CodeGuide

QUICK NOTES
FOR SMART LEARNING

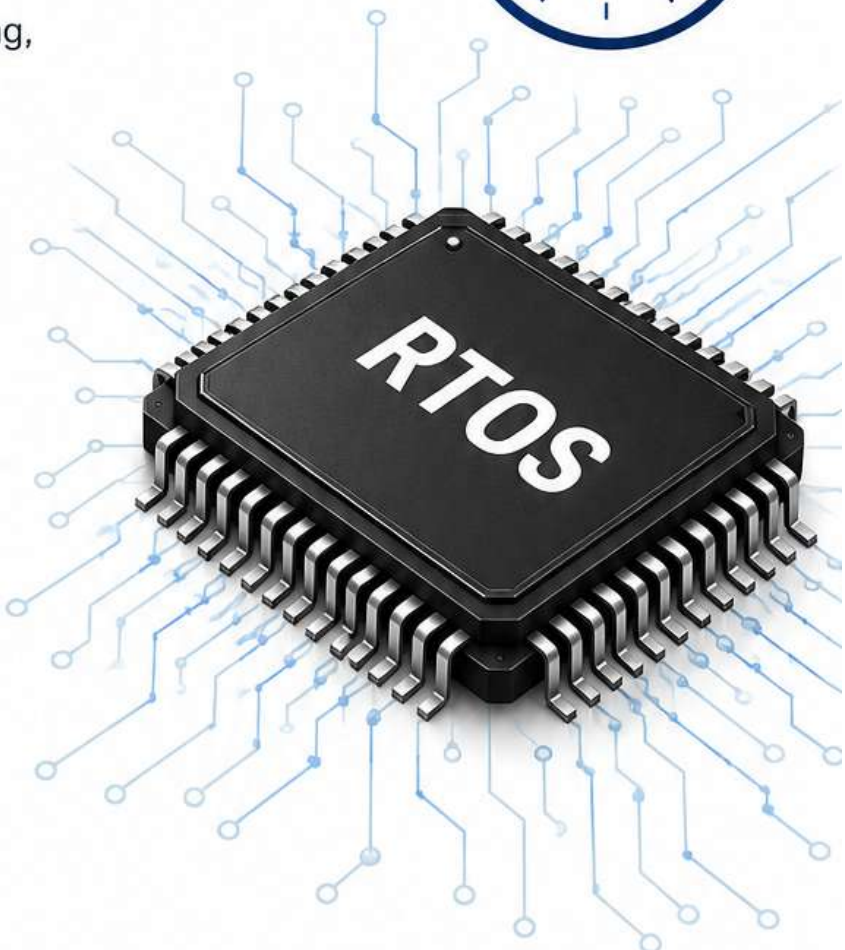
RTOS

REAL-TIME OPERATING SYSTEMS



A COMPLETE GUIDE

Concepts, Architecture, Scheduling, Synchronization, and Practical Applications



CORE CONCEPTS
Understand the fundamentals of RTOS with clarity

KEY TOPICS
Tasks, scheduling, synchronization, IPC, timers, and more

ARCHITECTURE
Explore RTOS architecture and its components

PRACTICAL APPROACH
Real-world examples and implementation insights

EXAM READY
Perfect for interviews, exams, and quick revision

STUDENTS
Build Strong Fundamentals

DEVELOPERS
Enhance Skills & Productivity

EMBEDDED SYSTEMS
Designed for Real-Time Applications

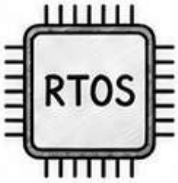
LEARN. CODE. BUILD. INNOVATE.

REAL TIME. REAL PERFORMANCE. REAL POSSIBILITIES.

RTOS BASICS

1. WHAT IS RTOS ?

- Real-Time Operating System
- Manages tasks and resources with deterministic timing
- Designed for embedded systems

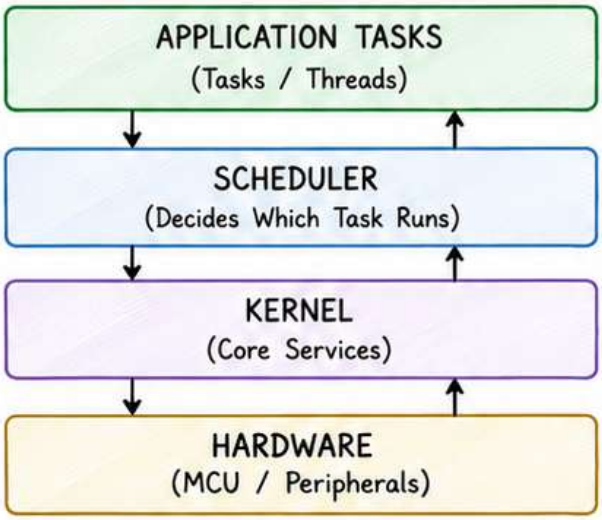


2. WHY RTOS ?

- ✓ Multitasking
- ✓ Fast and predictable response
- ✓ Better resource management
- ✓ Reliable and stable
- ✓ Scalable for complex systems



