time determinism means that the response time to any event is always constant, so it can be guaranteed that a particular event will occur at a fixed time. An RTOS achieves time determinism by using various scheduling algorithms, such as priority-based scheduling, to ensure that high-priority tasks are executed before low-priority tasks.**

Another important feature of an RTOS is its ability to support multi-tasking with a single core. Multi-tasking is the ability to run multiple tasks simultaneously on a single processor. **An RTOS achieves multi-tasking by using a scheduler that allocates processor time to each task based on their priority and the scheduling algorithm.** The scheduler ensures that each task is given a fair share of the processor time, and higher-priority tasks are executed before lower-priority tasks.

An RTOS also provides mechanisms for synchronization and communication between tasks. In an embedded system, **multiple tasks may need to access the same hardware resource, such as a sensor or an actuator. To avoid conflicts and ensure data integrity, an RTOS provides synchronization primitives such as mutexes, semaphores, and critical sections.** These mechanisms allow tasks to access shared resources in a coordinated and controlled way.

Finally, an RTOS provides a set of services and APIs for application development. These APIs include functions for task management, memory management, and inter-task communication. These APIs simplify the development of real-time applications and provide a standardized interface for application developers.

In summary, an RTOS is a critical component of many embedded systems where real-time tasks are essential. It provides time determinism, multi-tasking, synchronization, and communication mechanisms that enable the development of real-time applications with precise timing and a high degree of reliability. The use of an RTOS also simplifies the development of real-time applications by providing a standardized interface for application developers to access system services and resources.

💡 API stands for Application Programming Interface. It is a set of protocols, routines, and tools for building software applications. An API defines how software components should interact with each other, providing a standardized interface that simplifies the development of software applications.

APIs are used to abstract complex functionality and provide a simpler interface for developers to interact with. They allow developers to access the functionality of an existing software system without having to understand its underlying implementation details. APIs provide a way for different software components to communicate with each other, enabling developers to build complex software applications by combining different software components.

For example, suppose you want to build a weather application that displays current weather data for a specific location. Instead of developing your own weather data collection system, you can use an API provided by a weather service provider. The API would allow you to access the weather data collected by the provider's system, enabling you to develop your application quickly and easily.

APIs can be designed for different types of software components, including operating systems, databases, web services, and applications. They can be accessed using different programming languages, such as Python, Java, and C++, making them accessible to developers with different skill sets.

In summary, an API is a set of protocols and tools that enable developers to access the functionality of an existing software system, providing a standardized interface that simplifies the development of software applications

💡 An 8-bit MCU (Microcontroller Unit) is a type of microcontroller that has an 8-bit CPU (Central Processing Unit). An MCU is a small computer on a single integrated circuit that is designed to control a specific device or system. It typically includes a CPU, memory, input/output peripherals, and other components that are required to control the device or system.

The 8-bit MCU is one of the earliest and most widely used types of MCUs. It was first introduced in the 1970s and has since been used in various applications, including consumer electronics, automotive, and industrial control systems. The 8-bit MCU is known for its low cost, low power consumption, and ease of use, making it ideal for simple applications that require basic processing capabilities.

The CPU in an 8-bit MCU can process data in 8-bit chunks, meaning that it can manipulate data that is up to 8 bits in size. This limits the maximum amount of data that can be processed by the MCU but makes it suitable for applications that do not require complex processing capabilities. The low cost and simplicity of 8-bit MCUs make them suitable for applications that require a low-cost solution, such as consumer electronics devices, sensors, and small appliances.

In summary, an 8-bit MCU is a type of microcontroller that has an 8-bit CPU and is widely used in various applications, including consumer electronics, automotive, and industrial control systems. It is known for its low cost, low power consumption, and ease of use, making it ideal for simple applications that require basic processing capabilities.

HOW RTOS WORKS??

TASKS

💡 Task is a piece of code that is schedulable on the CPU to execute. So, if you want to perform some task, then it should be scheduled using kernel delay or using interrupts. This work is done by Scheduler present in the kernel. In a single-core processor, the scheduler helps tasks to execute in a particular time slice but it seems like different tasks are executing simultaneously. Every task runs according to the priority given to it.

An RTOS (Real-Time Operating System) works by managing the execution of multiple tasks in a real-time environment. A task is a piece of code that needs to be executed by the CPU, and the RTOS manages the scheduling and execution of these tasks.

The RTOS kernel provides services such as scheduling, synchronization, and inter-task communication to manage the execution of tasks. The scheduler is responsible for assigning CPU time to each task based on its priority and the scheduling algorithm. The priority of a task is determined by its importance and urgency in the system. Higher priority tasks are executed before lower priority tasks.

