GOVERNMENT COLLEGE OF ENGINEERING AND RESEARCH, AWASARI, PUNE

DEPARTMENT OF ELECTRONICS AND TELECOM. ENGINEERING

Embedded System and RTOS (Elective - III)

NAME :

_____

SUBJECT: Embedded System and RTOS (Elective - III)

_____

CLASS : BE (E&TC) **ROLL NO. :**

YEAR : _____**EXAM SEAT NO. :** _____

Awasari Khurd, Tal – Ambegaon,
Dist – Pune. Pin – 412405

INDEX

Sr. No.	Title	Date	Sign
1.	Multitasking in µCOS II RTOS using minimum 3 tasks on ARM7/ ARM Cortex- M.		
2.	Semaphore as signaling & Synchronizing on ARM7/ ARM Cortex- M.		
3.	Mailbox implementation for message passing on ARM7/ ARM Cortex- M.		
4.	Implementation of MUTEX using minimum 3 tasks on ARM7/ ARM Cortex- M.		
5.	Interfacing sensors and actuators with Arduino Uno- Door opener using Ultrasonic sensor and servo motor		
6.	Weather Station- Build a cloud-ready temperature and Humidity sensor (DHT-11/22) with the Node MCU and the any IoT Platform.		
7.	IoT based Wireless Controlled Home Automation using ESP8266.		
8.	Interfacing of 4 LED bank with Raspberry Pi to blink.		

CERTIFICATE

This is to Certified that Mr / Ms. _____Roll no._____ of B.E. _____ has satisfactorily completed the work of the subject Embedded System and RTOS (Elective - III).

Subject Teacher ***Head of Department***

Experiment No : 1

Title

Multitasking in µC/OS-II using minimum 3 tasks on ARM7 / ARM Cortex-M

Objective

- Learn to create, schedule and manage multiple tasks in µC/OS-II.
- Implement at least three concurrent tasks demonstrating task priorities, inter-task communication (semaphores/queues), and synchronization.
- Build and run a simple µC/OS-II project on ARM7 (e.g., LPC2148) or ARM Cortex-M (e.g., STM32F1/F4) using Keil uVision or an equivalent toolchain.

Learning Outcomes

By the end of this lab, students will be able to: - Initialize µC/OS-II and create tasks. - Assign priorities, stacks and manage task context switching. - Use semaphores and message queues for synchronization and data passing. - Debug basic RTOS issues (stack overflow, priority inversion, deadlocks).

Prerequisites

- C programming (embedded C)
- Basic understanding of ARM7 / Cortex-M MCU architecture
- Familiarity with hardware peripherals (GPIO, UART, ADC)
- Basic knowledge of RTOS concepts (tasks, scheduler, priorities)

Hardware Required

- Development board: ARM7 (e.g., NXP LPC2148) or ARM Cortex-M (e.g., STM32F103/STM32F407)
- USB-to-serial or onboard debugger (JTAG/SWD)
- Breadboard and jumper wires
- LEDs (3) + resistors
- Push-button (optional)
- Sensor like LM35 (optional) or potentiometer for ADC reading

Software Required

- Keil uVision (or GCC ARM + Makefile)

- µC/OS-II source code (ported for your MCU) — ensure you have a licensed copy if required
- Board support package (startup code, linker script, peripheral drivers)
- Terminal emulator (Tera Term / Putty)

Overview of the Experiment

In this lab you will build a µC/OS-II application that runs three tasks concurrently:

Task A (LED Task) — Toggle LED1 every 200 ms (low priority)

Task B (UART Task) — Send a periodic status message over UART every 500 ms (medium priority)

Task C (Sensor/ADC Task) — Read ADC or simulated sensor value every 300 ms and post data to a queue (high priority). The UART Task will consume the queue and transmit the sensor value.

This arrangement demonstrates preemptive scheduling, priority assignment, inter-task communication (OSQ – message queue) and synchronization (OSSem).

Design & Architecture

- OS initialization (OSInit)
- Create tasks with appropriate priorities using OSTaskCreateExt/OSTaskCreate
- Create inter-task primitives: OSQCreate, OSSemCreate (if needed)
- Use OSTimeDly/OSTimeDlyHMSM for periodic behavior
- Start the scheduler (OSStart)

Task priorities (example)

- Task C (Sensor): Priority 3 (highest)
- Task B (UART): Priority 5
- Task A (LED): Priority 10 (lowest)

  In µC/OS-II, lower numeric value = higher priority (priority 0 is the highest). Choose priorities appropriately for your system.

Implementation Steps (Detailed)

1. Project setup

1. Create new µVision project for target board.

2. Add startup file, system clock configuration, linker script, and CMSIS/driver files.
3. Add µC/OS-II source files (os_cpu_a.asm/c, os_cfg.h, os_cfg.c, os_core files) to the project.
4. Configure OS_CFG.H:
    - Set OS_TASKS to at least 5 (three user tasks + system tasks)
    - Enable OS_MAX_EVENTS and message queues if using OSQ
    - Set OS_STK_GROWTH, OS_STK_SIZE defaults as per port
    - Configure OS_TICKS_PER_SEC (e.g., 1000 for 1ms tick)

2. System tick configuration

- Ensure SysTick (Cortex-M) or timer interrupt (ARM7) is configured to call OSTimeTick() from the OS tick ISR. This is essential for OSTimeDly and time slicing.

3. Create Tasks

- Define stacks and TCBs (if using OSTaskCreateExt)

Example task prototypes and stacks (C):

```
#include "includes.h" // standard µC/OS-II include header

#define TASK_STK_SIZE 256

OS_STK TaskLedStk[TASK_STK_SIZE];
OS_STK TaskUARTStk[TASK_STK_SIZE];
OS_STK TaskSensorStk[TASK_STK_SIZE];

void TaskLED(void *pdata);
void TaskUART(void *pdata);
void TaskSensor(void *pdata);
```

Task creation in main() or Startup Task:

```
int main(void) {
    OSInit();

    // Create synchronization primitives
    // OS_EVENT *sensorDataQ = OSQCreate(...);

    OSTaskCreate(TaskLED, (void*)0, &TaskLedStk[TASK_STK_SIZE-1], 10);
    OSTaskCreate(TaskUART, (void*)0, &TaskUARTStk[TASK_STK_SIZE-1], 5);
    OSTaskCreate(TaskSensor, (void*)0, &TaskSensorStk[TASK_STK_SIZE-1], 3);

    OSStart();
    return 0;
}
```

Note: If you use OSTaskCreateExt, allocate OS_TCB structures and pass stack pointers and names — this helps debug.

4. Task code examples

LED Task (low priority)

```
void TaskLED(void *pdata) {
    (void)pdata;
    for (;;) {
        LED_Toggle(LED1);
        OSTimeDlyHMSM(0, 0, 0, 200); // 200 ms
    }
}
```

Sensor Task (high priority)

```
void TaskSensor(void *pdata) {
    (void)pdata;
    INT16S sensor_val;
    for (;;) {
        // Read ADC or simulate
        sensor_val = ReadADC_Channel(0);

        // Send pointer/value to queue
        OSQPost(sensorDataQ, (void *)((INT32U)sensor_val));

        OSTimeDlyHMSM(0,0,0,300); // 300 ms
    }
}
```

UART Task (medium priority)

```
void TaskUART(void *pdata) {
    (void)pdata;
    void *msg;
    INT32U msg_val;

    for (;;) {
        // Wait for queue message (blocking)
        msg = OSQPend(sensorDataQ, 0, &err);
        if (msg != (void*)0) {
            msg_val = (INT32U)msg;
            printf("Sensor: %lu\r\n", msg_val);
        }
        OSTimeDlyHMSM(0,0,0,500); // periodic print (optional)
    }
}
```

Use proper type casting and dynamic memory caution. For robust designs, post pointers to shared buffers with proper protection.

