Embedded_session_10

Embedded System Session 10 - Real-Time Operating Systems (RTOS)

Created: 2025-07-24
Author: Fares Hesham Mahmoud
Tags: AVR, Embedded system, I2C, RTOS
Status: #Formatted

اللهمَّ علِّمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علمًا. وافتح علينا فتحًا عظيمًا.

I2C Communication Scenarios (Continued)

Acknowledge (ACK) Signal Patterns

ACK signals are sent in specific scenarios:

From	To	Scenario	When Sent
Slave	Master Transmitter	✅	After receiving address/data
Slave	Master Receiver	❌	Never happens
Master Transmitter	Slave	❌	Never happens
Master Receiver	Slave	✅	After receiving data

I2C Communication Examples

Master Write Data to Slave

```
START → [7-bit Addr + W(0)] → ACK → [Data Byte] → ACK → STOP
  M            M                S         M          S      M

Timeline:
1. Master sends START condition
2. Master sends slave address with write bit (0)
```

```
3. Addressed slave sends ACK
4. Master sends data byte
5. Slave sends ACK to confirm reception
6. Master sends STOP condition
```

Master Read Data from Slave

```
START → [7-bit Addr + R(1)] → ACK → [Data Byte] → NACK → STOP
  M            M                S         S           M       M

Timeline:
1. Master sends START condition
2. Master sends slave address with read bit (1)
3. Addressed slave sends ACK
4. Slave sends data byte
5. Master sends NACK (no more data needed)
6. Master sends STOP condition
```

Multi-Master Arbitration Process

```
Bus State: IDLE
Master1:   START → 0 → 1 → 0 → 1 → ...
Master2:   START → 0 → 0 → 1 → 0 → ...
Bus Line:  START → 0 → 0 → ?
                        ↑
                  Master1 loses here
                  (sent 1, read 0)

Result: Master2 continues transmission
        Master1 waits for bus to become free
```

Combined Frame (Write then Read)

```
START → Addr+W → ACK → Data → ACK → RESTART → Addr+R → ACK → Data → NACK
→ STOP
```

This allows multiple operations in a single bus transaction without releasing control.

Software Architecture Evolution

Design Goals

Requirement	Description	Importance
Determinism	Know what happens at every point in time	Critical for real-time systems
Responsiveness	How fast system responds to events	User experience, safety
Periodicity	How often specific tasks are handled	System stability, performance

Software Architecture Types

1. Super Loop Architecture

```
int main(void) {
    // Initialization
    init_hardware();

    while(1) {
        check_sensors();
        update_display();
        process_user_input();
        control_motors();
        // ... more functions
    }
}
```

Advantages ✅

- Simple: Very easy to understand and implement
- Minimal Hardware: No complex timing requirements
- Highly Portable: Works on any microcontroller
- Predictable: Sequential execution flow

Disadvantages ❌

- Poor Responsiveness: Polling-based, slow reaction to events
- Timing Issues: No guaranteed execution timing
- High Power Consumption: CPU always running at full speed
- Scalability Problems: Becomes unwieldy with many functions

2. Foreground/Background Architecture

```
// Background (main loop)
int main(void) {
    init_system();
    enable_interrupts();

    while(1) {
        background_task1();
        background_task2();
        // Low priority tasks
    }
}

// Foreground (interrupts)
ISR(TIMER_vect) {
    // High priority, time-critical tasks
    critical_sensor_reading();
    urgent_control_action();
}
```

Advantages ✅

- Good Responsiveness: Interrupts provide immediate response
- Minimal Hardware: Uses basic interrupt capability
- Priority Handling: Critical tasks get immediate attention

Disadvantages ❌

- Increased Complexity: Interrupt management required
- Timing Determinism: Difficult to predict exact timing
- Shared Resource Issues: Race conditions possible

3. Real-Time Operating System (RTOS)

> **Recommended Reading** >
>
> "Patterns for Time-Triggered Embedded Systems" - Excellent resource for RTOS design patterns

RTOS provides multitasking ability and precise timing control for each function.

Priority Hierarchy:

```
Interrupts (Highest Priority)
     ↓
   RTOS Tasks
     ↓
Super Loop (Lowest Priority)
```

RTOS Components

Core Components

Component	Function	Description
Objects	Data types	Tasks, semaphores, queues, etc.
Services	API functions	Task management, communication
Scheduler	Task execution control	Determines when/which task runs

Scheduler Algorithms

1. Shortest Task First

```
Task A: 2ms execution time
Task B: 5ms execution time
Task C: 1ms execution time

Execution order: C → A → B
```

2. Priority-Based Scheduling

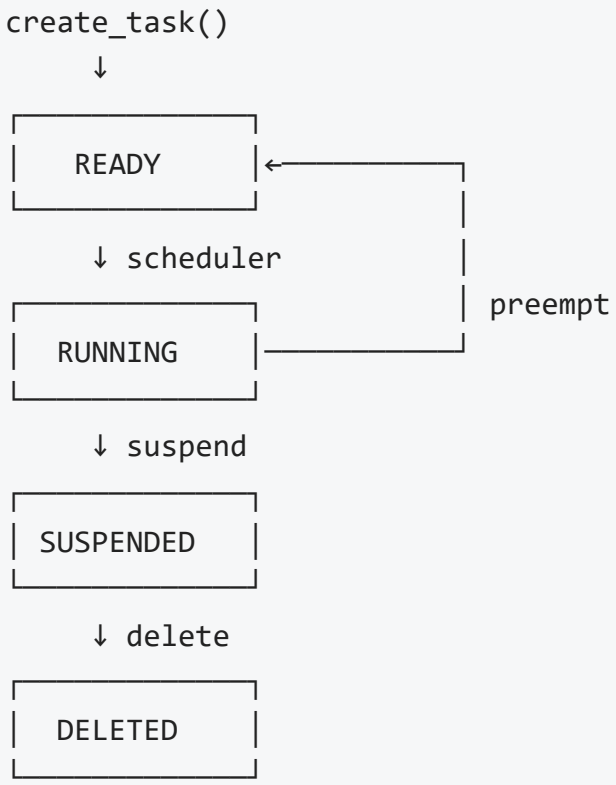
```
typedef enum {
    PRIORITY_HIGH = 1,
    PRIORITY_MEDIUM = 2,
    PRIORITY_LOW = 3
} task_priority_t;

// Higher priority tasks preempt lower priority ones
```