3. RTOS ARCHITECTURE



4. TYPES OF RTOS

Type	Description
Hard RTOS	Missing deadline is unacceptable
Soft RTOS	Missing deadline is tolerable
Firm RTOS	Late result is useless but system continues

5. KEY COMPONENTS

- Tasks
- Scheduler
- Queue
- Semaphore
- Ciutex
- Mutex
- Event Group
- Timers
- Memory Management

6. TASK STATES



7. POPULAR RTOS EXAMPLES

- FreeRTOS
- Zephyr
- RTX (Arm)



- ThreadX
- VxWorks
- µC/OS-II / III

8. APPLICATIONS

- Automotive Systems
- Medical Devices
- Industrial Automation
- IoT Devices
- Robotics
- Aerospace Systems

9. ADVANTAGES

- ★ Predictable timing
- ★ Efficient CPU utilization
- ★ Better resource sharing
- ★ Modular and scalable



10. CONCLUSION



RTOS helps embedded systems run multiple tasks efficiently with real-time guarantees, making systems reliable, scalable, and easier to manage.



TASKS (THREADS) IN RTOS

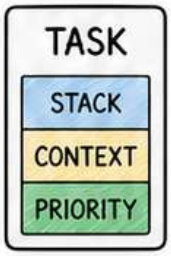
1. WHAT IS A TASK ?

- A task (or thread) is the smallest unit of execution in an RTOS.
- Each task runs independently with its own stack and context.
- Managed and scheduled by the RTOS kernel.

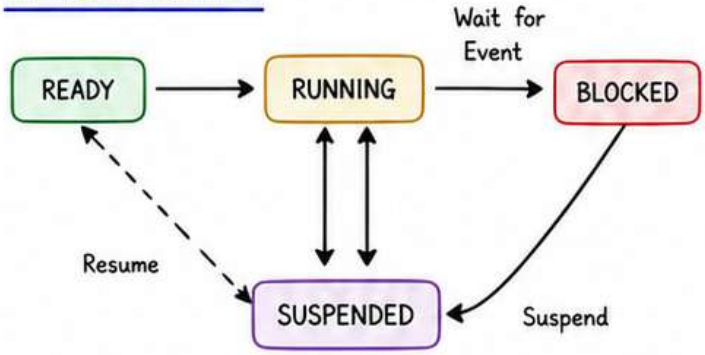


2. TASK CHARACTERISTICS

- ✓ Independent execution
- ✓ Own stack and context
- ✓ Has priority
- ✓ Can be blocked, delayed or suspended
- ✓ Communicates with other tasks



3. TASK STATES



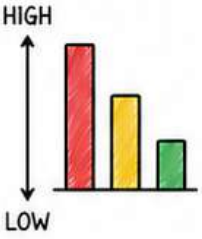
4. TASK CREATION (BASIC)

```
TaskHandle_t task;
xTaskCreate(
   taskFunction,      // Function
   "Task1",           // Name
   stackSize,         // Stack size
   parameters,        // Parameters
   priority,          // Priority
   &task              // Task handle
);
```



5. TASK PRIORITY

- Each task has a priority.
- Higher priority tasks run before lower priority tasks.
- If priorities are same, scheduler decides.
- Priority range depends on RTOS.



6. TASK DELAYS

- Task can wait (delay) for a specific time.
- Allows other tasks to run.
- Example:

```
vTaskDelay(1000);   // Delay for 1000 ms
```



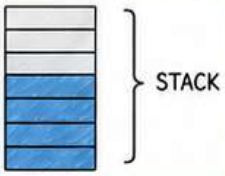
7. TASK YIELD

- A running task can give up CPU voluntarily.
- Allows other ready tasks to run.
- Example: `taskYIELD();`

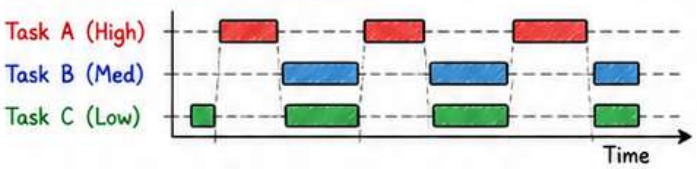


8. TASK STACK

- Each task needs its own stack.
- Stack stores local variables, function calls, and context.
- Stack size must be set carefully.