5. Inter-task communication

- Create the queue before starting tasks:

```
OS_EVENT *sensorDataQ;
void create_sync_objects(void) {
    sensorDataQ = OSQCreate(&sensorQStorage[0], 10); // storage array must be
defined
}
```

- Alternatively, use OSMboxPost / OSMboxPend or semaphores depending on the use case.

6. Build & Flash

- Compile the project and fix linker/startup warnings.
- Flash to target board via debugger.
- Open serial terminal (115200, 8N1) to view UART output.

Testing Procedure

5. Power the board and connect the debugger.
6. Load the firmware and start execution.
7. Verify LED toggling rate (~200 ms).
8. Check UART terminal for periodic messages and sensor readings.
9. Vary sensor input (if using potentiometer) and observe UART updates.
10. Intentionally create delay in UART task to observe preemption by higher-priority Sensor task.

Expected Observations

- LED toggles independently and doesn't block other tasks.
- Sensor task (higher priority) will preempt lower-priority tasks if it becomes ready.
- UART receives sensor data via queue and transmits it.

Questions

1. Explain how µC/OS-II schedules tasks. What determines which task runs?
2. If Task A (low priority) never yields, how does µC/OS-II ensure higher-priority tasks run? (Hint: tick interrupt and preemption)
3. How many stacks are required and why?
4. Propose a solution to protect a shared global buffer between TaskSensor and TaskUART.
5. If you observe UART prints being delayed, which debugging steps would you follow?

Common Issues & Troubleshooting

- **No OS tick**: Ensure SysTick/TIMER ISR calls `OSTimeTick()` and tick interrupt priority is correct (Cortex-M NVIC priorities).
- **Stack overflow**: Increase task stack sizes or enable `OS_TASK_STAT_EN` and `OS_TASK_CHANGE_PRIO` for debugging. Use canary patterns to check stack usage.
- **Wrong priorities**: Remember µC/OS-II uses lower numbers = higher priority.
- **Queue post/pending errors**: Check that the queue storage array is large enough and created before use.
- **Printf not working**: Ensure UART driver is initialized before using printf; retarget `fputc` if necessary.

Appendix: Useful µC/OS-II API snippets

- `OSInit();` — initialize OS
- `OSStart();` — start multitasking
- `OSTaskCreate(task, pdata, &stack[stack_size-1], prio);` — create a task
- `OSTimeDly(ticks);` / `OSTimeDlyHMSM(h,m,s,ms);` — delay task
- `OSQCreate(&qstorage[0], size);` — create queue
- `OSQPost(q, message);` — post to queue
- `OSQPend(q, timeout, &err);` — pend on queue
- `OSSemCreate(count);` — create semaphore
- `OSSemPost` / `OSSemPend` — semaphore ops

Ready-to-build Example Files

Below are **complete example snippets** you can drop into a Keil µVision project for two targets: **STM32F103 (Cortex-M3)** and **LPC2148 (ARM7/TDMI)**. These are minimal but functional µC/OS-II applications that create three tasks and use a queue for inter-task communication. Use the Board Support Package (startup, CMSIS headers, linker script) provided with your board.

> **Note:** These snippets are written to be easy to adapt. Change pin names and peripheral init calls to match your specific board.

A. STM32F103 (Cortex-M3) — Keil µVision

Files to add to project

- `main.c`

- os_cfg.h (or merge into your existing one)
- Add µC/OS-II core files (os_cpu.c, os_cpu_a.s/os_cpu_c.c depending on port, os_core.c, etc.)
- Ensure SystemInit() and startup_stm32.s are present (from CMSIS/device support)

main.c

```
#include "stm32f10x.h"
#include "includes.h"   // µC/OS-II include header (from your port)

#define TASK_STK_SIZE 256

OS_STK TaskLedStk[TASK_STK_SIZE];
OS_STK TaskUARTStk[TASK_STK_SIZE];
OS_STK TaskSensorStk[TASK_STK_SIZE];

OS_EVENT *sensorDataQ;
void *sensorQStorage[10];

void SysTick_Handler(void) {
    OSIntEnter();
    OSTimeTick();
    OSIntExit();
}

void Delay_ms(uint32_t ms) { OSTimeDly(ms); }

void init_hardware(void) {
    // Enable GPIO clocks (example for BluePill: PC13 LED)
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;
    // PC13 as push-pull output
    GPIOC->CRH &= ~(0xF << 20);
    GPIOC->CRH |=  (0x1 << 20);
    // UART and ADC init: implement per board or use stubs
}

void TaskLED(void *pdata) {
    (void)pdata;
    for (;;) {
        GPIOC->ODR ^= (1<<13);
        OSTimeDlyHMSM(0,0,0,200);
    }
}

void TaskSensor(void *pdata) {
    (void)pdata;
    INT32U sensor_val = 0;
    for (;;) {
```

```
        // Replace with real ADC read
        sensor_val += 1; if (sensor_val>1023) sensor_val = 0;
        OSQPost(sensorDataQ, (void*)sensor_val);
        OSTimeDlyHMSM(0,0,0,300);
    }
}

void TaskUART(void *pdata) {
    (void)pdata;
    void *msg;
    INT8U err;
    for (;;) {
        msg = OSQPend(sensorDataQ, 0, &err);
        if (msg != (void*)0) {
            INT32U val = (INT32U)msg;
            // Implement UART send (blocking or via DMA)
            // For simple debug, you may use semi-hosting or ITM
        }
        OSTimeDlyHMSM(0,0,0,500);
    }
}

int main(void) {
    SYSCLK_CONFIG(); // if you have system clock setup, else call SystemInit(
)
    init_hardware();

    OSInit();

    sensorDataQ = OSQCreate(&sensorQStorage[0], sizeof(sensorQStorage)/sizeof
(void*));

    // Setup SysTick for OS tick (1ms)
    SysTick_Config(SystemCoreClock/1000);

    OSTaskCreate(TaskLED, (void*)0, &TaskLedStk[TASK_STK_SIZE-1], 10);
    OSTaskCreate(TaskUART, (void*)0, &TaskUARTStk[TASK_STK_SIZE-1], 5);
    OSTaskCreate(TaskSensor, (void*)0, &TaskSensorStk[TASK_STK_SIZE-1], 3);

    OSStart();
    while (1) {}
}
```

os_cfg.h (minimum relevant settings)

```
#define OS_TASKS            8
#define OS_MAX_EVENTS       10
#define OS_MAX_QS           5
#define OS_TASK_IDLE_STK_SIZE 128
#define OS_STK_GROWTH       0 // Cortex-M: stack grows down
```

```
#define OS_TICKS_PER_SEC    1000
// Enable features you need
#define OS_MUTEX_EN         1
#define OS_Q_EN             1
#define OS_SEM_EN           1
```

Keil tips

- Add µC/OS-II source files to project and set optimization to 'None' for easier debugging.
- Retarget `fputc` to UART if using `printf`.
- Ensure `SysTick_Handler` has correct name and is not optimized out.

