ESP32 PROGRAMMING - HANDS-ON WORKSHOP

Kunaal Kiran Kumar

Chitkara University, Punjab

AGENDA

FOUNDATION

- Controller architecture and communication systems
- Introduction to ESP32 – IoT Development
- Introduction to Real-Time Operating Systems with ESP32

INTERMEDIATE

- Introduction to RTOS Kernel Objects

ADVANCED

- Timers and Interrupts in ESP32 with RTOS.
- Infinite delay, Deadlock and Starvation, Priority Inversion

FOUNDATION - CONTROLLER ARCHITECTURE & COMMUNICATION SYSTEMS

MICROCONTROLLERS – LET'S DISCUSS

Let's start the discussion with a desktop/laptop.

When you buy a new PC, what all specifications do you check for?

RAM, Clock speed, Storage, GPU, No. of processor cores and many more

When we say 32-bit or 64-bit system, what does it mean? How does the word length affect the performance of the controller?

What is the purpose of an operating system in a PC?

Manage multiple tasks, manage system and application software, ensure optimum performance of all applications, manage resources and memory

MICROCONTROLLERS – LET'S DISCUSS

What is a microcontroller?

What are the main components of a microcontroller?

1. ***Volatile memory***
2. ***Non-volatile memory***
3. ***CPU (Processor: Control unit + ALU)***
4. ***Registers***
5. ***Bus control (Boot, Reset, etc.)***
6. ***Interrupt control***
7. ***DMA control***
8. ***Timers & Oscillators***
9. ***I/O & Communication ports***

MICROCONTROLLERS – LET'S DISCUSS

- How do we tell a microcontroller what tasks it has to do?

  We write a program and store it in the read-only memory of the controller. The controller performs the tasks accordingly.

- Why do we need external I/O's in a microcontroller?
- When you give a signal to an input pin, how does the controller interpret it? Let's discuss in terms of a DHT Sensor. How does the controller know the value it is receiving is a temperature sensor value?

- Why do we need communication with external devices?

1. ***Sending and receiving data to other controllers***
2. ***Read/write operations with sensors and peripherals/displays etc.***
3. ***Connecting an independent autonomous system to the outside world***

MICROCONTROLLERS – LET'S DISCUSS

What are the steps involved between the C/C++ Program you wrote and the microcontroller processing the code to execute the tasks?

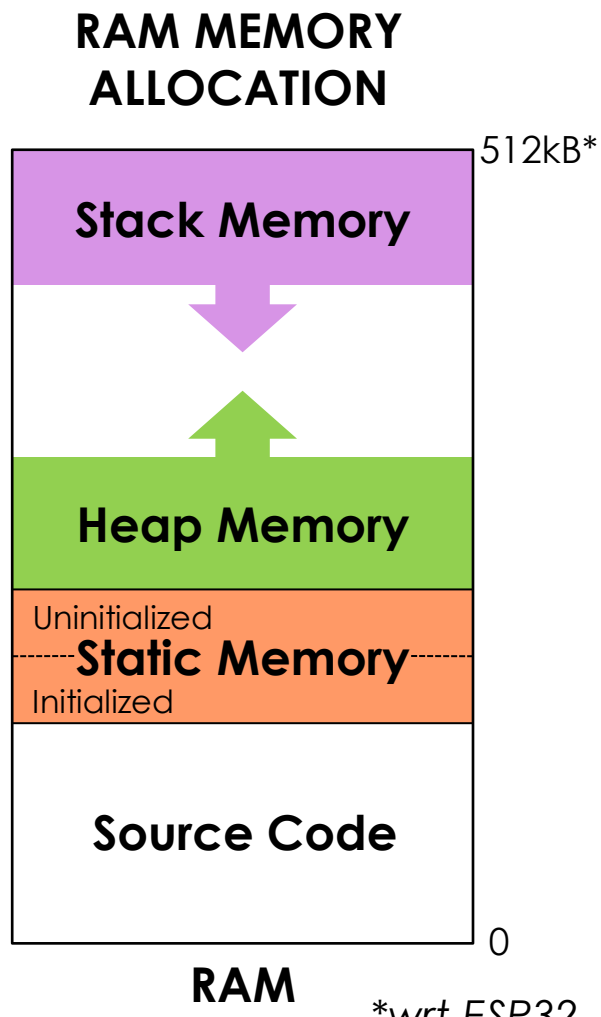
MICROCONTROLLERS – LET'S DISCUSS

```c
#include <stdio.h>

int global_var; // global variable -> static
static int static_global_var; // static global variable -> static
int global_var_init = 10; // global variable with initial value -> initialized static
int *global_ptr; // global pointer -> static

void function()
{
    int local_var; // local variable -> stack
    static int static_local_var; // static local variable -> static
    int local_var_init = 10; // local variable with initial value -> stack
    int i = malloc(sizeof(int)); // local pointer -> heap
}

void main()
{
    uint8_t x[10]; // array with fixed size and initialized to 0 -> stack
    uint8_t *y; // local pointer to a byte-sized value -> stack
    memcpy(y, x, 10); // Initializing a 10-byte memory block with 0 -> heap
}
```



MICROCONTROLLERS – LET'S DISCUSS

Let's see how data is stored in Flash. When we write to the flash memory of a microcontroller, the data remains even when the device is powered off. Let's see how it works. We are taking the 4MB flash of ESP32 as an example.

The ESP32 uses SPI protocol to send and receive data to the flash memory. It consists of a partition table that divides the flash memory as follows:

1.	**nvs**: Non-Volatile Storage	0x5000 ⟶ 20kB	**data**
2.	**otadata**: Holds app_version	0x2000 ⟶ 8kB	**data**
3.	**factory**: Factory app	Based on uploaded firmware	**app**
4.	**ota_1, ota_2, and so on**: Multiple firmware versions stored		**app**
5.	**spiffs**: Can be used to store data as a file (.txt, .csv, etc.)		**data**

MICROCONTROLLERS – LET'S DISCUSS

- What are the most popular communication protocols used in microcontrollers?

Wired

1. ***UART***
2. ***I2C***
3. ***SPI***

Wireless

1. ***Bluetooth***
2. ***Wi-Fi***
3. ***SIM-GSM***

FOUNDATION - ESP32 PROGRAMMING AND IOT DEVELOPMENT

ESP32 – OVERVIEW

- Manufactured by Espressif Systems
- Low-cost, low-power with a 32-bit dual-core Tensilica Xtensa LX6 processor max clock speed of 240 MHz
- It has a 512kB RAM and 4MB Flash memory
- Built-in components: Wi-fi, Bluetooth, ADC, DAC, Timers
- Operational voltage at 3.3V, built-in pull-up and pull-down resistors
- Is designed to handle up to 64 interrupts (32 on each core)
- High number of configurable GPIO's, supports major communication protocols such as UART, I2C and SPI with designated ports that are configured for the same, moreover, it has PWM compatible GPIO's
- Can be programmed in embedded C/C++ language (or python) and supports RISC architecture.