9. MULTITASKING EXAMPLE



Higher priority tasks run first.

10. KEY POINTS

- ⭐ Task is the basic unit of execution in RTOS.
- ⭐ Each task has its own stack, priority and context.
- ⭐ Scheduler decides which task runs.
- ⭐ Proper use of tasks enables efficient multitasking.



CONCLUSION

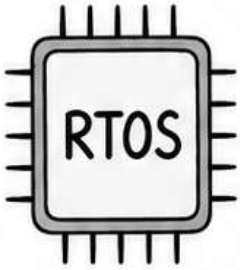


Tasks are the heart of any RTOS. Understanding task states, priority, and behavior is essential for building efficient and reliable embedded applications.

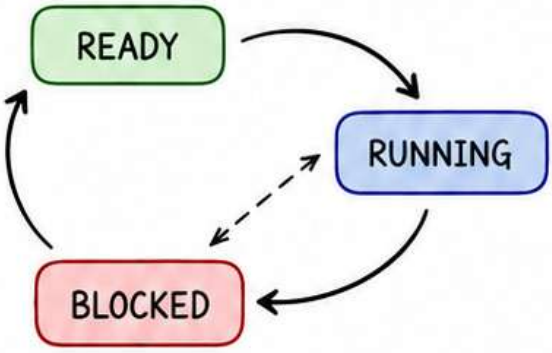
PROCESS IN RTOS

1 WHAT IS PROCESS?

- A process (or task) is the basic unit of execution in an RTOS.
- It has its own stack, context and state.
- Managed by the scheduler.



2 PROCESS STATES



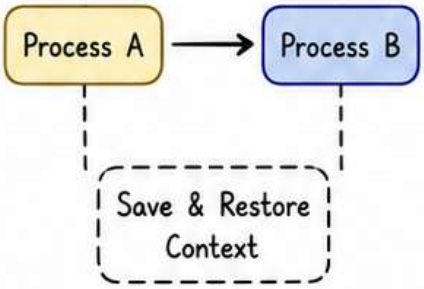
- READY : Waiting for CPU
- RUNNING : Executing
- BLOCKED : Waiting for Event/I/O

3 PCB (PROCESS CONTROL BLOCK)

Process ID
Priority
State
Stack Pointer
Program Counter

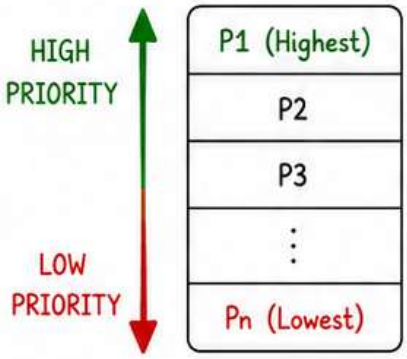
PCB stores all information required to manage and resume a process.

4 CONTEXT SWITCH



- CPU switches from one process to another.
- Saves current context.
- Restores next process context.

5 PRIORITY



- Each process has a priority.
- Higher priority gets CPU first.
- Low priority waits for CPU.

6 IMPORTANCE



Real-Time Response
Ensures tasks complete on time.



Multitasking
Multiple tasks run seemingly at the same time.



Better CPU Utilization
CPU is used efficiently.

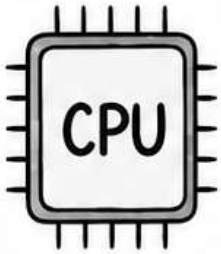


Efficient Resource Management
Resources are used and shared in a controlled way.

CONTEXT SWITCHING

1 WHAT IS CONTEXT SWITCHING?

- Context switching is the process of saving the state of the current process and loading the state of another process.
- It allows the CPU to switch from one process to another.

