School of Computer Science and Engineering
Winter Semester 2024-25
Continuous Assessment Test – I1

Programme Name & Branch:B.Tech (CSE) **SLOT: A1+TA1**
Course Name & Code: Embedded Systems – BCSE305L
Class Number (s): **Faculty Name (s): All**
Exam Duration: 90 Min. **Maximum Marks: 50**

Q. No.	Note: The evaluation must be based on the individual answers. The keys are provided just for the reference	
	Questions	**Keys**
1.	Construct an FSM Model for the Given Scenario and also Illustrate the Control Data Flow Graph (CDFG) for the system: "Autonomous Vehicle Component Classifier" **Specification:** 1. Environment: An automotive manufacturing plant where robotic systems are used for sorting, quality inspection, and packaging of vehicle components. 2. Component Storage: Bins or containers for different types of components, such as engine parts, tires, sensors, and electronic modules. 3. Component Types: Varied components like pistons, brake pads, airbags, and control units. 4. Robotic System: A mobile robot equipped with a 360° rotational arm for picking, inspecting, and placing components. **Requirements:** 1. Components must be sorted based on type, size, quality, and weight. 2. Collision avoidance must be implemented to prevent damage to components, robotic arms, or other equipment. Define relevant states, events, and actions to ensure smooth and efficient operation.	FSM Model for "Autonomous Vehicle Component Classifier" Entities: 1. Environment: Automotive manufacturing plant with robotic systems for sorting, quality inspection, and packaging. 2. Component Storage: Bins or containers for different types of components (engine parts, tires, sensors, electronic modules). 3. Component Types: Pistons, brake pads, airbags, control units, etc. 4. Robotic System: Mobile robot with a 360° rotational arm for picking, inspecting, and placing components. FSM States, Events, and Transitions States: 1. Idle: Robot is waiting for tasks. 2. Move-Forward: Robot is moving forward. 3. Move-Backward: Robot is moving backward. 4. Turn-Left: Robot is turning left. 5. Turn-Right: Robot is turning right. 6. Pick: Robot is picking a component. 7. Sort: Robot is sorting the component based on attributes. 8. Place: Robot is placing the component in a bin. 9. Halt: Robot has stopped its current action. Events: 1. Pick Component: A component is detected and needs to be picked. 2. Sort: Component needs to be sorted based on attributes. 3. Place: Component needs to be placed in a bin. 4. Bin-Empty: The bin is empty and ready to receive components. 5. Bin-Full: The bin is full and cannot accept more components.

6. Obstacle-True: An obstacle is detected in the robot's path.
7. Obstacle-False: No obstacle is detected in the robot's path.

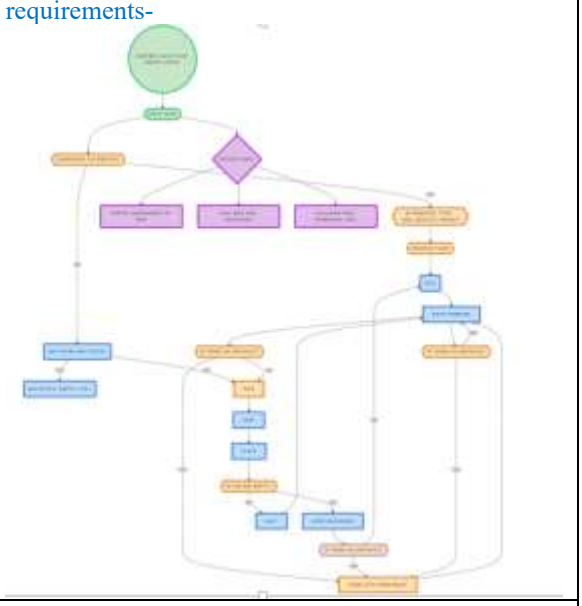
Transitions Table:

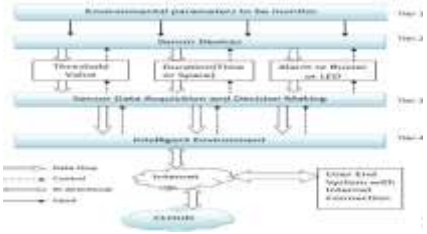
From State	Event	To State
Idle	Pick Component	Move-Forward
Move-Forward	Obstacle-True	Turn-Left/Turn-Right
Move-Forward	Obstacle-False	Pick
Pick	Sort	Sort
Sort	Place	Place
Place	Bin-Empty	Move-Backward
Place	Bin-Full	Halt
Move-Backward	Obstacle-True	Turn-Left/Turn-Right
Move-Backward	Obstacle-False	Idle
Turn-Left	Obstacle-False	Move-Forward
Turn-Right	Obstacle-False	Move-Forward
Halt	Bin-Empty	Move-Forward

Control Data Flow Graph (CDFG)

The CDFG illustrates the flow of control and data between the FSM states. Below is a textual representation:

1. Input Data:
   - Component attributes (type, size, quality, weight).
   - Bin states (empty, full).
   - Obstacle detection signals.
2. Process Flow:
   - Idle → Move-Forward: Triggered by "Pick Component" event.
   - Move-Forward → Turn-Left/Turn-Right: Triggered by "Obstacle-True" event.
   - Move-Forward → Pick: Triggered by "Obstacle-False" event.
   - Pick → Sort: Triggered by "Sort" event.
   - Sort → Place: Triggered by "Place" event.
   - Place → Move-Backward: Triggered by "Bin-Empty" event.
   - Place → Halt: Triggered by "Bin-Full" event.
   - Move-Backward → Turn-Left/Turn-Right: Triggered by "Obstacle-True" event.

		○ Move-Backward → Idle: Triggered by "Obstacle-False" event. ○ Turn-Left/Turn-Right → Move-Forward: Triggered by "Obstacle-False" event. ○ Halt → Move-Forward: Triggered by "Bin-Empty" event. 3. Output Data: ○ Sorted components in appropriate bins. ○ Full bins ready for packaging. ○ Collision-free operation logs. Note: Draw FSM and CDFG model for all the requirements- 
2.	Design and analyze a Real-Time Weather Forecast Station that continuously monitors weather conditions and provides up-to-date forecasts to users such as meteorologists, pilots, and emergency response teams. The system may include sensors for measuring temperature, humidity, barometric pressure, wind speed, and precipitation. Given Criteria: 1. Types of Events 2. Functional Correctness 3. Timeliness Evaluate the system design based on these criteria to ensure it meets the requirements for real-time weather monitoring and forecasting.	System Design Key Components Sensors: Temperature Sensor: Humidity Sensor: Barometric Pressure Sensor: Anemometer: . Rain Gauge: . Data Acquisition Unit (DAU): Real-Time Processing Unit: Communication Module: User Interface: Storage and Analytics: Evaluation Based on Given Criteria 1. Types of Events Data Collection Events: Data Processing Events: Alert Generation Events: User Interaction Events: Maintenance Events: 2. Functional Correctness Accuracy of Data Collection: Reliability of Algorithms: System Redundancy: User Feedback: 3. Timeliness Data Collection:

		Data Processing: Communication: Alert Generation: Note: Draw the weather forecast models 1. Sensors : Place sensors at the real world 2. Data Acquisition: Connect sensors to the DAU. 3. Processing: Connect the DAU to the Real-Time Processing Unit. 4. Communication: Show data flowing to the cloud or server. 5. Storage and Analytics: Represent databases and analytics engines. 6. User Interface: Show dashboards and mobile connect to user. 
3.	Analyze the requirements and design challenges involved in code optimization for embedded system design. Based on your analysis, apply some suitable optimization techniques to the provided code snippet. Reorganize the code to achieve optimal performance while ensuring that relevant validations are properly incorporated *int main() {* *float a, b, c, r1, r2, d,f,m;* *printf("Enter the values of a b c: ");* *scanf(" %f %f %f", & a, & b, & c);* *d = b * b - 4 * a * c;* *if (d > 0) {* *r1 = -b + sqrt(d) / (2 * a);* *r2 = -b - sqrt(d) / (2 * a);* *printf("The real roots = %f %f", r1, r2);* *} else if (d == 0) {* *r1 = -b / (2 * a);* *r2 = -b / (2 * a);* *printf("Roots are equal =%f %f", r1, r2);* *} else* *printf("Roots are imaginary");* *f=m*m*m;* *return 0;* *}*	Requirements of code optimization • Constrained resources a. Memory b. Processing c. Power • Design Challenges a. Time and space complexity b. Scalability – features c. Cost optimization d. Time to market – customization, faster portability, backward compatibility Optimization Techniques: Optimization Techniques Applied 1. Common Subexpression Elimination: o The expression 2 * a is calculated multiple times. It can be computed once and reused. 2. Strength Reduction: o The division operation / (2 * a) can be replaced with a multiplication by the precomputed reciprocal * inv_2a for faster computation. 3. Code Motion: o The calculation of sqrt(d) is repeated twice. It can be computed once and stored in a variable. 4. Dead code elimination : statement f = m*m*m cannot be reached