LPC2148 (ARM7 TDMI)

Files to add

- `main.c`
- `os_cfg.h`
- µC/OS-II port files for ARM7 (os_cpu_a.s / os_cpu.c) included
- Timer0 interrupt configured for OS tick

main.c (LPC214x minimal)

```
#include <lpc214x.h>
#include "includes.h"

#define TASK_STK_SIZE 256
OS_STK TaskLedStk[TASK_STK_SIZE];
OS_STK TaskUARTStk[TASK_STK_SIZE];
OS_STK TaskSensorStk[TASK_STK_SIZE];

OS_EVENT *sensorDataQ;
void *sensorQStorage[10];

void timer0_init_for_tick(void) {
    T0TCR = 0x02; // reset
    T0PR = (VPB_CLOCK/1000) - 1; // prescaler for 1ms tick
    T0MR0 = 1; // count
    T0MCR = 3; // interrupt and reset on MR0
    VICIntSelect &= ~(1<<4); // clear for timer0
    VICIntEnable |= (1<<4);
    T0TCR = 1; // start
}

__irq void timer0_isr(void) {
    VICVectAddr = 0; // Clear vector
    OSIntEnter();
    OSTimeTick();
```

```
    OSIntExit();
}

void TaskLED(void *pdata) {
    (void)pdata;
    for (;;) {
        IO0PIN ^= (1<<10); // example LED pin
        OSTimeDly(200);
    }
}

void TaskSensor(void *pdata) {
    (void)pdata;
    INT32U cnt=0;
    for(;;) {
        cnt++;
        OSQPost(sensorDataQ, (void*)cnt);
        OSTimeDly(300);
    }
}

void TaskUART(void *pdata) {
    (void)pdata;
    void *msg; INT8U err;
    for(;;) {
        msg = OSQPend(sensorDataQ, 0, &err);
        if (msg) {
            // send over UART
        }
        OSTimeDly(500);
    }
}

int main(void) {
    // Board init: enable GPIO/PIO etc.
    OSInit();
    sensorDataQ = OSQCreate(&sensorQStorage[0], 10);
    timer0_init_for_tick();

    OSTaskCreate(TaskLED, (void*)0, &TaskLedStk[TASK_STK_SIZE-1], 10);
    OSTaskCreate(TaskUART, (void*)0, &TaskUARTStk[TASK_STK_SIZE-1], 5);
    OSTaskCreate(TaskSensor, (void*)0, &TaskSensorStk[TASK_STK_SIZE-1], 3);

    OSStart();
    for(;;);
}
```

os_cfg.h (LPC2148)

```
#define OS_TASKS            8
#define OS_MAX_EVENTS       10
#define OS_Q_EN             1
#define OS_TICKS_PER_SEC    1000
#define OS_STK_GROWTH       1 // ARM7 port may use growth up config; check yo
ur port
```

Experiment No : 2

Title

Semaphore as Signaling & Synchronizing on ARM7 / ARM Cortex-M using µC/OS-II

Objective

- Understand semaphore concepts (binary and counting) and their uses for signaling and synchronization in real-time systems.
- Implement semaphores in µC/OS-II to coordinate tasks, protect shared resources, and signal from ISRs.
- Demonstrate at least three tasks using semaphores on an ARM7 (e.g., LPC2148) or ARM Cortex-M (e.g., STM32F1) development board.

Learning Outcomes

Students will be able to:

- Differentiate binary vs counting semaphores and choose the correct type for an application.

-Use `OSSemCreate`, `OSSemPend`, and `OSSemPost` correctly in tasks and from ISRs (using OSIntEnter/Exit).

- Use semaphores to implement producer-consumer, mutual exclusion, and ISR-to-task signaling patterns. - Diagnose semaphore-related bugs (missed posts, deadlocks, priority inversion) and mitigate them.

Prerequisites

- Embedded C programming skills
- Basic RTOS concepts: tasks, priorities, scheduling
- Familiarity with ARM7 or Cortex-M development environment and toolchain (Keil µVision or GCC)

Hardware Required

- Development board: LPC2148 (ARM7) or STM32F103 (ARM Cortex-M3) or equivalent
- LEDs (3) + resistors
- Push buttons (2)
- Buzzer or additional LED for ISR signaling (optional)
- USB debugger (JTAG/SWD) and serial terminal

Software Required

- Keil µVision or GCC ARM toolchain
- µC/OS-II source code (appropriate port for the MCU)

- Board support package (startup, linker, system files)
- Terminal emulator (PuTTY/TeraTerm)

Theory:

Semaphore: a kernel-managed counter used to control access to resources or signal events between tasks/ISRs.

- **Binary semaphore:** 0 or 1 — used for signaling (event notification) or simple mutual exclusion when priority inheritance is not required.
- **Counting semaphore:** integer ≥ 0 — used to represent a pool of identical resources (e.g., N buffers) or count events.

Common µC/OS-II API: - `OS_EVENT *OSSemCreate(INT16U cnt);` — create semaphore initialized with `cnt`. - `INT8U OSSemPost(OS_EVENT *pevent);` — increment semaphore (signal / release). - `INT16U OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *perr);` — wait on semaphore (decrement) — blocking or timed.

ISR notes: when signaling from an ISR, wrap OS calls with `OSIntEnter()` and `OSIntExit()` and use the ISR-safe version of the primitive (most ports allow OSSemPost from ISR). Always check your port docs.

Experiment Overview

We will implement three demonstration scenarios (each is a sub-experiment) using at least three tasks:

6. **ISR-to-Task Signaling:** A timer (or external interrupt from a push-button) signals a task via a binary semaphore. Task wakes up and toggles an LED or processes the event.
7. **Producer-Consumer (Buffer Pool):** Producer task produces data (simulated sensor reading) and posts to a counting semaphore representing available buffer slots; consumer task waits on the semaphore to consume data and frees the slot.
8. **Resource Protection / Mutual Exclusion:** Two tasks access a shared buffer/resource. Use a binary semaphore to serialize access and prevent race conditions.

Each scenario includes test steps, expected observations and debugging tips.

Design & Architecture

- OS initialization (`OSInit`) and tick setup (SysTick for Cortex-M, Timer for ARM7).
- Create semaphores before creating tasks.

- Use distinct priorities: ISR > consumer/worker (higher priority) > less critical tasks.
- Demonstrate proper ISR signaling using `OSIntEnter()`/`OSIntExit()`.

Suggested Priorities

- ISR-signal task (Event Handler Task): Priority 3 (high)
- Producer / Sensor Task: Priority 5
- Logger / Low priority task: Priority 10 (low)

Lower numeric priority value means higher scheduling priority in µC/OS-II.

Implementation Steps

1. Set up project and µC/OS-II files (see prerequisite lab manual if needed).
2. Configure `OS_CFG.H` (set `OS_Q_EN`, `OS_SEM_EN`, `OS_TASKS`, `OS_TICKS_PER_SEC`).
3. Implement hardware init and tick Timer.
4. Create semaphores in `main()` (or startup task): `OS_EVENT *binSem`, `OS_EVENT *countSem` etc.
5. Create the tasks with stacks and priorities.
6. Implement ISR to post semaphore and wrap with `OSIntEnter/Exit`.
7. Run and observe behavior on LEDs and UART prints.

Example Code — LPC2148 (ARM7)

```
#include <lpc214x.h>
#include "includes.h"

#define TASK_STK_SIZE 256
OS_STK TaskEventStk[TASK_STK_SIZE];
OS_STK TaskProducerStk[TASK_STK_SIZE];
OS_STK TaskConsumerStk[TASK_STK_SIZE];

OS_EVENT *binSem;
OS_EVENT *countSem;

__irq void EINT0_ISR(void) {
    VICVectAddr = 0;
    OSIntEnter();
    // clear external interrupt flag
    EXTINT = 0x01;
    OSSemPost(binSem);
    OSIntExit();
}
```

```
void TaskEventHandler(void *pdata) {
    INT8U err;
    for(;;) {
        OSSemPend(binSem, 0, &err);
        if (err == OS_ERR_NONE) {
            IO0PIN ^= (1<<10); // toggle LED
        }
    }
}

void TaskProducer(void *pdata) {
    INT8U err; INT32U i=0;
    for(;;) {
        OSSemPend(countSem, 0, &err);
        if (err==OS_ERR_NONE) {
            // store data in buffer or OSQPost
            i++;
        }
        OSTimeDly(200);
    }
}

void TaskConsumer(void *pdata) {
    INT8U err;
    for(;;) {
        // msg = OSQPend(dataQ, 0, &err);
        // process
        OSSemPost(countSem); // release buffer slot
        OSTimeDly(100);
    }
}

int main(void) {
    // hardware init
    OSInit();
    countSem = OSSemCreate(3);
    binSem = OSSemCreate(0);
    // setup EINT0 with VIC vector and enable

    // setup timer0 for tick
    timer0_init_for_tick();

    OSTaskCreate(TaskEventHandler, NULL, &TaskEventStk[TASK_STK_SIZE-1], 3);
    OSTaskCreate(TaskProducer, NULL, &TaskProducerStk[TASK_STK_SIZE-1], 5);
    OSTaskCreate(TaskConsumer, NULL, &TaskConsumerStk[TASK_STK_SIZE-1], 7);

    OSStart();
    while(1);
}
```

Testing Procedure (Lab Steps)

Part A — ISR-to-Task Signaling

1. Build and flash firmware.
2. Press the external button (or trigger the interrupt). The ISR posts `binSem`.
3. Observe that `TaskEventHandler` wakes and toggles LED or logs message.
4. Repeat rapidly — ensure no missed events (if needed, measure with oscilloscope).