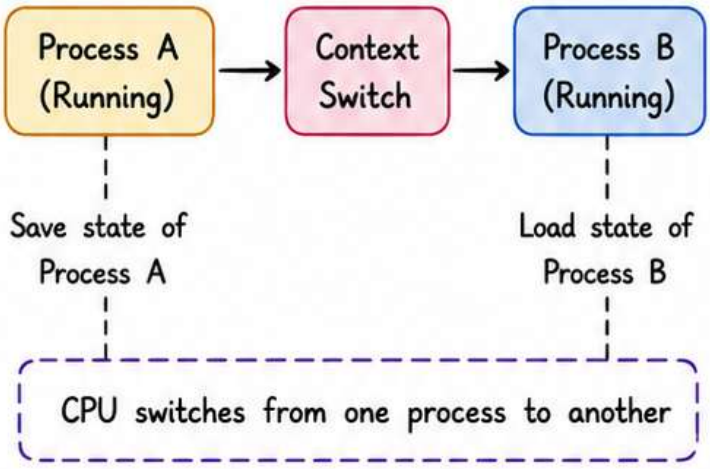


2 WHY IS IT NEEDED?

- Enables multitasking.
- Allows fair CPU usage among processes.
- Ensures responsiveness in real-time systems.



3 PROCESS FLOW (A → B)



4 STEPS OF CONTEXT SWITCHING

1. Save the context of the current process (PC, Registers, Stack Pointer, etc.).
2. Store it in the PCB of that process.
3. Select the next process by the scheduler.
4. Load the context of the next process from its PCB.
5. Resume execution of the next process.



5 WHAT IS SAVED?

The following information (context) is saved in PCB during a context switch:

PC	Program Counter
R	CPU Registers
SP	Stack Pointer
	Other CPU status (Flags, PSW, etc.)

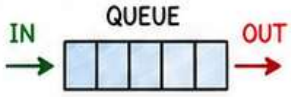
6 BENEFITS

- Enables Multitasking
- Better CPU Utilization
- Real-Time Responsiveness
- Efficient Task Management

QUEUES IN RTOS

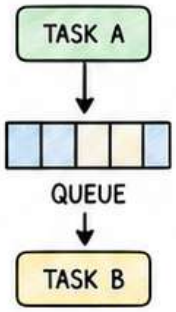
1. WHAT IS A QUEUE ?

- A queue is a data structure that stores items in FIFO (First In, First Out) order.
- Used for data communication between tasks.
- Provided and managed by the RTOS kernel.



2. WHY USE QUEUES ?

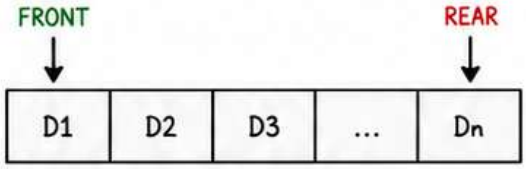
- ✓ Safe data exchange between tasks
- ✓ Works across different priorities
- ✓ Tasks don't need to run at the same time
- ✓ Avoids data corruption
- ✓ Supports task synchronization



3. QUEUE OPERATIONS

Operation	Description
xQueueCreate()	Create a queue
xQueueSend()	Send data to queue
xQueueReceive()	Receive data from queue
uxQueueMessagesWaiting()	Check number of messages in queue
vQueueDelete()	Delete a queue

4. QUEUE STRUCTURE

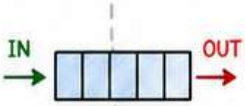


- Front : Points to the first data
- Rear : Points to the next empty position
- Size : Maximum number of items queue can hold

5. SEND AND RECEIVE

xQueueSend()

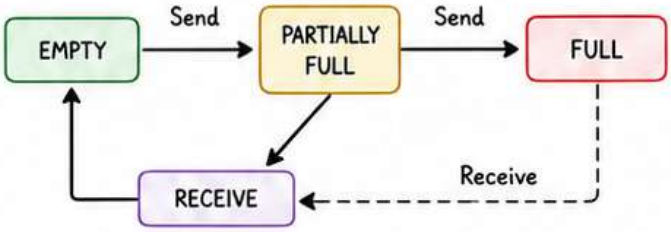
- Sends data to queue
- If queue is full, the task may block (if block time > 0)
- Returns pdPASS on success



xQueueReceive()

- Receives data from queue
- If queue is empty, the task may block (if block time > 0)
- Returns pdPASS on success

6. QUEUE STATES

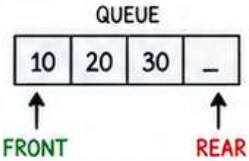


7. QUEUE EXAMPLE

Queue size = 4

xQueueSend()

Send: 10, 20, 30

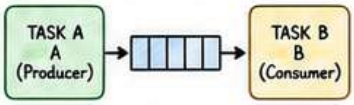


xQueueReceive()

Receive: 10

8. TYPICAL USES

- Communication between tasks
- ISR to task communication
- Event/message passing
- Buffering data
- Producer - Consumer problem



9. IMPORTANT POINTS

- ⭐ Queue follows FIFO principle.
- ⭐ Tasks can block if queue is full (on send) or empty (on receive).
- ⭐ RTOS handles queue management internally.
- ⭐ Use queues for reliable and safe data transfer.