REQUIREMENTS

Note: Students can work as individuals or in teams of 2-3 people.
Each team must have:

1. ESP32 Microcontroller + Micro-USB Data cable
2. At least 3 sensors / devices (Ex: DHT sensor, I2C LCD display, Ultrasonic sensor, etc.)
3. Jumper wires & breadboard for connection

SETUP

- Download and install the Arduino IDE from the official Arduino website. Launch the Arduino IDE application.
- Open **File > Preferences**
- In the "**Additional Boards Manager URLs**", type or paste this link: https://dl.espressif.com/dl/package_esp32_index.json
- Click **OK** to save the configuration.
- Next, go to **Tools > Board > Boards Manager** and search and download ESP32 board package from there. After that, select **Tools > Board** and select ***ESP32 Dev Module*** under the newly appeared 'ESP32' category, and you're all set.
- Simply connect the device to your PC via a USB cable and go to **Tools > Port** and select the available COM port to connect to it to the Arduino IDE.
- Your Arduino IDE is now all set to program the ESP32.

ABOUT ARDUINO IDE DEVELOPMENT

1. Superloop (setup and loop functions) architecture
2. Supports major data types of C/C++
3. Has control structures like loops and decision statements
4. Functions - built-in as well as user-defined
5. Libraries - a wide range of libraries/drivers in **Tools > Manage Libraries**
6. Macros, user-defined data types, and // comments for /* explanation */

TO UPLOAD A SKETCH TO ESP32

1. Write a sketch in .ino for our ESP32 microcontroller
2. On the top left corner, use the verify button to compile the sketch, and the upload button to write the sketch to the ESP32
3. Once uploaded, the ESP32 should work as per to the uploaded code

LIST OF PROGRAMS TO GET STARTED WITH ESP32 & IOT DEVELOPMENT

1. Blink the built-in LED
2. Serial Read & Write
3. Read/write data using a GPIO pin (sensor)
4. Connect to a Bluetooth device
5. Create a HTTP Server on ESP32 using Wi-Fi
6. Use your ESP32 as an AP to send/receive data to a HTTP server
7. Create your first IoT application with ESP32

FOUNDATION - INTRODUCTION TO REAL-TIME OPERATING SYSTEMS

WHAT IS AN OPERATING SYSTEM?

GPOS

1. Human Interaction is involved, and for the best user experience, it is prioritized to cater to the same.
2. It ensures responsiveness to human interaction and may delay meeting timing deadlines of other background tasks.
3. Since most operating systems are fast, the delays may not be noticeable.
4. Scheduler is non-deterministic, which means we have no way of knowing which task will run for how long.
5. In some cases, for example engine controllers, or medical devices, missing a timing deadline may cause trouble.

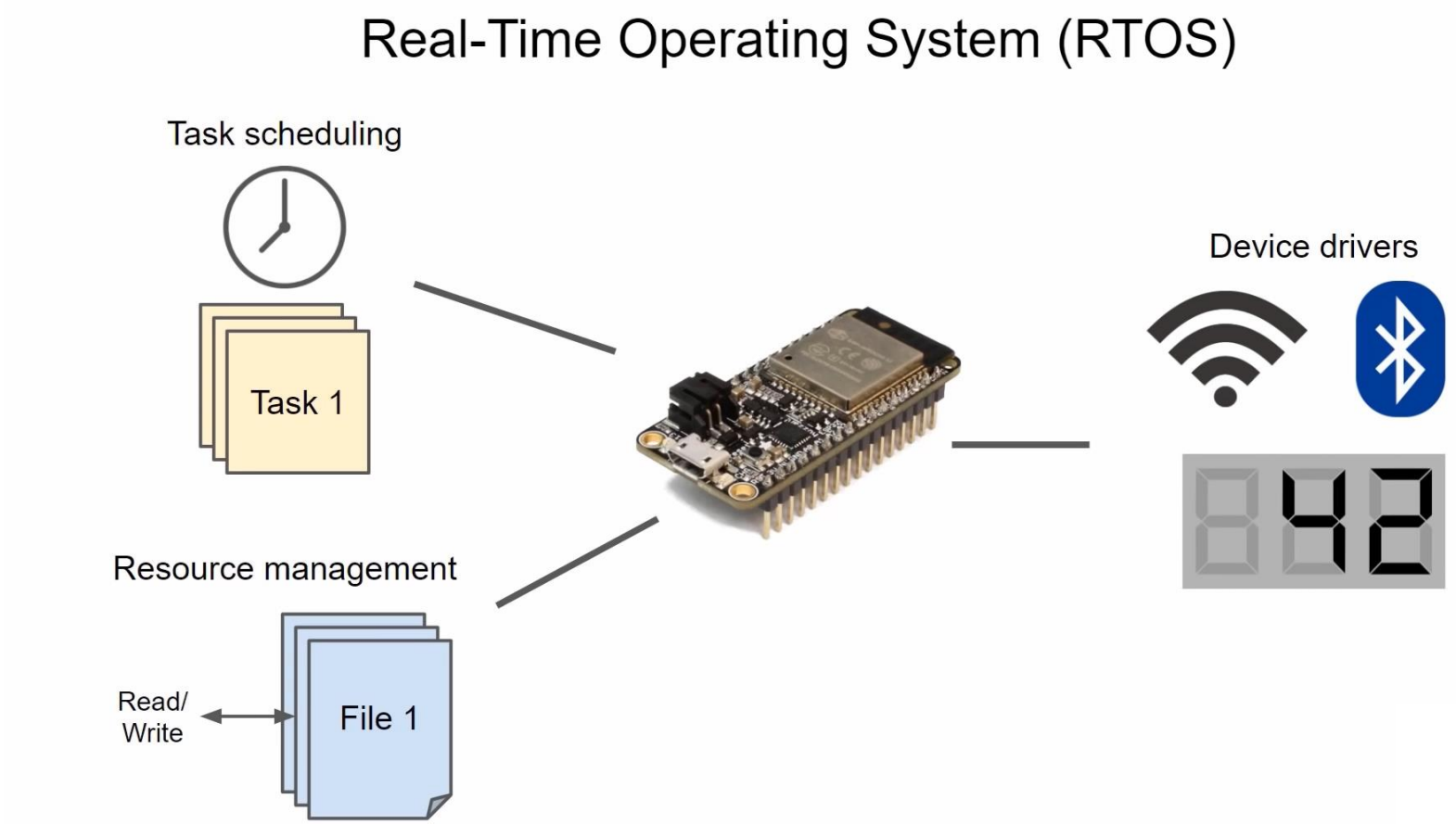
RTOS

1. This is where RTOS comes to place, when we have very specific needs, limited set of tasks, and we need to meet very specific deadlines.
2. As you can see, the drivers for GPOS may differ from RTOS because of the different use cases.

FREERTOS REAL-TIME OPERATING SYSTEMS

FreeRTOS is one of the most extensively used OS for Embedded devices, and as of 2017, Amazon has taken over this open-source project. There are other options and new comers like the Zephyr RTOS backed by the Linux Foundation.

WHAT IS AN OPERATING SYSTEM?



SUPERLOOP ARCHITECTURE

Superloop AKA bare metal approach is a simple approach, which I am sure most of you are familiar with.

Advantages of Superloop:

- Easy to write, easy to debug
- For a limited number of tasks, super loop is the best option
- Tasks that require to meet strict timing deadlines can be attached to interrupts

Disadvantages:

Since the tasks are arranged in an order, any delay in one task will affect the other tasks as well.