		Note: Code issues should be reorganized with rewriting correct CODE snippet.-
		#include <stdio.h>
		#include <math.h>
		int main() {
		float a, b, c, r1, r2, d;
		float inv_2a, sqrt_d; // Variables for optimization
		// Input values
		printf("Enter the values of a b c: ");
		scanf("%f %f %f", &a, &b, &c);
		// Calculate discriminant
		d = b * b - 4 * a * c;
		// **Common Subexpression Elimination**: Compute 2*a once and store its reciprocal
		inv_2a = 1.0f / (2 * a);
		if (d > 0) {
		// **Code Motion**: Compute sqrt(d) once and store it
		sqrt_d = sqrt(d);
		// **Strength Reduction**: Replace division with multiplication by precomputed reciprocal
		r1 = (-b + sqrt_d) * inv_2a;
		r2 = (-b - sqrt_d) * inv_2a;
		// Output real roots
		printf("The real roots = %f %f\n", r1, r2);
		} else if (d == 0) {
		// **Strength Reduction**: Replace division with multiplication by precomputed reciprocal
		r1 = -b * inv_2a;
		// Output equal roots
		printf("Roots are equal = %f %f\n", r1, r1);
		} else {
		// Output imaginary roots
		printf("Roots are imaginary\n");
		}
		// **Dead Code Elimination**: Removed unreachable statement `f = m * m * m`
		return 0;
		}

4.	a)Using the given dataset, show that RMS creates a feasible schedule. If the current execution times do not allow for a feasible schedule, adjust the execution times, create new values, and try RMS scheduling again. Draw a task timeline graph for at least three cycles. The dataset is as follows:. Task | Execution Time | Period T1 | 5 | 15 T2 | 7 | 10 T3 | 10 | 20 b)Evaluate how do the following parameters affect the schedulability of real time tasks:- • Arrival Time • Current Time / Scheduling point • Execution Time • Rate or Period • Deadline Suggest an optimal scheduling scheme using any three relevant parameters as mentioned above and apply it for the following dataset:- Task Arrival Time Execution Time Period Deadline Note: Consider current time as per the scheduling points. Illustrate the task time-line graph for at least three cycles.	Rate Monotonic Scheduling (RMS) Step 1: Check Schedulability Using RMS The schedulability condition for RMS is: U≤n(21/n−1) Where: • U= Total CPU utilization. • n = Number of tasks. Calculate CPU Utilization: U=C1T1+C2T2+C3T3 U=155+107+2010 U=0.333+0.7+0.5=1.533U=0.333+0.7+0.5=1.533 Calculate RMS Bound: For n=3 n(21/n−1)=3(21/3−1)≈3(1.26−1)=0.78 Since U=1.533>0.78 the tasks are not

Task	Arrival Time	Execution Time	Period	Deadline
T1	0	5	20	6
T2	0	4	15	3
T3	0	4	10	9

schedulable under RMS with the given execution times.