Part B — Producer-Consumer with Counting Semaphore

1. Ensure `countSem` initialized to 3 (three free buffer slots).
2. Observe Producer posts data only if a slot is available. Consumer frees a slot after processing.
3. If Producer runs faster than Consumer, Producer should block on `OSSemPend` until Consumer posts.

Part C — Resource Protection (Mutual Exclusion via Binary Semaphore)

1. Replace the counting semaphore with a binary semaphore used as a mutex (initial 1).
2. Two tasks that access a shared variable must `OSSemPend` before access and `OSSemPost` after.
3. Verify no data races by toggling a shared variable and printing from UART.

Expected Observations

- ISR-triggered events are reliably delivered to waiting task using binary semaphore.
- Counting semaphore enforces available buffer slots and blocks producers if full.
- Proper mutual exclusion prevents race conditions.

Debugging & Common Issues

- **Missed ISR posts:** Ensure ISR clears the hardware interrupt flag before posting and uses `OSIntEnter/Exit` wrappers.
- **Semaphores never posted:** Confirm semaphores created before tasks start and ISR can access semaphore variable.
- **Deadlock:** Avoid having tasks hold semaphores while waiting for other semaphores (circular wait). Keep critical sections short.
- **Priority inversion:** Binary semaphores don't provide priority inheritance. If priority inversion is a concern, design with priority regime or use a mutex primitive that supports inheritance (check your RTOS/version).

- **Stack overflow:** Increase task stacks if unexpected crashes occur. Use stack-coloring or canary techniques.

Viva Questions

1. Explain difference between binary and counting semaphores with examples.
2. Why must `OSIntEnter()` / `OSIntExit()` be used when an ISR interacts with µC/OS-II?
3. Describe a situation where a semaphore can cause priority inversion and how to mitigate it.
4. How would you modify the example to use a message queue instead of semaphores for producer-consumer?

Experiment No: 3

Title

Mailbox Implementation for Message Passing on ARM7 / ARM Cortex-M

Objective

1. Implement inter-task message passing using a mailbox abstraction on ARM7 (with µC/OS-II) and ARM Cortex-M (with FreeRTOS/CMSIS-RTOS).
2. Demonstrate producer-consumer communication using mailboxes (single-message mailbox and multi-message queue).
3. Measure and observe blocking/non-blocking behavior, message integrity, and synchronization.

Apparatus / Software Required

- Development board:
    - ARM7 (e.g., Keil demo board with LPC2148) **or**
    - ARM Cortex-M (e.g., STM32F4Discovery, NUCLEO-F103)
- Toolchain/IDE:
    - Keil MDK (uVision) for ARM7 / Cortex-M (or GCC + OpenOCD)
- RTOS:
    - µC/OS-II (for ARM7 example)
    - FreeRTOS (for Cortex-M example)
- Serial terminal on PC (PuTTY/TERMINAL) for observing printf logs
- USB cable / JTAG debugger
- C compiler and linker (installed with Keil or toolchain)

Theory :

- **Mailbox**: an inter-task communication mechanism that stores a message (pointer or data) that one task posts and another task pends (reads). A mailbox typically allows a single message (pointer) to be stored; for multiple messages use a queue.
- **µC/OS-II API** (typical):

    - OS_EVENT *OSMboxCreate(void *pmsg);
    - INT8U OSMboxPost(OS_EVENT *pevent, void *pmsg);
    - void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *perr);
- **FreeRTOS API** (mailbox behavior via queues):
    - xQueueHandle xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
    - xQueueSend(xQueue, &item, portMAX_DELAY);
    - xQueueReceive(xQueue, &item, portMAX_DELAY);
- **Blocking vs Non-blocking**: Mailbox pend can block until message available (with timeout) or return immediately (non-blocking). Posting can be from ISR or task context (use special APIs in ISR).

Design

Implement two versions:

1. **µC/OS-II (ARM7)** — Use OSMbox to pass pointers to message structs between Producer and Consumer task.
2. **FreeRTOS (Cortex-M)** — Use xQueue as mailbox/queue to send fixed-size message structures.

Messages: use a small struct

```
typedef struct {
  uint32_t id;
  uint32_t timestamp;
  uint8_t  payload[16];
} msg_t;
```

Procedure

Common steps

1. Create a new project in Keil uVision (or your IDE) for your board.
2. Add RTOS source (µC/OS-II or FreeRTOS) and configure the kernel in the project.
3. Configure system clock and UART (for printf/debug output).
4. Add the example code (below), compile and flash the binary to target.

5. Open serial terminal at the configured baud rate and observe messages.

Example A — µC/OS-II (ARM7 / Keil)

Notes: µC/OS-II mailbox holds pointer-sized messages. For multiple messages, use message queues (OSQ* APIs). The example below shows a single mailbox used to pass pointers to msg_t allocated statically.

```
/* File: mailbox_ucos2.c
   Build with µC/OS-II kernel
*/
#include "includes.h"   // µC/OS-II includes, uC/OS-II config, BSP, stdio

typedef struct {
  uint32_t id;
  uint32_t timestamp;
  uint8_t payload[16];
} msg_t;

#define TASK_PRIO_PRODUCER  5
#define TASK_PRIO_CONSUMER  6
#define TASK_STK_SIZE       512

OS_EVENT *mailbox;
msg_t producer_msg;   // single message buffer

static OS_STK TaskProdStk[TASK_STK_SIZE];
static OS_STK TaskConsStk[TASK_STK_SIZE];

void ProducerTask(void *pdata) {
    INT8U err;
```

```
    uint32_t counter = 0;
    (void)pdata;
    for (;;) {
        // prepare message
        producer_msg.id = counter++;
        producer_msg.timestamp = OSTimeGet();
        snprintf((char*)producer_msg.payload, sizeof(producer_msg.payload),
                "P%lu", (unsigned long)producer_msg.id);

        // Post pointer to mailbox (blocking not needed here)
        err = OSMboxPost(mailbox, &producer_msg);
        if (err == OS_ERR_NONE) {
            printf("Producer: posted id=%lu time=%lu\n",
                   (unsigned long)producer_msg.id,
                   (unsigned long)producer_msg.timestamp);
        } else {
            printf("Producer: mailbox post error %d\n", err);
        }

        OSTimeDlyHMSM(0,0,0,500); // 500 ms
    }
}
void ConsumerTask(void *pdata) {
    INT8U err;
    msg_t *rmsg;
    (void)pdata;
    for (;;) {
        // Pend on mailbox with timeout (0 -> no timeout, block forever)
        rmsg = (msg_t*)OSMboxPend(mailbox, 0, &err);
```

```
        if (err == OS_ERR_NONE && rmsg != NULL) {
            printf("Consumer: got id=%lu ts=%lu payload=%s\n",
                   (unsigned long)rmsg->id,
                   (unsigned long)rmsg->timestamp,
                   (char*)rmsg->payload);
        } else {
            printf("Consumer: pend error %d\n", err);
        }
        OSTimeDlyHMSM(0,0,0,100); // small processing delay
    }
}

int main(void) {
    // Board init (clock, UART, etc.)
    OSInit();
    mailbox = OSMboxCreate(NULL);

    OSTaskCreate(ProducerTask, (void*)0, &TaskProdStk[TASK_STK_SIZE-1],
                 TASK_PRIO_PRODUCER);
    OSTaskCreate(ConsumerTask, (void*)0, &TaskConsStk[TASK_STK_SIZE-1],
                 TASK_PRIO_CONSUMER);

    OSStart(); // start multitasking
    return 0;
}
```

Build/Run:

- Ensure printf is retargeted to UART or semihosting.
- Compile, flash, open serial terminal (e.g., 115200 8N1).
- You should see alternating Producer posts and Consumer receives.