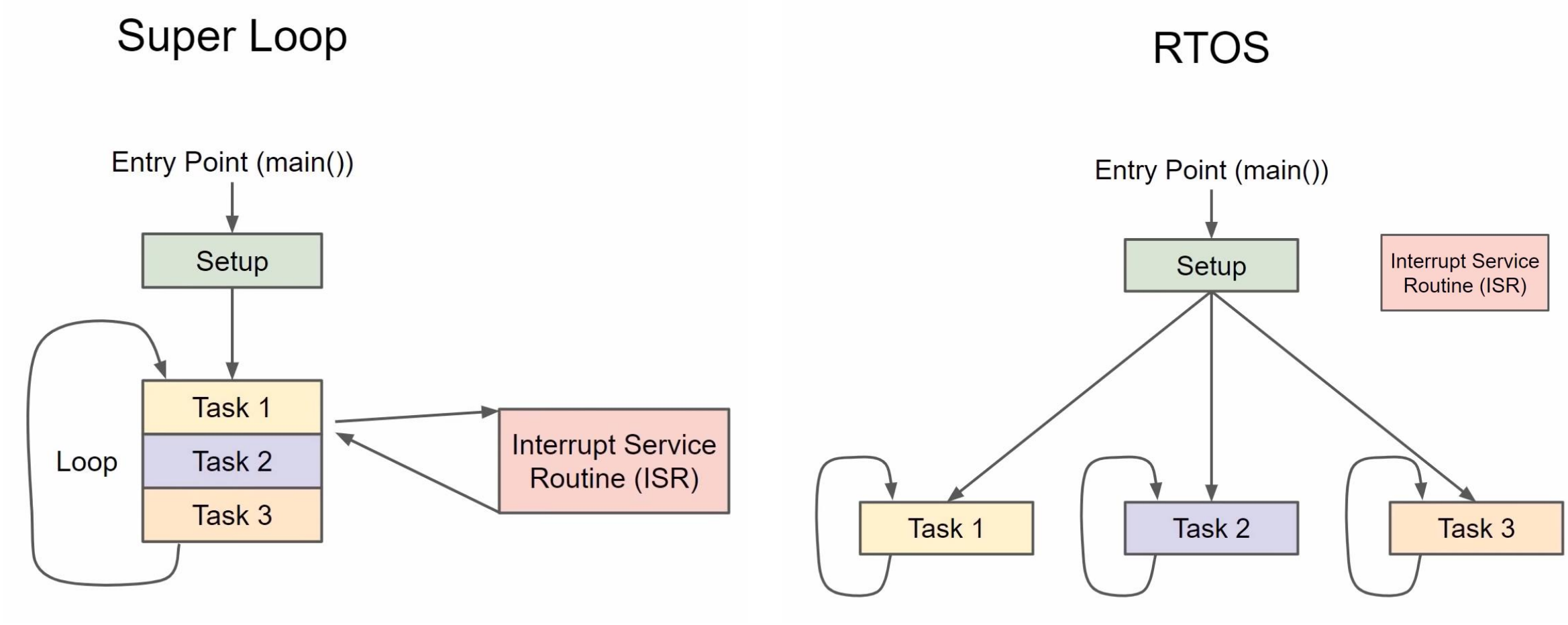
RTOS ARCHITECTURE

For multi-core processors, multiple tasks can run concurrently, which is true in the case of ESP32, however, many other controllers only have one processor.

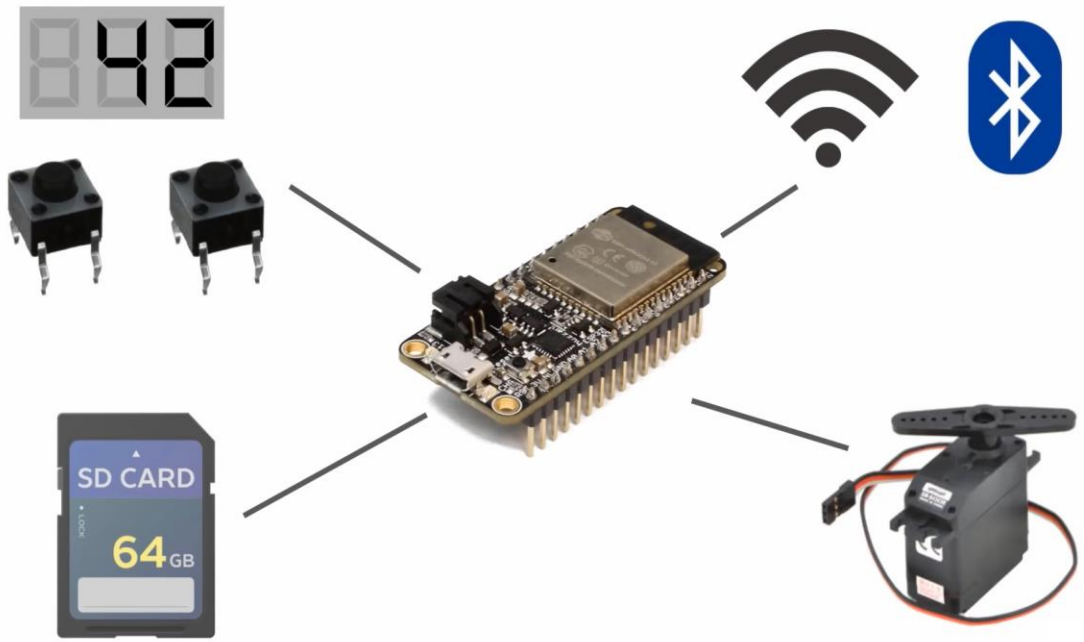
Therefore, RTOS allows the CPU to split its time between the different tasks. In this case, some tasks can be given higher priority, to allow them more CPU time.

This is decided by you, as a developer, to choose which task gets which priority. Note: RTOS is also capable of handling interrupts and ISRs.

SUPERLOOP VS RTOS



UNLOCK MULTITASKING CAPABILITIES WITH RTOS



For simpler controllers such as Arduino, it is apt to use Superloop, while as we go towards more powerful controllers such as STM32 and ESP32, the RTOS approach, though it may be more difficult, it becomes a better option, with the amount of Flash we have to store schedulers and other RTOS elements. RTOS helps us utilize the processing power of the controller to the fullest, for example, a controller as powerful as the STM32 or ESP32 is capable of handling so many tasks with minimal or no delays in any of the tasks.