In a single-core processor, the scheduler uses a time-slicing algorithm to execute tasks in a particular time slice. This gives the impression that different tasks are executing simultaneously. Each task runs for a predetermined amount of time, known as its time slice or quantum, before being preempted by the scheduler and replaced with another task.

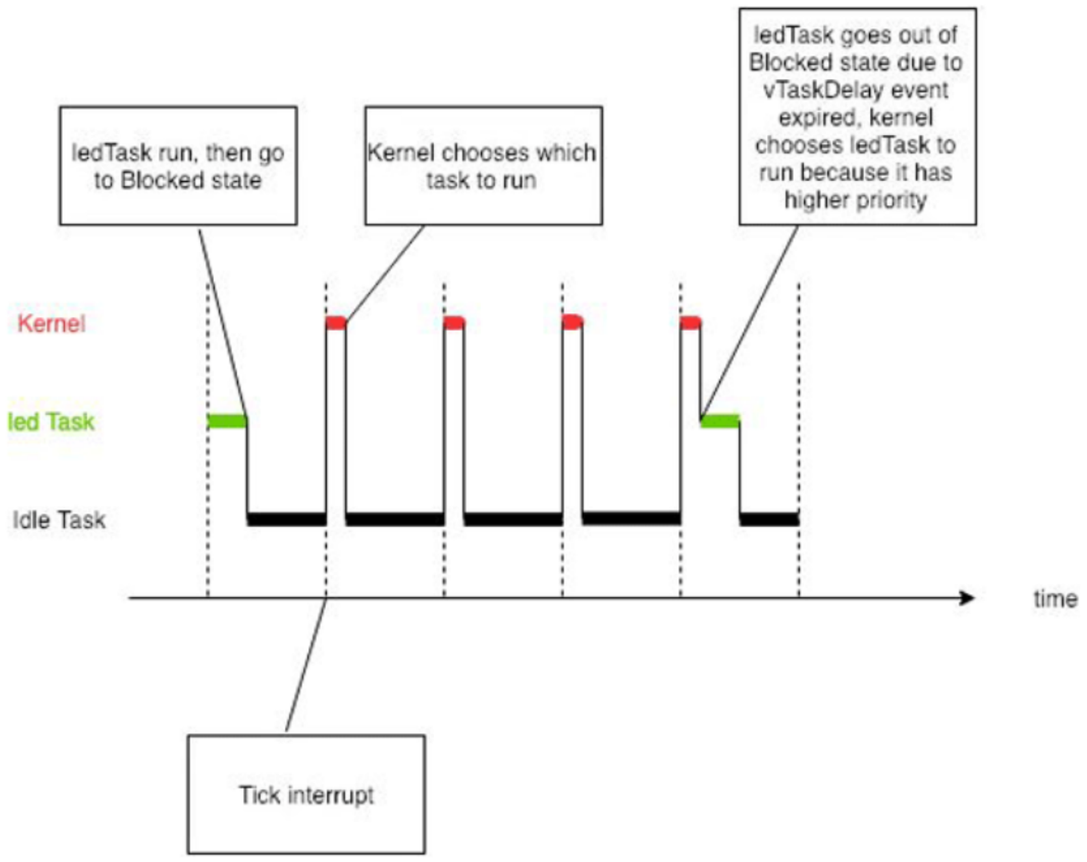
The RTOS also provides synchronization mechanisms to prevent conflicts and ensure data integrity between tasks that share system resources. These mechanisms include semaphores, mutexes, and critical sections. They allow tasks to access shared resources in a coordinated and controlled way.

An RTOS provides several mechanisms for inter-task communication, including:

1. Queues: A queue is a data structure that allows tasks to send and receive messages or data. Tasks can add messages to the queue or remove them from the queue. Queues can be used for synchronization or to pass data between tasks.
2. **Semaphores: A semaphore is a synchronization mechanism that *allows tasks to signal each other.* Tasks can wait for a semaphore to become available or signal a semaphore when they are done with a resource.**

3. **Mutexes: A mutex is a synchronization mechanism that allows tasks to protect shared resources from concurrent access. Only one task can acquire a mutex at a time, which ensures that only one task can access a shared resource at a time.**
4. Event flags: An event flag is a synchronization mechanism that ***allows tasks to wait for specific events to occur.*** Tasks can wait for event flags to be set or clear, which signals that a particular event has occurred.
5. Pipes: A pipe is a unidirectional communication mechanism that **allows tasks to send data from one task to another**. The data flows in one direction, from the sending task to the receiving task.