Notes:

- OSMbox holds a single message pointer; if Producer posts when mailbox already contains a pointer, behavior depends on µC/OS-II version/error codes (usually returns OS_ERR_EVENT_FULL).
- For buffering multiple messages, use OSQCreate/OSQPost/OSQPend.

Example B — FreeRTOS (ARM Cortex-M)

Notes: FreeRTOS queues are flexible and can hold multiple fixed-size items. Use xQueueCreate(length, sizeof(msg_t)) to create a queue mailbox.

```
/* File: mailbox_freertos.c
   Build with FreeRTOS
*/
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>

typedef struct {
  uint32_t id;
  uint32_t timestamp;
  uint8_t payload[16];
} msg_t;

QueueHandle_t xMailbox = NULL;

void vProducerTask(void *pvParameters) {
    msg_t out;
    uint32_t cnt = 0;
    (void)pvParameters;
    for (;;) {
```

```
        out.id = cnt++;
        out.timestamp = (uint32_t)xTaskGetTickCount();
        snprintf((char*)out.payload, sizeof(out.payload), "P%lu", (unsigned long)out.id);

        // send to queue (block up to 100 ms if full)
        if (xQueueSend(xMailbox, &out, pdMS_TO_TICKS(100)) == pdPASS) {
            printf("Producer: sent id=%lu\n", (unsigned long)out.id);
        } else {
            printf("Producer: queue full, drop id=%lu\n", (unsigned long)out.id);
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vConsumerTask(void *pvParameters) {
    msg_t in;
    (void)pvParameters;
    for (;;) {
        // wait indefinitely
        if (xQueueReceive(xMailbox, &in, portMAX_DELAY) == pdPASS) {
            printf("Consumer: recv id=%lu ts=%lu payload=%s\n",
                   (unsigned long)in.id,
                   (unsigned long)in.timestamp,
                   (char*)in.payload);
        } else {
            printf("Consumer: receive failed\n");
        }
        vTaskDelay(pdMS_TO_TICKS(100));
    }
```

```
}

int main(void) {
   // system init, UART init for printf
   xMailbox = xQueueCreate(5, sizeof(msg_t)); // mailbox with 5 slots

   if (xMailbox == NULL) {
      printf("Failed to create queue\n");
      while (1);
   }

   xTaskCreate(vProducerTask, "Producer", 256, NULL, 2, NULL);
   xTaskCreate(vConsumerTask, "Consumer", 256, NULL, 2, NULL);

   vTaskStartScheduler(); // should never return
   for (;;);
   return 0;
}
```

Build/Run:

- Configure FreeRTOSConfig.h for kernel tick and priorities.
- Compile, flash, and open serial terminal.
- Expect sequential "sent" and "recv" messages. Because queue length = 5, Producer can send up to 5 messages before the queue is full.

ISR-safe posting:

- From ISR use xQueueSendFromISR() and OSMboxPost() ISR variants if available.

Testing / Observation

Test cases:

1. **Basic transfer**: Producer posts every 500 ms, Consumer receives immediately — check message integrity and order.
2. **Slow consumer**: Artificially increase consumer delay to cause queue/mailbox backlog; observe blocking or dropped messages.
3. **Timeout behavior**: Pend with a timeout and check return codes when no message arrives.
4. **ISR posting**: Post from a simulated ISR and verify consumer receives.
5. **Multi-producer / multi-consumer**: Create additional producers or consumers to observe fairness and ordering.

Observation table (example)

Test No.	Producer Rate	Consumer Rate	Mailbox Type	Outcome
1	500 ms	100 ms	µC/OS-II single mailbox	Consumer receives every message (posted pointer reused)
2	100 ms	1000 ms	FreeRTOS queue(5)	Queue fills; producer gets pdPASS until queue full — then send fails if nonblocking
3	500 ms	500 ms	FreeRTOS queue(1)	Works as mailbox; acts like single-message mailbox

---

Expected Output (serial)

µC/OS-II:

Producer: posted id=0 time=10

Consumer: got id=0 ts=10 payload=P0

Producer: posted id=1 time=20

Consumer: got id=1 ts=20 payload=P1

...

FreeRTOS:

Producer: sent id=0

Consumer: recv id=0 ts=10 payload=P0

Producer: sent id=1

Consumer: recv id=1 ts=520 payload=P1

...

Result

- Demonstrated message passing using a single-pointer mailbox (µC/OS-II) and multi-slot queue (FreeRTOS).
- Verified blocking behavior, timeouts, and queue capacity effects.
- Confirmed ISR-safe posting (when implemented with proper APIs).

Discussion / Analysis

- **µC/OS-II mailbox** is lightweight for pointer passing but cannot buffer multiple messages. It's ideal when one message at a time is sufficient.
- **FreeRTOS queues** provide flexible buffering and typed item storage; use queues when ordering and buffering are necessary.
- Choose mailbox vs. queue based on throughput, memory, and required buffering.
- For large payloads, prefer passing pointers to statically allocated buffers with careful ownership rules to avoid race conditions.

Precautions / Notes

- Ensure printf usage is thread-safe or guarded; printing from multiple tasks can corrupt output.
- When passing pointers, ensure the data's lifetime outlives the use by consumer.
- Use mutexes or critical sections if multiple producers update shared buffers.
- Use ISR-safe APIs when interacting with queue/mailbox from interrupt context.

Viva Questions

1. What is a mailbox and how does it differ from a queue?
2. Which RTOS APIs implement mailboxes in µC/OS-II and FreeRTOS?
3. What are pros and cons of passing pointers vs copying message structures into mailboxes/queues?
4. How does blocking pend differ from non-blocking pend with timeout?
5. How would you post a message from an ISR safely?
6. When would you prefer a mailbox over a queue in embedded systems?
7. How do you avoid race conditions when a producer reuses a static buffer passed by pointer?

Experiment No : 4

Title:

Implementation of MUTEX using minimum 3 tasks on ARM7/ARM Cortex-M

Objective:

To implement mutual exclusion (MUTEX) mechanism using minimum 3 tasks on ARM7/ARM Cortex-M processor under an RTOS environment (e.g., µC/OS-II or FreeRTOS).

Apparatus Required:

S. No	Equipment/Software	Description
1	ARM7/ARM Cortex-M Development Board	LPC2148 (ARM7) / STM32 (Cortex-M)
2	Keil µVision IDE / STM32CubeIDE	For program development and debugging
3	Flash Magic / ST-Link Utility	For programming the microcontroller
4	Serial Cable / USB	For communication with host PC
5	µC/OS-II or FreeRTOS Library	RTOS kernel for task scheduling
6	LED or Serial Terminal	For displaying task execution results

---

Theory:

MUTEX (Mutual Exclusion):

- A **MUTEX** is a synchronization primitive used to protect shared resources from concurrent access by multiple tasks.
- Only one task can own the MUTEX at a time.
- Other tasks attempting to acquire the same MUTEX are blocked until the owning task releases it.