FEATURES PROVIDED BY THE RTOS ARCHITECTURE

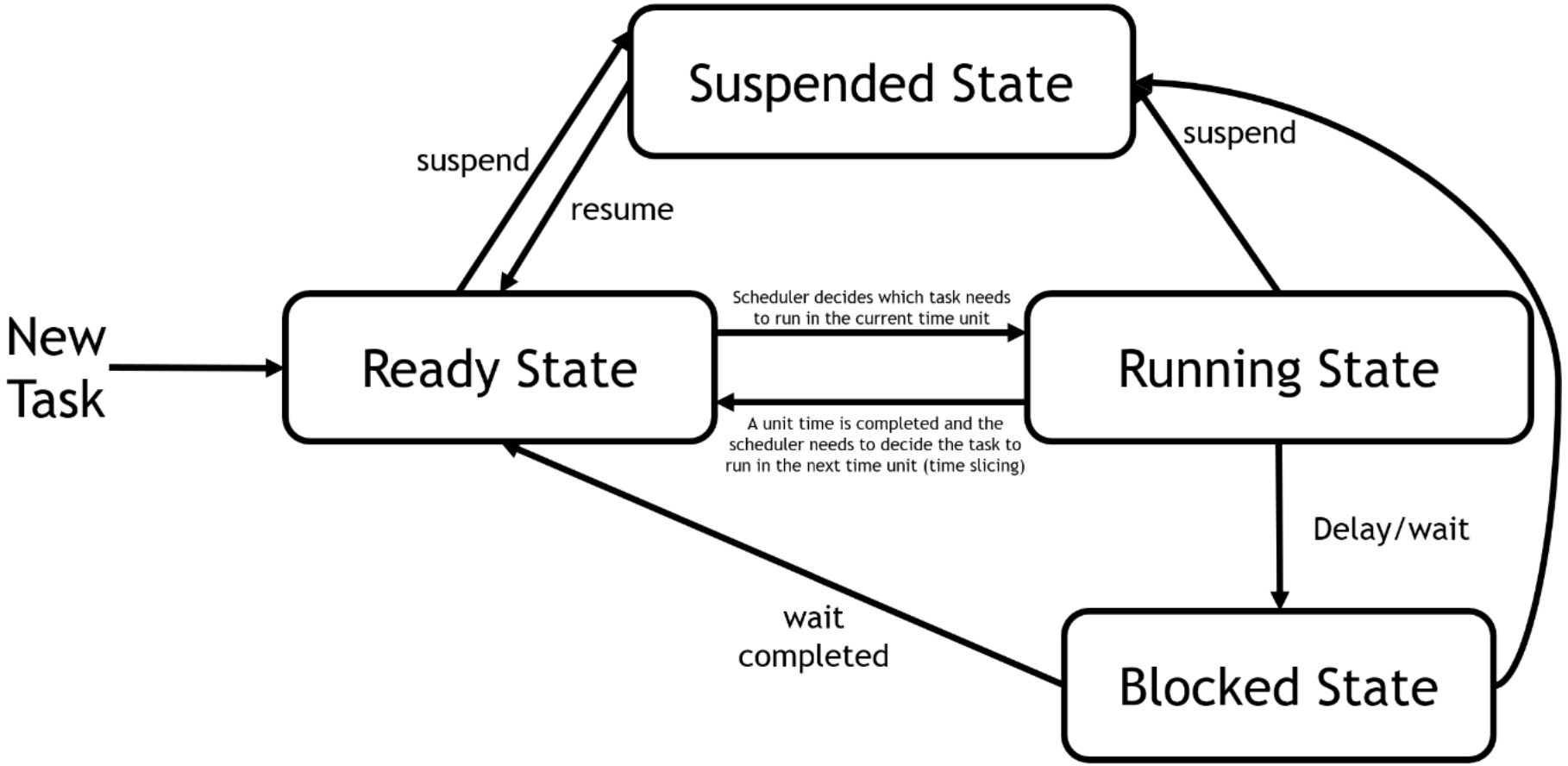
- Scheduler (Task Management, Multi-tasking, Meeting Timing Deadlines)
- Memory Management (Static and dynamic memory, Global variables, and Shared resources)
- Task Management, including timers, interrupts & ISRs.

FEW TERMS TO REMEMBER BEFORE WE PROCEED

- Task: It is a certain set of instructions to serve a certain set of applications. For example, blinking an LED, monitoring a sensor, etc.
- Thread: It is a sequence of instructions managed independently by the CPU. It has its own program counter and stack.
- Tick: It is the smallest unit of CPU's hardware timer which continuously resets the CPU. It is like a clock cycle for the Operating System. Usually, 1 Tick is 1ms. With every tick, the CPU decides which task needs to run, based on task priority and to meet any timing deadline.
- Task overhead stack: 768 bytes reserved for a Task control block.
- Non-blocking function: A task function that does not block the CPU from executing other tasks of the system.

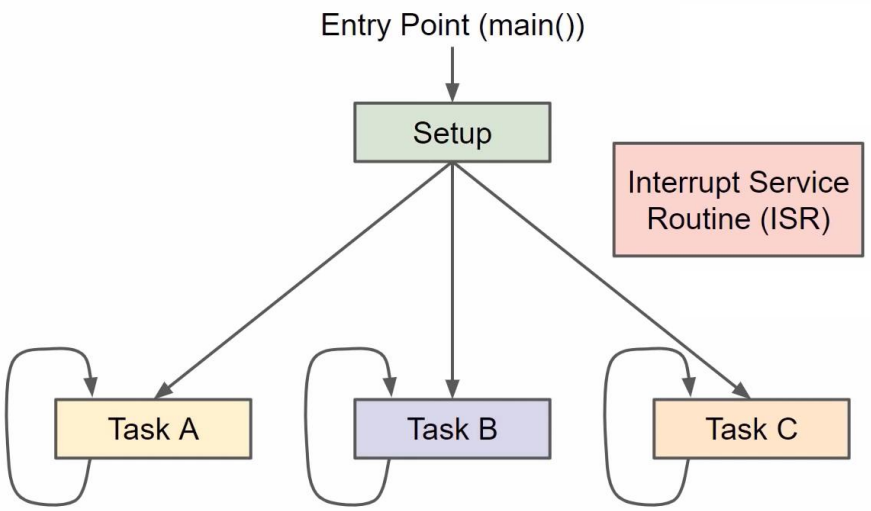
We will explore more about the above as we proceed further.

TASK STATES


Suspended State
suspend
resume
suspend
New
Task
Ready State
Scheduler decides which task needs
to run in the current time unit
Running State
A unit time is completed and the
scheduler needs to decide the task to
run in the next time unit (time slicing)
Delay/wait
wait
completed
Blocked State