These mechanisms allow tasks to communicate and synchronize their activities in a coordinated manner. They are essential for building complex embedded systems that require multiple tasks to work together to achieve a common goal.


ledTask run, then go to Blocked state
Kernel chooses which task to run
ledTask goes out of Blocked state due to vTaskDelay event expired, kernel chooses ledTask to run because it has higher priority
Kernel
led Task
Idle Task
time
Tick interrupt

Suppose we want to create a task for LED blinking with a one-second interval and put this task on the highest priority. We would first define the LED blinking task and its associated parameters, including its priority level and time interval.

Once we create the LED blinking task, we would add it to the list of tasks managed by the RTOS kernel. The kernel would then schedule the task for execution based on its priority level. Since the LED blinking task has the highest priority, it would be scheduled to run first.

When the LED blinking task is executed, it would perform its task of blinking the LED and then enter the blocked state by calling vTaskDelay() for a one-second interval. This means that the task is not eligible to be scheduled for execution until the one-second interval has elapsed.

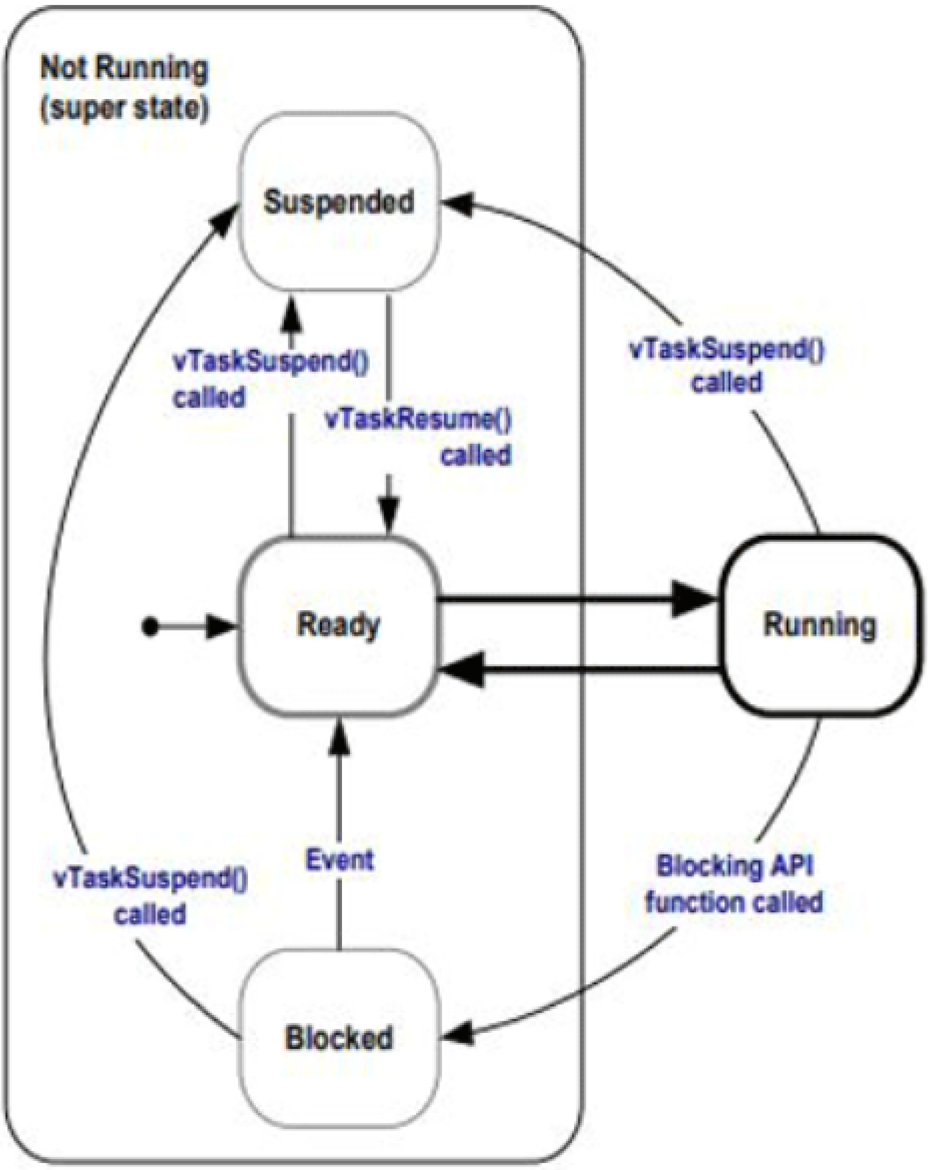
While the LED blinking task is in the blocked state, the kernel continues to choose which task to run based on their priority levels. If there are other tasks with higher priority levels, they will be scheduled to run instead of the LED blinking task.