3. Round Robin (Time Slicing)

```
Task A: 10ms time slice
Task B: 10ms time slice
Task C: 10ms time slice
```

```
Timeline: A(10ms) → B(10ms) → C(10ms) → A(10ms) → ...
```

Task States



Delay Types in RTOS

Soft Delay (Soft Real-Time)

- Characteristic: Missing deadline is undesirable but not catastrophic
- Example: Video frame rendering, user interface updates
- Response: System continues operation with reduced quality

Hard Delay (Hard Real-Time)

- Characteristic: Missing deadline causes system failure
- Example: Airbag deployment, anti-lock braking system
- Response: System failure, potential safety hazard

Execution Time Measurement

Methods for Measuring Task Execution Time

Method	Description	Accuracy	Complexity
Assembly Analysis	Count instruction cycles	High	High
Oscilloscope	Toggle pin at start/end	Medium	Low
Software Tools	Profiling tools	High	Medium
Timer-based	Use hardware timers	High	Medium
.lss File Analysis	Assembly listing analysis	High	Medium

Oscilloscope Method

```
void task_function(void) {
    GPIO_SetPin(DEBUG_PIN);     // Set pin HIGH

    // Task code here
    sensor_read();
    calculate_output();
    update_actuator();

    GPIO_ClearPin(DEBUG_PIN);   // Set pin LOW
}
// Measure time using oscilloscope between rising and falling edges
```

Task Control Block (TCB)

Each task has its own Task Control Block stored in RAM:

```
typedef struct {
    uint8_t task_id;
    uint16_t stack_pointer;     // SP: Current stack position
    uint16_t periodicity;       // How often task runs
    task_state_t state;         // Current task state
    void (*task_function)(void); // Function pointer
    uint8_t priority;
    uint32_t next_run_time;
} tcb_t;
```

Key TCB Parameters

- Stack Pointer (SP): Points to task's stack location

- **Periodicity:** Time interval between task executions
- **Context Switch:** Process of saving/restoring task state
- **First Delay:** Initial delay before first execution

RTOS Driver Interface

Core RTOS Functions

```
// Task Management
void create_task(uint8_t task_id, uint16_t periodicity, void(*task_func)
(void));
void suspend_task(uint8_t task_id);
void resume_task(uint8_t task_id);
void delete_task(uint8_t task_id);

// Task States
typedef enum {
    TASK_READY,
    TASK_RUNNING,
    TASK_SUSPENDED,
    TASK_DELETED
} task_state_t;
```

Example RTOS Implementation

```
#define MAX_TASKS 8

typedef struct {
    uint8_t id;
    uint16_t period;
    uint32_t next_execution;
    task_state_t state;
    void (*function)(void);
} task_t;

static task_t task_list[MAX_TASKS];
static uint8_t task_count = 0;

void create_task(uint8_t id, uint16_t period, void(*func)(void)) {
    if (task_count < MAX_TASKS) {
        task_list[task_count].id = id;
        task_list[task_count].period = period;
        task_list[task_count].next_execution = get_system_time() +
```

```
period;
        task_list[task_count].state = TASK_READY;
        task_list[task_count].function = func;
        task_count++;
    }
}

void scheduler(void) {
    uint32_t current_time = get_system_time();

    for (uint8_t i = 0; i < task_count; i++) {
        if (task_list[i].state == TASK_READY &&
            current_time >= task_list[i].next_execution) {

            // Execute task
            task_list[i].state = TASK_RUNNING;
            task_list[i].function();

            // Schedule next execution
            task_list[i].next_execution += task_list[i].period;
            task_list[i].state = TASK_READY;
        }
    }
}

// Main RTOS loop
int main(void) {
    init_system();

    // Create tasks
    create_task(1, 100, sensor_task);      // Every 100ms
    create_task(2, 50,  control_task);     // Every 50ms
    create_task(3, 1000, display_task);    // Every 1000ms

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

Task Examples

```
// Periodic sensor reading task
void sensor_task(void) {
    static uint16_t sensor_value;
```

```
    sensor_value = ADC_Read(SENSOR_CHANNEL);
    // Process sensor data
}

// Control algorithm task
void control_task(void) {
    static int16_t error, output;
    error = setpoint - current_value;
    output = pid_controller(error);
    PWM_SetDutyCycle(output);
}

// User interface update task
void display_task(void) {
    LCD_Clear();
    LCD_WriteString("System Status: OK");
    LCD_WriteNumber(sensor_value);
}
```

RTOS vs Other Architectures

Feature	Super Loop	Foreground/Background	RTOS
Complexity	Low	Medium	High
Responsiveness	Poor	Good	Excellent
Determinism	Poor	Medium	Excellent
Memory Usage	Low	Medium	Higher
Power Efficiency	Poor	Medium	Good
Debugging	Easy	Medium	Complex

References

- "Patterns for Time-Triggered Embedded Systems" by Michael J. Pont
- FreeRTOS Documentation
- µC/OS-II Real-Time Kernel Documentation
- Real-Time Systems Design and Analysis