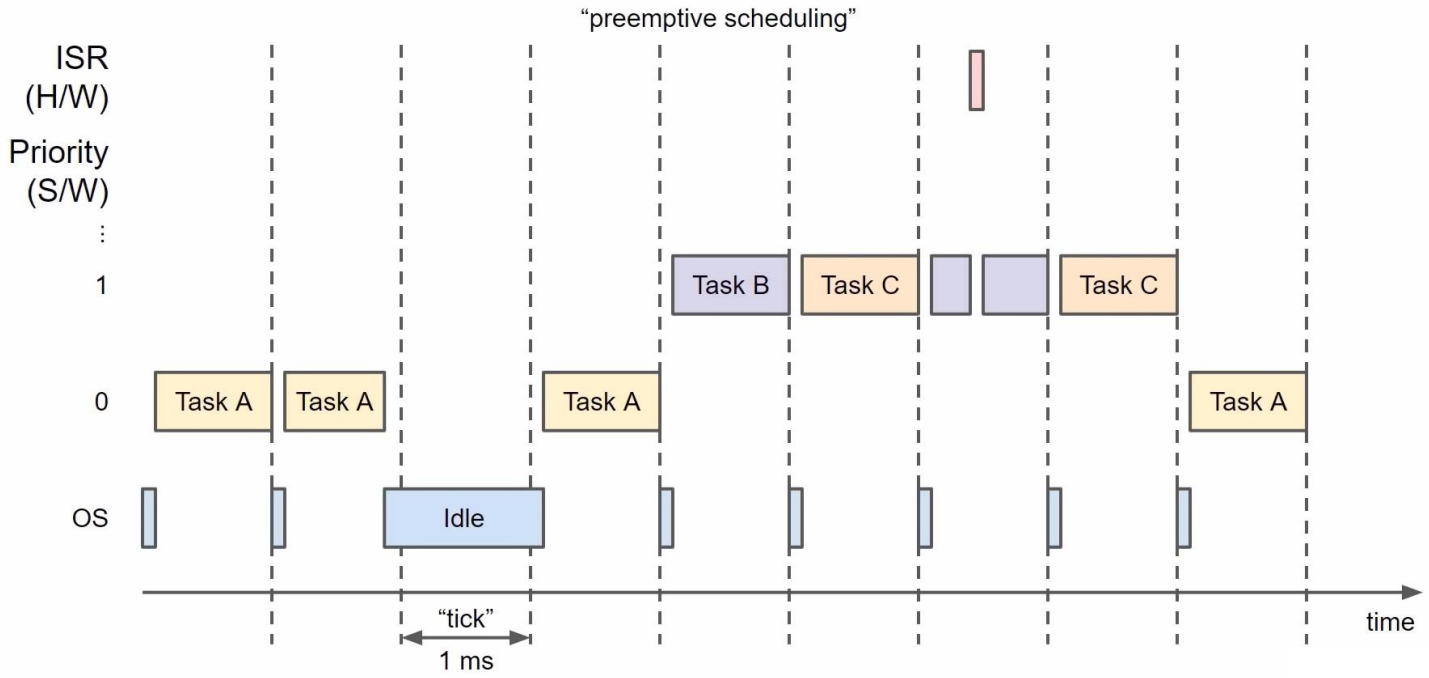
TASK SCHEDULING

What our code looks like



What actually happens*

*assuming single-core processor



TASK PRIORITY CONCEPT

PRIORITY ORDER (HIGH TO LOW)

- H/W Interrupt
- Timer Interrupt
- Numbered S/W Task priorities
- Operating System

DOWNLOAD FREERTOS

- Go to freeRTOS.org
- Download the first available freeRTOS package
- Include the libraries as required
- *Some basic functions may not require freeRTOS libraries at all*

1. LED BLINK

- Single task
- DIY Blink patterns using multiple tasks (Change priority levels, delay durations, etc. to see how the patterns change)

2. SERIAL READ/WRITE

- A simple task to read and write data to the Serial Monitor
- Multiple tasks printing different strings/characters (different priorities and delays) to the Serial monitor

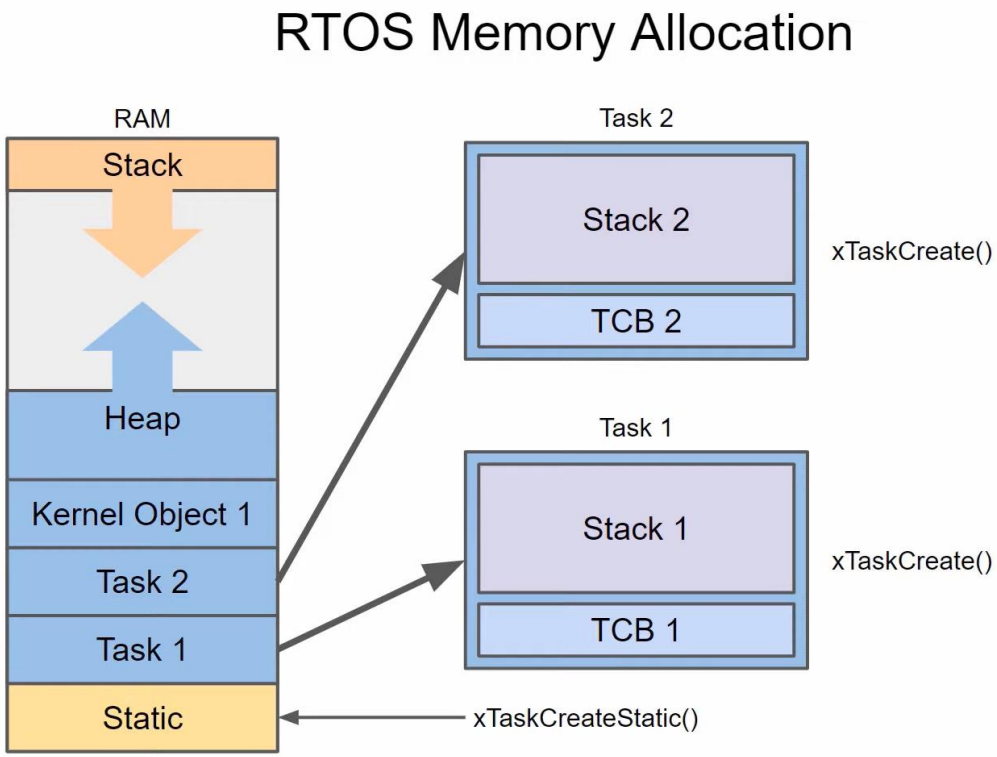
3. DIY COMMAND PROMPT

- One task reads the serial monitor and writes back the text, except if the console input is “delay x” in which case, the delay is updated to x milliseconds.
- The other task blinks the LED at interval of x milliseconds.

4. DHT SENSOR + WI-FI

- One task created to read the DHT temperature and humidity
- Another task is created to update the value to the web server created with ESP32 as an access point

MEMORY ALLOCATION



PROBLEMS RELATED TO MEMORY ALLOCATION

Memory Overflow – When the stack or heap flow into each other, it is called a memory overflow. Mostly occurs when memory that is no longer required is not freed regularly that triggers an exception, handled by resetting the processor.

Task Stack Overflow – Happens when the task stack exceeds beyond the allocated size. Usually, the last few bytes of the Stack are affixed, when they are overwritten, means the task is about to run out of stack, which also triggers an exception that is handled by resetting the processor.

Memory Fragmentation – Due to its purely dynamic nature, the Heap Memory can be fragmented in between, causing the heap to fill faster. The contiguous memory locations are separated by empty spaces, and to avoid this, there are some options provided by the Operating Systems configuration to manage the heap.