After the one-second interval has elapsed, the LED blinking task goes out of the blocked state due to the vTaskDelay event expiring. At this point, the kernel chooses the LED blinking task to run again because it has the highest priority level. The LED blinking task then performs its task of blinking the LED again and enters the blocked state again for another one-second interval.

This process continues, with the LED blinking task being scheduled for execution whenever its vTaskDelay event expires and its priority level is the highest among all tasks in the system.

Apart from the LED task, there will be one more task which is created by the kernel, it is known as an **idle task**. The idle task is created when no task is available for execution. This task always runs on the lowest priority i.e. 0 priority. If we analyze the timing graph given above, it can be seen that execution starts with an LED task and it runs for a specified time then for remaining time, the idle task runs until a tick interrupt occurs. Then kernel decides which task has to be executed according to the priority of the task and total elapsed time of the LED task. When 1 second is completed, the kernel chooses the led task again to execute because it has a higher priority than the idle task, we can also say that the LED task preempts the idle task. If there are more than two tasks with the same priority then they will run in round-robin fashion for a specified time.

Below the state diagram as it shows the **switching of the non-running task into running state.**



Every newly created task goes in Ready state (part of not running state). If the created task (Task1) has the highest priority than other tasks, then it will move to running state. If this running task preempts by the other task, then it will go back to the ready state again. Else if task1 is blocked by using blocking API, then CPU will not engage with this task until the timeout defined by the user.

If Task1 is suspended in running state using Suspend APIs, then Task1 will go to Suspended state and it is not available to the scheduler again. If you resume Task1 in the suspended state then it will go back to the ready state as you can see in the block diagram.

This is the **basic idea of how Tasks run and change their states**. In this tutorial, we will implement two tasks in Arduino Uno using FreeRTOS API.

Frequently used terms in RTOS

1. **Task:** As explained earlier, a task is a unit of code that can be executed by the CPU. The RTOS schedules tasks based on their priority and ensures that they run in a timely manner.
2. **Scheduler:** The scheduler is a core component of the RTOS that decides which task to execute next. It is responsible for managing the task scheduling algorithm and maintaining the ready-to-run task list. The scheduler runs periodically and selects the next highest priority task from the list of ready-to-run tasks.
3. **Preemption:** Preemption occurs when the RTOS interrupts a currently running task to switch to a higher priority task. This ensures that high-priority tasks are executed in a timely manner, even if lower-priority tasks are still running. **Preemption can occur due to an interrupt, a higher-priority task becoming ready to run, or a time-slice expiration**.
4. **Context Switching**: When the RTOS switches from one task to another, it needs to save the context of the current task (such as its registers, stack pointer, and program counter) and restore the context of the new task. This process is known as context switching. **Context switching can be a time-consuming operation,** especially on low-end microcontrollers with limited memory and processing power. **Therefore, RTOS designs typically try to minimize the frequency of context switching to improve system performance**.
5. **Types of Scheduling policies:**
    - **Preemptive Scheduling**: In this type of scheduling policy, the tasks are allocated equal time slices, regardless of their priority. The CPU time is divided into fixed time slices, and each task is allowed to run for the time slice. If a task is not finished, it is preempted and another task is scheduled to run. This approach ensures that all tasks get a fair share of the CPU time, but may not be optimal for real-time applications with strict timing requirements.

- **Priority-based Preemptive:** In this type of scheduling policy, the tasks are allocated priorities, and the CPU always runs the highest priority task that is ready to run. If a higher priority task becomes ready to run, the currently running task is preempted and the higher priority task is scheduled to run. This approach ensures that high priority tasks get executed in a timely manner, but may cause lower priority tasks to starve if they never get a chance to run.
- **Cooperative Scheduling:** In this type of scheduling policy, the running task continues to run until it voluntarily yields the CPU to another task. The task only relinquishes control of the CPU when it explicitly calls a yield function. This approach is simple and efficient, but requires the tasks to be well-behaved and cooperative.

6. **Kernel Objects:** RTOS provides kernel objects to enable synchronization between tasks and provide inter-task communication. Some commonly used kernel objects include:

    - Events: Signals the occurrence of an event to one or more waiting tasks.
    - Semaphores: A synchronization mechanism that allows tasks to wait for a shared resource to become available.
    - Queues: A buffer that allows tasks to send and receive messages or data.
    - Mutexes: A synchronization mechanism that ensures only one task has access to a shared resource at a time.
    - Mailboxes: A specialized queue that allows only one message at a time, typically used for inter-task communication.

    These kernel objects are essential for the proper functioning of an RTOS and provide a way for tasks to communicate and synchronize their activities.