Key Functions in RTOS:

Function	Description
OSMutexCreate()	Creates a mutex object
OSMutexPend()	Locks or waits to acquire the mutex
OSMutexPost()	Releases the mutex
OSTaskCreate()	Creates tasks
OSStart()	Starts the RTOS scheduler

Working Principle:

- In this experiment, three tasks share a common resource (e.g., an LED or serial output).
- The MUTEX ensures that only one task accesses the shared resource at a time.
- When a task finishes using the resource, it releases the MUTEX for others.

Algorithm:

1. Initialize the RTOS and create a MUTEX.
2. Create three tasks (Task1, Task2, Task3).
3. Each task attempts to access a shared resource (e.g., LED or print statement).
4. Use OSMutexPend() before accessing the resource.
5. Use OSMutexPost() after releasing the resource.
6. Start the scheduler using OSStart().
7. Observe task execution—only one task accesses the resource at a time.

Program: (FreeRTOS Example)

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>

SemaphoreHandle_t xMutex;
```

```
void Task1(void *pvParameters)
{
   while(1)
   {
      if (xSemaphoreTake(xMutex, portMAX_DELAY))
      {
         printf("Task 1 is accessing shared resource\n");
         vTaskDelay(1000 / portTICK_PERIOD_MS);
         printf("Task 1 released the resource\n\n");
         xSemaphoreGive(xMutex);
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
   }
}

void Task2(void *pvParameters)
{
   while(1)
   {
      if (xSemaphoreTake(xMutex, portMAX_DELAY))
      {
         printf("Task 2 is accessing shared resource\n");
         vTaskDelay(1000 / portTICK_PERIOD_MS);
         printf("Task 2 released the resource\n\n");
         xSemaphoreGive(xMutex);
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
   }
}
```

```
void Task3(void *pvParameters)
{
   while(1)
   {
      if (xSemaphoreTake(xMutex, portMAX_DELAY))
      {
         printf("Task 3 is accessing shared resource\n");
         vTaskDelay(1000 / portTICK_PERIOD_MS);
         printf("Task 3 released the resource\n\n");
         xSemaphoreGive(xMutex);
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
   }
}

int main(void)
{
   xMutex = xSemaphoreCreateMutex();
   if (xMutex != NULL)
   {
      xTaskCreate(Task1, "Task1", 128, NULL, 1, NULL);
      xTaskCreate(Task2, "Task2", 128, NULL, 1, NULL);
      xTaskCreate(Task3, "Task3", 128, NULL, 1, NULL);
      vTaskStartScheduler();
   }
   for (;;);
}
```

Expected Output:

Task 1 is accessing shared resource

Task 1 released the resource

Task 2 is accessing shared resource

Task 2 released the resource

Task 3 is accessing shared resource

Task 3 released the resource

- Tasks take turns accessing the shared resource.
- No two tasks use the resource simultaneously.

Result:

The MUTEX-based synchronization was successfully implemented using three tasks on ARM7/ARM Cortex-M. The shared resource was accessed by one task at a time, demonstrating mutual exclusion.

Viva Questions:

1. What is the difference between a semaphore and a mutex?
2. Why do we need mutexes in RTOS applications?
3. What happens if a task forgets to release a mutex?
4. Can a mutex be used for inter-task signaling?
5. How does priority inversion occur in mutex-based systems?

Experiment No: 5

Title:

Interfacing Sensors and Actuators with Arduino Uno – Door Opener using Ultrasonic Sensor and Servo Motor

---

Objective:

To design and implement an automatic door-opening system using an **ultrasonic sensor** to detect the presence of an object (person) and a **servo motor** to open or close the door automatically.

Apparatus / Components Required:

S.No	Component	Specification	Quantity
1	Arduino Uno board	ATmega328P based	1
2	Ultrasonic sensor	HC-SR04	1
3	Servo motor	SG90 micro servo (180° rotation)	1
4	Breadboard	–	1
5	Jumper wires	Male-to-male / Female-to-male	As required
6	USB cable	Type A to B	1
7	Power supply	5V (via Arduino)	1

Theory:

1. Ultrasonic Sensor (HC-SR04):

The HC-SR04 ultrasonic sensor measures distance using sound waves. It sends out an ultrasonic pulse and measures the time it takes for the echo to return.
The distance is calculated using the formula:

Distance (cm)=Time (µs)×0.03432\text{Distance (cm)} = \frac{\text{Time (µs)} × 0.0343}{2}Distance (cm)=2Time (µs)×0.0343

- **Trigger pin:** Sends ultrasonic pulse (10 µs HIGH signal).
- **Echo pin:** Receives reflected pulse; the pulse duration indicates distance.

2. Servo Motor:

A **servo motor** is an actuator that rotates to a specific angle, typically between 0° and 180°, based on the control signal.

- It uses **PWM (Pulse Width Modulation)** for control.
- A pulse of 1 ms moves the servo to 0°, 1.5 ms to 90°, and 2 ms to 180° approximately.

3. Working Principle of the System:

- The ultrasonic sensor continuously measures distance in front of the door.
- When an object (person) is detected within a threshold distance (e.g., < 30 cm), the Arduino sends a control signal to the servo motor to rotate (open the door).
- After a delay (e.g., 5 seconds), the servo motor returns to its initial position (close the door).

Circuit Diagram:

Connections:

- Ultrasonic Sensor
  - VCC → 5V (Arduino)
  - GND → GND
  - TRIG → Digital Pin 9
  - ECHO → Digital Pin 10
- Servo Motor
  - VCC → 5V
  - GND → GND
  - Signal → Digital Pin 6

Algorithm:

1. Start the Arduino system.

2. Initialize the ultrasonic sensor pins and the servo motor.
3. Send a trigger pulse to the ultrasonic sensor.
4. Measure the echo pulse duration.
5. Calculate the distance.
6. If the distance is less than 30 cm:
    - Rotate the servo to 90° (door open).
    - Wait for 5 seconds.
    - Rotate the servo back to 0° (door closed).
7. Repeat steps 3–6 continuously.

Program Code (Arduino):

```
#include <Servo.h>
Servo doorServo;

int trigPin = 9;
int echoPin = 10;
long duration;
int distance;

void setup() {
  Serial.begin(9600);
  doorServo.attach(6);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  doorServo.write(0); // Door closed
}

void loop() {
  // Trigger ultrasonic sensor
```

```
 digitalWrite(trigPin, LOW);
 delayMicroseconds(2);
 digitalWrite(trigPin, HIGH);
 delayMicroseconds(10);
 digitalWrite(trigPin, LOW);

 // Measure echo time
 duration = pulseIn(echoPin, HIGH);
 distance = duration * 0.034 / 2;

 Serial.print("Distance: ");
 Serial.print(distance);
 Serial.println(" cm");

 // Door operation
 if (distance < 30) {
   doorServo.write(90);   // Open door
   delay(5000);          // Wait 5 seconds
   doorServo.write(0);    // Close door
 }