INTERMEDATE - INTRODUCTION TO RTOS KERNEL OBJECTS

KERNEL OBJECTS

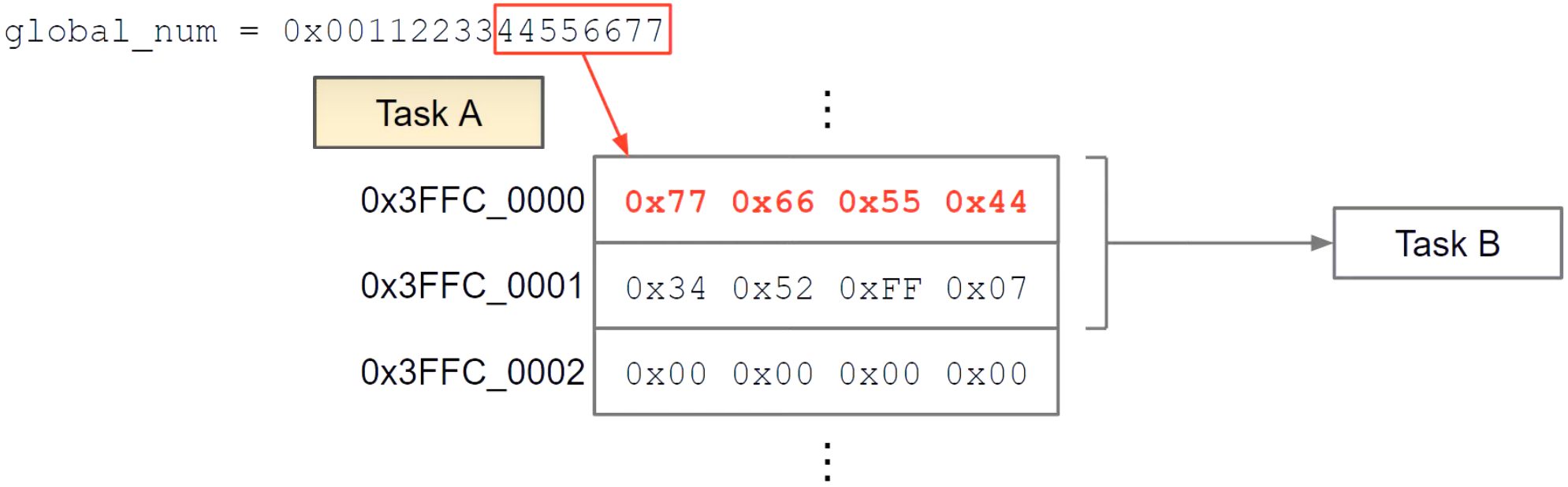
Kernel objects are blocks of memory created by the kernel, accessible to all the processes. These are usually **global** in scope, and their primary objective is to manage resources between the tasks, and optimize inter-task communication. Most RTOS functions are **atomic** functions, which means, the execute in a single instruction cycle.

There are three main Kernel objects that we mostly use.

1. Queue
2. Mutex
3. Semaphore

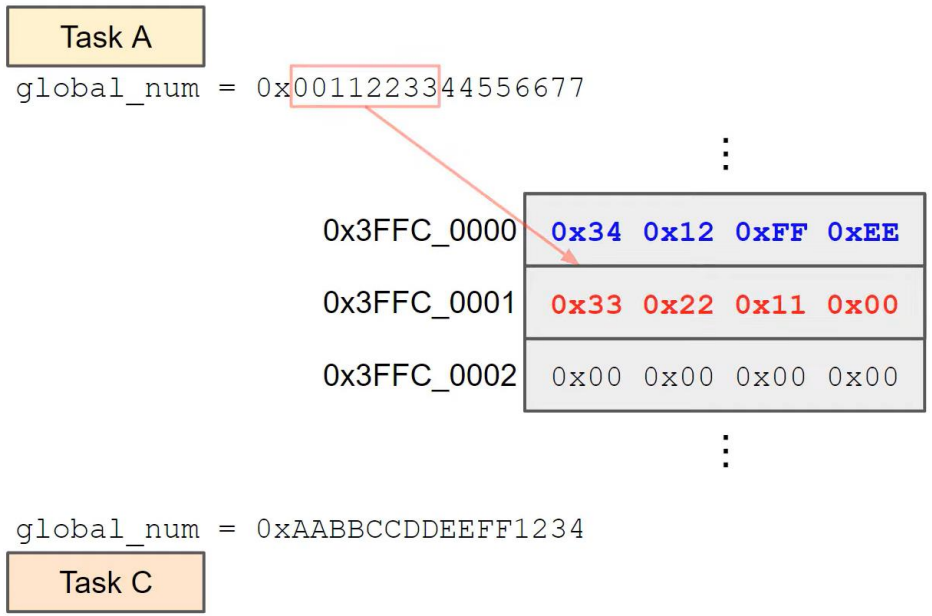
QUEUES

Assume we have multiple tasks updating the value of a global variable, multiple tasks reading the value same variable. Look at the below situation where Task A was writing a 64-bit value to a 32-bit register, and was interrupted by Task B which read the incorrect value, since the global variable is not thread safe.



QUEUES

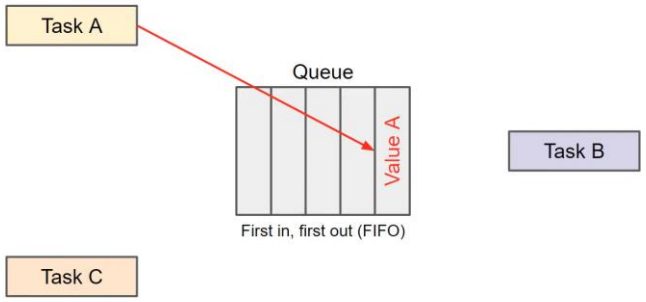
It can get even more challenging than that! Continuing from the previous example, suppose when Task A wrote the first half of the value, and then Task C preempted A and the whole number without interruption. Then, Task A picks up where it left off and writes the second half of the global variable. The value then read by Task B is also wrong.

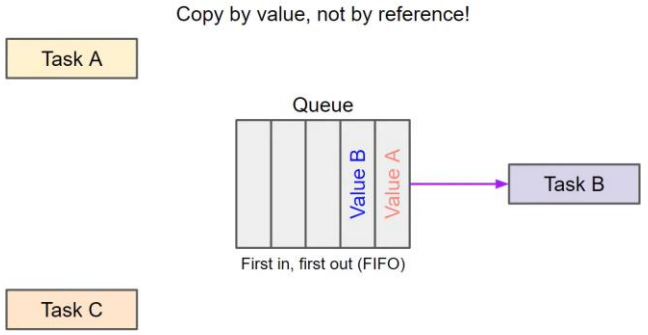


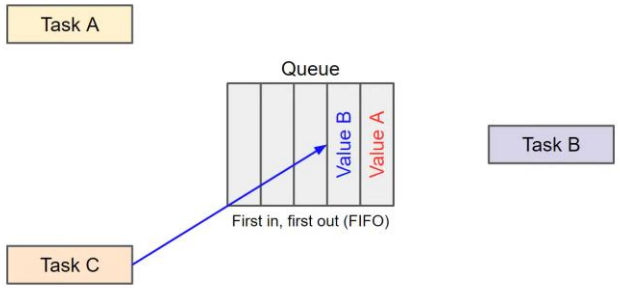
QUEUES

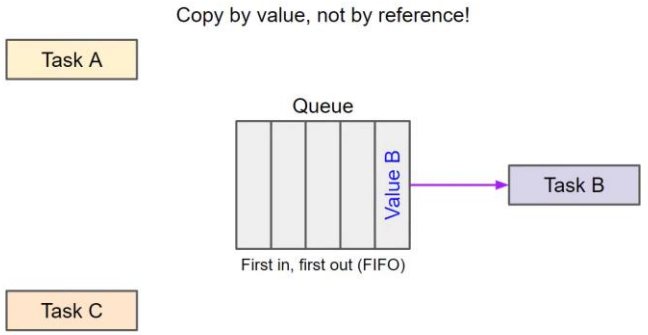
- We can avoid memory **overwrite** problems by a simple kernel object called a queue. As we know, queue is a FIFO data structure. The FreeRTOS Queue functions are **thread safe**, which means the task cannot be preempted while reading from or writing to a queue.
- Note that items are passed into the queue by **value** and not by **reference**.
- It is inconsiderate of datatype, and new data can stored in it until it is full.
- Let us see a demonstration of how global variables are in a queue.