Step 2: Modify Execution Times

To make the tasks schedulable, we reduce the execution times such that the total CPU utilization $U \leq 0.78$

New Execution Times:

Let's reduce the execution times as follows:

- T1: 3 → 2
- T2: 4 → 3
- T3: 5 → 4

Final Dataset:

Task	Execution Time	Period
T1	2	15
T2	3	10
T3	4	20

Recalculate CPU Utilization:

U=215+310+420
U=0.133+0.3+0.2=0.633

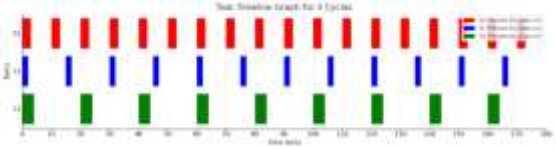
Now, $U=0.633 \leq 0.78$, so the tasks are schedulable under RMS.

Step 3: RMS Scheduling

Priorities:

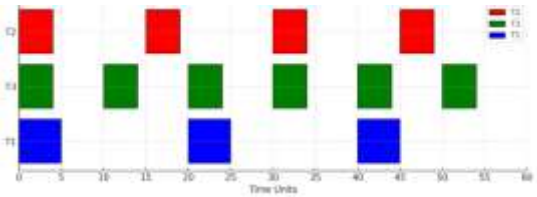
- T2 (Period = 10) has the highest priority.
- T1 (Period = 15) has the next priority.
- T3 (Period = 20) has the lowest priority.

T2 (Period = 10, Execution Time = 3):
Runs at t = 0, 10, 20, 30, 40, 50
T1 (Period = 15, Execution Time = 2):
Runs at t = 0, 15, 30, 45
T3 (Period = 20, Execution Time = 4):
Runs at t = 0, 20, 40
Note: Draw the Task Timeline graph for 3 cycles.



Second tasks set – can schedulable under EDF

- Arrival Time:
  - The time at which a task becomes ready for execution.
  - Tasks with earlier arrival times may have higher priority in scheduling decisions.
- Current Time / Scheduling Point:

		○ The time at which the scheduler makes a decision. ○ Determines which tasks are ready to be executed at that moment. • Execution Time: ○ The time required for a task to complete its execution. ○ Longer execution times may lead to higher resource utilization and potential deadline misses. • Rate or Period: ○ The interval between consecutive executions of a periodic task. ○ Tasks with shorter periods may have higher priority to ensure timely execution. • Deadline: ○ The time by which a task must complete its execution. ○ Tasks with earlier deadlines are typically given higher priority to ensure schedulability Case-: (execution time, current time & period) Every task will be successfully scheduled. Note: Draw the Task Timeline graph for three cycles. One cycle is provided; continue with the next two cycles. 
5.	A database system allows multiple clients to read data simultaneously, but only one client can write data at a time. Using POSIX semaphores, implement a solution to: • Allow concurrent reads. • Ensure exclusive access for writes. Prevent starvation of readers or writers. Explain the role of each semaphore and provide implementation with suitable code.	○ sem_wait(semaphore) – Wait to acquire a semaphore. ○ sem_post(semaphore) – Release a semaphore. ○ sem_init(semaphore, value) – Initialize a semaphore. ○ sem_destroy(semaphore) – Destroy a semaphore ○ pthread_mutex_init(): Initializes a mutex. ○ pthread_mutex_lock(): Locks the mutex (if already locked, the thread is blocked). ○ pthread_mutex_unlock(): Unlocks the mutex. ○ pthread_mutex_destroy(): Destroys the mutex when it's no longer needed. Initialize semaphores: mutex, write_sem Set reader_count to 0 Function reader(id): While true: Wait on mutex Increment reader_count If reader_count == 1: Wait on write_sem Signal mutex

		Print "Reader id is reading..." Sleep for reading time Wait on mutex Decrement reader_count If reader_count == 0: Signal write_sem Signal mutex Sleep for time before next read Function writer(id): While true: Wait on write_sem Print "Writer id is writing..." Sleep for writing time Signal write_sem Sleep for time before next write Main function: Initialize mutex and write_sem Create NUM_READERS reader threads Create NUM_WRITERS writer threads Wait for all reader and writer threads to finish Destroy mutex and write_sem