10. CONCLUSION



Queues are powerful tools in RTOS for safe and efficient communication between tasks. They help in synchronization, data transfer and building robust real-time applications.

MUTEX

1. WHAT IS A MUTEX?

- Mutex (Mutual Exclusion) is a synchronization primitive used to protect shared resources from concurrent access.
- It ensures that only one task/thread can access a critical section at a time.

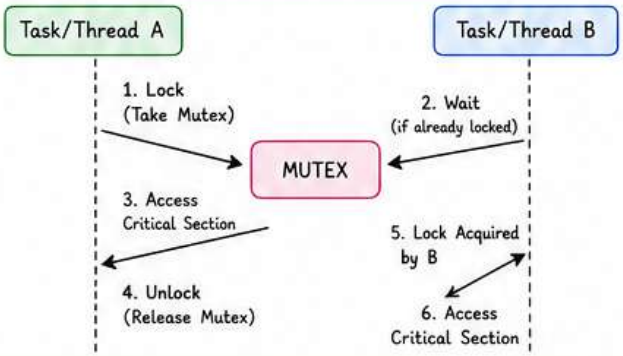


2. WHY USE A MUTEX?

- ✓ Prevents race conditions
- ✓ Ensures data integrity
- ✓ Protects shared resources
- ✓ Avoids unpredictable behavior
- ✓ Improves system reliability



3. HOW MUTEX WORKS?



4. MUTEX STATES

State	Description
Unlocked (Available)	Mutex is free. Any task/thread can lock and use the resource.
Locked (Busy)	Mutex is taken by a task/thread. Others must wait.
Blocked (Waiting)	Tasks/threads waiting for the mutex to become available.

5. MUTEX API (EXAMPLE - FREERTOS)

- `xSemaphoreHandle xSemaphoreCreateMutex(void);`
  `// Create a mutex`
- `xSemaphoreTake(mutex, timeout);`
  `// Take (lock) the mutex`
- `xSemaphoreGive(mutex);`
  `// Give (unlock) the mutex`
- `xSemaphoreDelete(mutex);`
  `// Delete the mutex`

6. MUTEX USAGE EXAMPLE

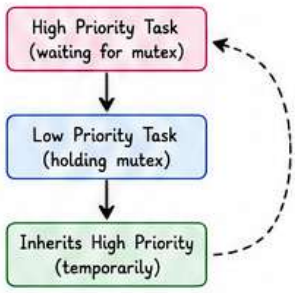
```
SemaphoreHandle_t mutex;
void task_function(void *arg)
{
    while (1)
    {
        xSemaphoreTake(mutex, portMAX_DELAY);   // Lock
        // ---- Critical Section ----
        // Access shared resource
        // -------------------------
        xSemaphoreGive(mutex);                  // Unlock
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

7. MUTEX vs BINARY SEMAPHORE

Feature	Mutex	Binary Semaphore
Ownership	Has owner (task/thread)	No owner
Release	Can be released only by owner	Can be released by any task
Priority Inheritance	Yes (avoids priority inversion)	No
Use Case	Protect shared resources, critical sections	Event signaling, resource availability

8. PRIORITY INHERITANCE

- Priority Inversion: A low priority task holds the mutex needed by a high priority task.
- Priority Inheritance: The low priority task inherits the high priority until it releases the mutex.



9. BEST PRACTICES

- ✓ Keep critical sections short.
- ✓ Always lock before accessing shared resources.
- ✓ Always unlock after use (even on error).
- ✓ Avoid nested mutex locks.
- ✓ Use priority inheritance mutexes (if available).
- ✓ Do not use mutex in ISRs.

10. COMMON ISSUES

- X Deadlock – Two or more tasks waiting for each other.
- X Priority Inversion – Low priority task blocks high priority task.
- X Forgot to Unlock – Leads to permanent blocking.
- X Long Critical Section – Increases blocking time.



CONCLUSION

Today we learned :

- What is a Mutex
- How Mutex Works
- Mutex States
- Mutex APIs
- Mutex vs Binary Semaphore
- Priority Inheritance
- Best Practices
- Common Issues