QUEUES


Task A
Queue
Value A
Task B
First in, first out (FIFO)
Task C


Copy by value, not by reference!
Task A
Queue
Value B
Value A
Task B
First in, first out (FIFO)
Task C


Task A
Queue
Value B
Value A
Task B
First in, first out (FIFO)
Task C


Copy by value, not by reference!
Task A
Queue
Value B
Task B
First in, first out (FIFO)
Task C

QUEUES

Let's implement queues in this manner:

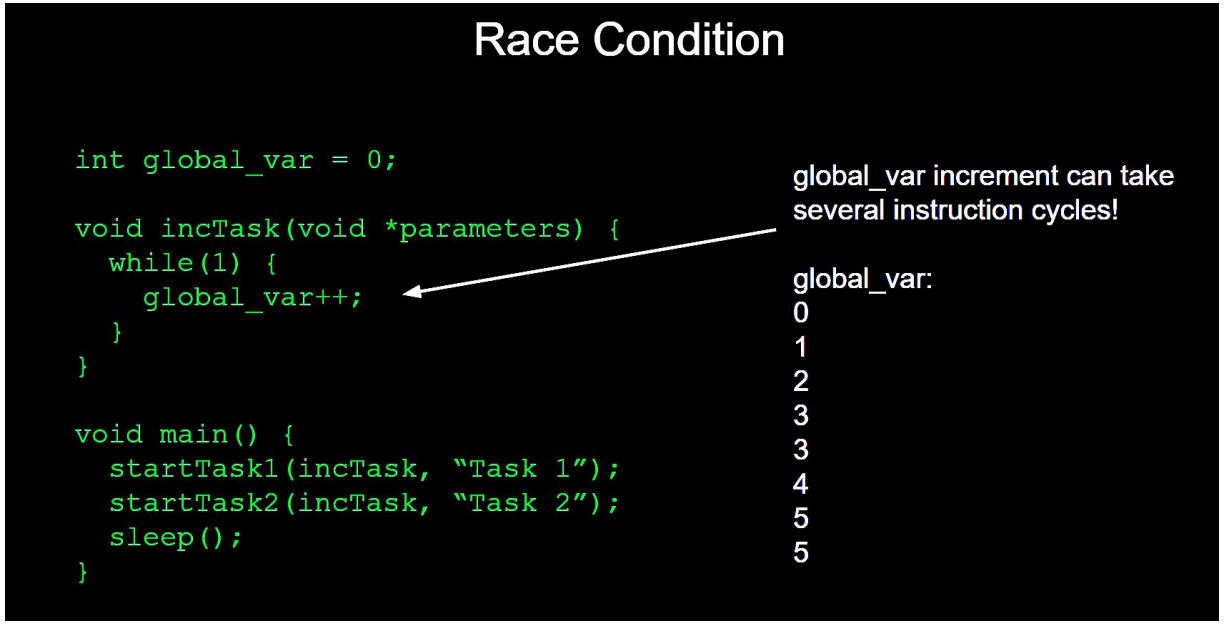
1. Task A sends a value to the Queue at an interval of 'X'
2. Task B reads the value from the Queue at an interval of 'Y' and prints it to the serial monitor.

Practice:

Remember the code where one task reads the value from serial monitor and other task modifies the LED blink based on the input, and also prints to the serial monitor? Let's implement the same program using Queue instead of a global variable to interact with the serial monitor.

MUTEX

- Before we dive in to mutex, let us understand a race condition. Suppose an operation on a shared resource takes more than one instruction cycle. We cannot determine when another task will preempt the current task. Let's see the pseudo code below as an example of how a race condition may occur.



MUTEX

A queue can help protect a shared resource, but what if the resource is a counter or a flag? Counters or Boolean flags are not advisable to be written in Queues as that would be a complex architecture as well as unnecessary utilization of excess memory.

We use a MUTEX (**Mut**ual **Ex**clusion) in this case. Let's understand them with a simple lock and key analogy.

Consider the restroom (**shared resource**) locked. There is only one key (**mutex**) that only one person (**task**) can take to use the restroom (**critical section**). The key is absent, which indicates that no other person can enter the restroom until the person holding the key has exited and returned the key to its location. This analogy is perfect to understand how a mutex works.

MUTEX

Let's implement the MUTEX to a counter:

- Task A increments the value at 'X' delay and prints the value
- Task B increments the value at 'Y' delay and prints the value
- A mutex is used to protect the critical section where the counter is updated

Note: Mutex is somewhat similar to a binary semaphore in architecture, and we use semaphore functions to create and use a mutex, although the functionality of mutex and binary semaphore may differ.

SEMAPHORE

By textbook definition, semaphore is a generalization of a mutex. To a great extent, this is true. A counting semaphore allows certain number of tasks to enter a critical section at once. If a counting semaphore has a maximum count of 3, which means that only three tasks can enter a critical section simultaneously.

In functionality, semaphores are not actually used to protect a critical section. Rather, they are used to signal other tasks that a shared resource is ready to use. Let's understand with an example.

SEMAPHORE

Let us take an analogy of a queue of hungry people (**tasks**) in a buffet waiting for spring roll (**shared resource**). Let's assume each person gets only one of it (when they need another second roll, they have to come again later at the back of the line). When the waiter places the spring rolls in the tray, it is an indication that a certain number (count of the **counting semaphore**) of spring rolls are ready and people can take spring rolls as long as there is any left in the tray. When the number becomes zero, the remaining people in the line have to wait for the spring rolls to become available again.

This may not be an accurate demonstration, but it will help you understand what a semaphore actually does.

SEMAPHORE

Let's implement a binary semaphore in this manner:

- Task A reads an integer from the serial monitor and gives a semaphore.
- Task B checks for the semaphore. If it sees that the semaphore is available, it reads the value and prints it back to the serial monitor.

Practice counting semaphores:

Let's create a Task A to write a counter to a queue every three seconds. Use a counting semaphore to signal the number of integers available in the queue to tasks B and C so they can print the values to the serial monitor at every 5 and 7 second intervals. Remember to print the task name as well so you know which task is printing which integer.

ADVANCED - TIMERS AND INTERRUPTS WITH ESP32 USING RTOS

TIMERS

- Delays can usually be blocking and can block the task from the CPU for a certain duration. What if you want to toggle an LED at one second intervals while also executing other instructions in the task? Timers are the perfect non-blocking approach.
- FreeRTOS provides atomic functions to use software timers efficiently. How timers work is simple: The default hardware timer runs at a certain speed of 160 or 240 MHz. The software timer can use a user-defined pre-scaler to run at a certain speed which is a divisor of the hardware clock speed. We can divide the default clock speed by 160k to make it a 1kHz clock, which essentially means, each tick of the clock will be of 1 millisecond. Note that FreeRTOS uses the same pre-scaler to create TICK_PERIOD of 1ms.
- When the timer count reaches a certain defined number, it will be reset, and start over again. Note that this is how a scheduler resets after every tick and chooses the next task to run on the CPU.

TIMERS

Let us implement a task which has a timer to blink an LED at intervals of 1 second, while also printing a character '*' on the serial monitor indefinitely.

Practice:

Create a task to print the Alphabet on the serial monitor indefinitely. Keep an extremely low Serial baud rate so you can see what is going on. Create a timer that prints an '*' at 100ms intervals. Note that you do not need to create separate tasks for this exercise.

INTERRUPTS

RTOS makes it possible to allow all tasks to **almost** meet timing deadlines, but what if you have certain tasks that cannot miss the timing deadline by even 1 microsecond? For example, the emergency system of a car cannot wait for the previous task to complete its CPU cycle before it can deploy airbags. In this case, we use interrupts to allow the emergency system to take over in the CPU and deploy the airbags.