 delay(500);
}
```

Procedure:

1. Connect all components as per the circuit diagram.
2. Open the Arduino IDE on your computer.
3. Select the correct board (**Arduino Uno**) and port.

4. Copy and paste the above code into the IDE.
5. Upload the program to the Arduino board.
6. Place an object (e.g., your hand) near the ultrasonic sensor and observe the servo motor.
7. The servo should rotate when an object is detected within the threshold distance.

Observation Table:

S.No	Distance (cm)	Servo Angle (°)	Door Status
1	> 30	0	Closed
2	< 30	90	Open

Result:

An automatic door opener system using Arduino Uno, ultrasonic sensor, and servo motor was successfully designed and implemented. The door opens automatically when an object is detected within a specific range and closes after a delay.

Applications:

- Automatic doors in malls, offices, and hospitals.
- Touchless entry systems for hygiene and convenience.
- Smart home automation systems.

Viva Questions:

1. What is the function of the ultrasonic sensor in this project?
2. How does the servo motor differ from a DC motor?
3. What is PWM, and how is it used to control servo motors?
4. Why do we divide the time by two in the distance calculation?
5. Can this system be implemented for multiple doors? If yes, how?

Experiment: 6

Title:

Weather Station - Build a Cloud-Ready Temperature and Humidity Sensor (DHT11/22) with NodeMCU and IoT Platform

---

Objective:

To design and implement a simple weather monitoring system that measures **temperature and humidity** using the **DHT11/DHT22 sensor**, processes the data through **NodeMCU (ESP8266)**, and uploads it to a **cloud-based IoT platform** for real-time monitoring.

Apparatus / Components Required:

Sl. No.	Component	Specification	Quantity
1	NodeMCU ESP8266	Wi-Fi-enabled microcontroller	1
2	DHT11 or DHT22 sensor	Temperature and humidity sensor	1
3	Jumper wires	Male-to-Female	As required
4	Breadboard	Standard	1
5	USB cable	For NodeMCU connection	1
6	Laptop/PC	Arduino IDE installed	1
7	IoT Platform	Thingspeak / Blynk / Ubidots / MQTT Dashboard	1 account

---

Theory:

1. NodeMCU (ESP8266):

NodeMCU is an open-source IoT platform based on the ESP8266 Wi-Fi module. It includes firmware that runs on the ESP8266 Wi-Fi SoC and hardware based on the ESP-12 module. It allows users to connect sensors and send data to the Internet via Wi-Fi.

2. DHT11/DHT22 Sensor:

The DHT11/DHT22 sensors are widely used digital sensors that measure both temperature and humidity.

- DHT11: Measures temperature (0–50°C) and humidity (20–90%) with moderate accuracy.
- DHT22: Measures temperature (−40–80°C) and humidity (0–100%) with better accuracy.

3. IoT Concept:

The Internet of Things (IoT) connects physical devices to the Internet for monitoring and control. In this experiment, the NodeMCU acts as an IoT node that uploads sensor data to a cloud platform (like ThingSpeak or Blynk) for visualization and analysis.

Circuit Diagram:

Connections:

DHT11 Pin	NodeMCU Pin
VCC	3.3V
DATA	D4 (GPIO2)
GND	GND

(Use a 10kΩ pull-up resistor between VCC and DATA pin for stable output)

Procedure:

1. **Hardware Setup:**
    - Connect the DHT11/DHT22 sensor to the NodeMCU as per the connection table.
    - Ensure all connections are tight and properly placed on the breadboard.
2. **Software Setup:**
    - Install **Arduino IDE**.
    - Go to **File → Preferences** → Add the URL: http://arduino.esp8266.com/stable/package_esp8266com_index.json
    - Install the **ESP8266 Board Package** via **Board Manager**.
    - Install required libraries:
        - *DHT sensor library* by Adafruit
        - *Adafruit Unified Sensor*
        - *WiFi client library*
        - *IoT platform library* (e.g., ThingSpeak or Blynk)

3. **Code Upload:**
    - Open the provided code in Arduino IDE.
    - Replace WiFi_SSID, WiFi_PASSWORD, and API keys from your IoT platform.
    - Select **Tools → Board → NodeMCU 1.0 (ESP-12E Module)**.
    - Connect NodeMCU via USB and upload the code.
4. **Cloud Setup:**
    - Create an account on an IoT platform (e.g., **ThingSpeak**).
    - Create a new **Channel** with two fields:
        - Field 1: Temperature
        - Field 2: Humidity
    - Copy the **Write API Key** and paste it into the Arduino code.
    - Monitor real-time updates on the ThingSpeak dashboard.

Sample Code:

```
#include "DHT.h"
#include <ESP8266WiFi.h>
#include "ThingSpeak.h"

#define DHTPIN D4
#define DHTTYPE DHT11  // Change to DHT22 if using DHT22
DHT dht(DHTPIN, DHTTYPE);
const char* ssid = "Your_WiFi_SSID";
const char* password = "Your_WiFi_Password";
WiFiClient client;
unsigned long myChannelNumber = YOUR_CHANNEL_NUMBER;
const char * myWriteAPIKey = "YOUR_API_KEY";

void setup() {
```

```
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");
  dht.begin();
  ThingSpeak.begin(client);
}
void loop() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  Serial.print("Temperature: ");
  Serial.print(t);
  Serial.print(" °C, Humidity: ");
  Serial.print(h);
  Serial.println(" %");
  ThingSpeak.setField(1, t);
  ThingSpeak.setField(2, h);
  int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  if (x == 200) {
    Serial.println("Data successfully sent to ThingSpeak");
  } else {
    Serial.println("Problem sending data. HTTP error code " + String(x));
  }
  delay(20000); // Update every 20 seconds
}
```

Observation Table:

Sl. No.	Temperature (°C)	Humidity (%)	Time (HH:MM:SS)
1			
2			
3			

Result:

The real-time temperature and humidity data were successfully measured using the DHT11/DHT22 sensor and uploaded to the cloud IoT platform using NodeMCU. The data can be visualized graphically on the platform dashboard.

Conclusion:

The experiment successfully demonstrates how to interface the DHT11/DHT22 sensor with NodeMCU and upload environmental data to a cloud platform. This forms the foundation for IoT-based weather monitoring systems.

Applications:

- Smart home weather monitoring
- Greenhouse monitoring
- IoT-based remote weather stations

Viva Questions:

1. What is the role of the NodeMCU in IoT applications?
2. How does the DHT11/DHT22 sensor measure humidity and temperature?
3. What is the difference between DHT11 and DHT22?
4. Why is Wi-Fi connectivity important in IoT systems?
5. What type of protocol does ThingSpeak use to send data?
6. How can you visualize and analyze sensor data on the cloud?

Experiment No: 7

Title:

IoT Based Wireless Controlled Home Automation using ESP8266

Objective:

To design and implement a home automation system that allows users to control electrical appliances wirelessly using the **ESP8266 NodeMCU** and a suitable **IoT platform (like Blynk or ThingSpeak)** through the Internet.

Apparatus / Components Required:

Sl. No.	Component	Specification	Quantity
1	NodeMCU ESP8266	Wi-Fi-enabled microcontroller	1
2	Relay Module (5V)	Single channel / 4-channel	1
3	Light Bulb / Fan / LED	AC/DC load	As required
4	Jumper Wires	Male–Female / Male–Male	As required
5	Breadboard	Standard	1
6	Power Supply	5V DC	1
7	USB Cable	For NodeMCU connection	1
8	IoT Platform	Blynk App / ThingSpeak / MQTT Dashboard	1 account

---

Theory:

1. Internet of Things (IoT):

IoT connects physical devices to the Internet, allowing them to send and receive data. In this experiment, IoT is used to remotely control electrical devices via a cloud-based platform.

2. NodeMCU (ESP8266):

NodeMCU is an open-source IoT platform based on the ESP8266 Wi-Fi module. It provides GPIO pins for controlling devices and has built-in Wi-Fi capabilities, making it ideal for IoT automation applications.

3. Relay Module:

A relay acts as an electrically operated switch that allows a low-voltage circuit (from NodeMCU) to control high-voltage appliances. When the GPIO pin sends a HIGH or LOW signal, the relay toggles the connected load ON or OFF.

4. Blynk / IoT Platform:

Blynk is a mobile IoT platform that provides a graphical interface to control and monitor IoT devices. It uses virtual pins and APIs to communicate with hardware via the Internet.

Circuit Diagram:

Connections:

Relay Pin	NodeMCU Pin
VCC	3.3V or 5V
GND	GND
IN	D1 (GPIO5)

Appliance Connection:

- Connect the **COM** and **NO (Normally Open)** terminals of the relay to the AC load (e.g., bulb or fan).
- Ensure all electrical connections are properly insulated.

Procedure:

1. **Hardware Setup:**
    - Connect NodeMCU and relay as per the connection table.
    - Connect the appliance through relay terminals (COM and NO).
    - Power up the NodeMCU via USB.
2. **Software Setup:**
    - Install the **Blynk app** on your smartphone (available on Android/iOS).
    - Create a new project in Blynk and select **NodeMCU** as the device.

- Choose **Wi-Fi** as the connection type.
- Add a **Button Widget** to the dashboard and assign it to **Virtual Pin V1**.
- Note the **Auth Token** sent to your registered email.

3. **Arduino IDE Setup:**
   - Install the **Blynk Library** and **ESP8266 Board** in Arduino IDE.
   - Open Arduino IDE and paste the given code.
   - Replace Auth Token, WiFi SSID, and Password in the code.
   - Select **Tools → Board → NodeMCU 1.0 (ESP-12E Module)**.
   - Upload the program to NodeMCU.
4. **Testing:**
   - Open the Blynk app.
   - Press the button widget to toggle the relay ON/OFF.
   - Observe the connected device switching accordingly.

Sample Code:

```
#define BLYNK_TEMPLATE_ID "Your_Template_ID"
#define BLYNK_DEVICE_NAME "Home Automation"
#define BLYNK_AUTH_TOKEN "Your_Auth_Token"

#include <ESP8266WiFi.h>
#include <BlynkSimpleEsp8266.h>
char auth[] = "Your_Auth_Token";
char ssid[] = "Your_WiFi_SSID";
char pass[] = "Your_WiFi_Password";
int relayPin = D1;  // Relay connected to D1
void setup() {
  Serial.begin(115200);
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, HIGH); // Relay off initially
```

```
  Blynk.begin(auth, ssid, pass);
}
BLYNK_WRITE(V1) {  // Virtual Pin V1 from Blynk button
  int value = param.asInt();
  if (value == 1) {
    digitalWrite(relayPin, LOW); // Turn ON
    Serial.println("Appliance ON");
  } else {
    digitalWrite(relayPin, HIGH); // Turn OFF
    Serial.println("Appliance OFF");
  }
}
void loop() {
  Blynk.run();
}
```

Observation Table:

Sl. No.	Blynk Button State	Relay Output	Appliance Status
1	ON	LOW	ON
2	OFF	HIGH	OFF

Result:

The IoT-based home automation system was successfully implemented using NodeMCU and a relay module. The user could control appliances wirelessly using the Blynk mobile app through the Internet.

Conclusion:

This experiment demonstrates how IoT technology can be used to remotely control home appliances. The NodeMCU acts as a bridge between the Internet and the electrical devices, allowing wireless automation and smart control.

Applications:

- Smart Home Systems
- Remote appliance monitoring
- Energy-efficient automation
- Industrial IoT control systems

Viva Questions:

1. What is the role of the NodeMCU in IoT-based home automation?
2. How does a relay module control AC appliances?
3. What are the advantages of using Blynk in IoT applications?
4. Explain the difference between Normally Open (NO) and Normally Closed (NC) terminals in a relay.
5. What are the security challenges in IoT-based automation systems?
6. Can this system be extended to voice control? If yes, how?
7. What is the importance of cloud connectivity in home automation?

Experiment No: 8

Title:

Interfacing of 4 LED Bank with Raspberry Pi to Blink

Objective:

To interface a **4-LED bank** with the **Raspberry Pi GPIO pins** and write a **Python program** to blink the LEDs in a specific sequence.

Apparatus / Components Required:

Sl. No.	Component	Specification	Quantity
1	Raspberry Pi (3/4 Model B)	1 GB/2 GB RAM	1
2	LEDs	5mm, any color	4
3	Resistors	220Ω each	4
4	Breadboard	Standard	1
5	Jumper Wires	Male–Female	As required
6	Power Supply	5V for Raspberry Pi	1
7	SD Card	with Raspbian OS installed	1
8	HDMI Cable & Display	For monitoring output	1

Theory:

1. Raspberry Pi:

The Raspberry Pi is a small single-board computer used for learning electronics and programming. It contains **GPIO (General Purpose Input/Output)** pins that allow interfacing with sensors, actuators, and LEDs.

2. GPIO Pins:

GPIO pins are programmable pins used to control external devices.

- **GPIO Mode:** BCM (Broadcom SOC channel) or BOARD (physical pin numbering).
- **Voltage Levels:**
    - HIGH = 3.3V
    - LOW = 0V (Ground)

3. LED (Light Emitting Diode):

An LED emits light when current passes through it. A **current-limiting resistor** (220Ω) is used in series to prevent damage.

4. Python GPIO Library:

The **RPi.GPIO** library in Python allows controlling GPIO pins easily. It provides functions like:

- GPIO.setup(pin, GPIO.OUT) – to configure pin as output
- GPIO.output(pin, state) – to turn ON/OFF an LED
- time.sleep() – to provide time delay

Circuit Diagram:

Connections:

LED No.	Raspberry Pi GPIO Pin (BCM)	Physical Pin (BOARD)
LED1	GPIO17	Pin 11
LED2	GPIO18	Pin 12
LED3	GPIO27	Pin 13
LED4	GPIO22	Pin 15

- Connect the **anode (+)** of each LED to respective GPIO pins via 220Ω resistor.
- Connect all **cathodes (–)** to the **GND** pin of Raspberry Pi.

Procedure:

1. **Hardware Setup:**
    - Place 4 LEDs on the breadboard.
    - Connect each LED through a 220Ω resistor to the respective GPIO pins as per the connection table.

- Connect all GNDs of LEDs to a common GND on Raspberry Pi.
- Power ON the Raspberry Pi.

2. **Software Setup:**
   - Open the Raspberry Pi terminal or Thonny Python IDE.
   - Import the required GPIO and time libraries.
   - Set the GPIO mode (BCM or BOARD).
   - Configure LED pins as output.
3. **Program Execution:**
   - Write a Python program to blink LEDs in sequence.
   - Save the program as led_blink.py.
   - Run the program using the command:
   - python3 led_blink.py
   - Observe the LEDs blinking sequentially.

Sample Python Code:

```
# Program: Interfacing 4 LEDs with Raspberry Pi

import RPi.GPIO as GPIO
import time

# Use BCM pin numbering
GPIO.setmode(GPIO.BCM)

# Define LED pins
led_pins = [17, 18, 27, 22]

# Set all pins as output
for pin in led_pins:
    GPIO.setup(pin, GPIO.OUT)
```

```
# Blink LEDs in sequence
try:
    while True:
        for pin in led_pins:
            GPIO.output(pin, GPIO.HIGH)
            print("LED on GPIO", pin, "is ON")
            time.sleep(0.5)
            GPIO.output(pin, GPIO.LOW)
            print("LED on GPIO", pin, "is OFF")
            time.sleep(0.5)

except KeyboardInterrupt:
    print("Program stopped")
    GPIO.cleanup()
```

Observation Table:

LED No.	GPIO Pin	Blink Delay (sec)	Observation
1	GPIO17	0.5	ON/OFF alternately
2	GPIO18	0.5	ON/OFF alternately
3	GPIO27	0.5	ON/OFF alternately
4	GPIO22	0.5	ON/OFF alternately

Result:

The 4-LED bank was successfully interfaced with the Raspberry Pi. The LEDs blinked sequentially according to the programmed delay, demonstrating GPIO output control.

Conclusion:

This experiment demonstrates how to use the **GPIO pins** of the Raspberry Pi to control external devices. LEDs were blinked in a programmed sequence using a Python script, forming the basis for future automation and embedded control projects.

Applications:

- Status indicator systems
- Pattern generation (e.g., running lights)
- Learning GPIO output control
- Foundation for home automation or alert systems

Viva Questions:

1. What is the purpose of GPIO pins in Raspberry Pi?
2. What are the voltage levels of GPIO pins in Raspberry Pi?
3. Why do we use resistors with LEDs?
4. Differentiate between BCM and BOARD pin numbering.
5. What function is used to configure a GPIO pin as output in Python?
6. How is the GPIO.cleanup() function useful?
7. Can the same setup be used to control other devices like relays or buzzers? How?