In this case, the crash triggered an interrupt from the emergency system. The CPU will leave the execution of the current task and start executing the **ISR** or the interrupt service routine, which in this case could be to deploy the airbags.

Due to the nature of these events, it is recommended that ISR's should be very short and not consume too much of the CPU time since they are the highest priority tasks.

INTERRUPTS

For example, it is not advised to perform instructions in the ISR like getting the GPS location and updating it to the medical emergency services, as things like that are time consuming.

Instead, you can update another task using a semaphore, which has a lower priority than the ISR, and that task can extract the GPS location and update the paramedics, without blocking the other important tasks of the system. This is also an excellent use case for semaphores.

FreeRTOS has separate functions for each aspect which can be used in ISRs. For example, xQueueSend() can be used in a task, but in an ISR, we need to use xQueueSendFromISR() as the function is optimized for running in ISRs.

Let's implement a simple push button interrupt with ESP32 using the freeRTOS functions in the ISR.

ADVANCED - INFINITE DELAY, DEADLOCK AND STARVATION, PRIORITY INVERSION

DISCLAIMER

The following part of this workshop is more theoretical. While we can use the kernel objects to protect shared resources, there are certain problems that can occur, which are usually more difficult to diagnose and resolve.

When we have multiple tasks at different priority levels, require multiple shared resources to complete a certain process, some problems that can occur are as follows:

1. Infinite delay
2. Deadlock
3. Starvation
4. Priority Inversion

While there are no specific solutions to these problems since they are corner cases, their occurrence is very much possible when you have complex embedded systems, and you have to debug them manually.

HOW TO DIAGNOSE UNEXPECTED ERRORS IN A MICROCONTROLLER?

The microcontroller is equipped with a software timer called **Watchdog** timer. When a task functions properly, the TCB is designed to reset the watchdog timer, usually before it reaches the timeout. However, when the task crashes or becomes unresponsive due to an unexpected error, it does not reset the watchdog timer before timeout, at which point, the timer triggers an interrupt in the CPU.

Different microcontroller drivers have different ISRs when a watchdog timeout is triggered. Some reset the controller while, others may just print out a warning over the UART (Serial Monitor). The watchdog timer can help diagnose errors, but it **cannot** tell which kind of error has occurred. These usually need to be diagnosed by dry runs, and they can only be resolved by manual debugging.

INFINITE DELAY

Sometimes, we create tasks that wait for responses from other devices, such as Serial monitor inputs, response from I2C devices, or SPI devices, etc., and there is no possible way to tell how long it has to wait in the running state till it gets a response from the target device. This situation is called an infinite delay.

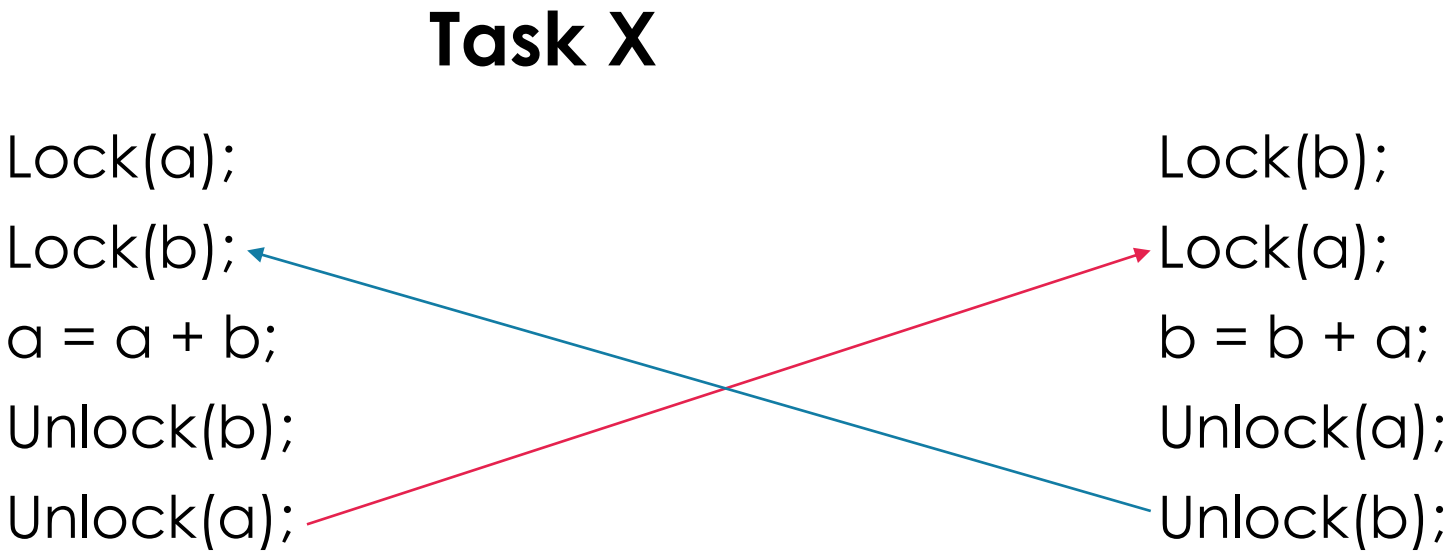
Diagnosis is quite easy, since the infinite delay usually occurs in the parts of the program where communication with an external device is involved.

How microcontroller drivers such as ESP-IDF and STM-HAL tackle these issues is simple. They use a timeout, where they usually wait for the input for a certain timeout before they exit the function.

Similar solutions can be implemented for preventing other exceptions as well.

DEADLOCK

Let's visualize how a deadlock may occur using an example. Let **a** and **b** be global variables. Tasks X and Y require **a** and **b** to perform their sum and update them.



Task X locks variable a. Just as it is about to lock b as well, Task Y pre-empts it and locks b. Next, when task X wants to lock b, it cannot, and task Y cannot lock a. Neither of the tasks will unlock until they have both mutexes and have performed the sum. In this situation, both tasks will not be able to complete resulting in infinite delay. This is called a deadlock.

STARVATION

Imagine a hospital emergency room. There are patients (**tasks**) with a higher priority due to more injury and other patients with a low priority due to less injury. Who will the doctors (**CPU scheduler**) attend first? It goes without saying that they will attend the more emergent cases. It is possible that some lower priority patients may not even be visited the doctor till the end of the day. This means that the lower priority patients undergo starvation.

This may not be the best analogy, but it does help us understand the concept of starvation.

When a task is not able to gain access to the CPU for a long duration of time, the task is known to be in **starvation**. Starvation is also difficult to diagnose, but again, it can only be resolved by manual debugging.

PRIORITY INVERSION

Let's consider the scenario which we used to understand mutexes. The restroom example. A person who has kidney issues (**task**) has a *higher priority* to use the restroom (**shared resource**). Another person has already occupied the restroom and is taking a long time, hence making the person with higher priority to wait. In this case, due to the mutex, a lower priority task has pre-empted the higher priority task, this situation is called priority inversion.

You can solve this problem by creating a simple timeout. You can use a software timer to do the same. If a particular task holds a mutex for over a certain time, you can use a software timer to create an ISR, so the timer will definitely interrupt the processor, and block the lower priority task, so the higher priority task can take place.

THANK YOU