TAZROUT

IoT Irrigation System

Day 3 Learning Report

Traffic Light State Machine - Part 1

Developer: KHENFRI Moussa

Date: February 3, 2026
Week: 1 - ESP32 Fundamentals
Session Duration: 3 hours 20 minutes
Planned Duration: 2-2.5 hours
Status: Completed - Major Milestone!

Module: ESP32 Sensor & Network Communication Layer

Platform: Wokwi ESP32 Simulator

Project: TAZROUT IoT Irrigation System

Prepared as part of the 8-week ESP32 development plan
Personal learning journal and technical documentation

Contents

1 Executive Summary

Day 3 represents a significant leap in complexity and professional embedded systems development. Today I transitioned from simple sequential code to implementing a real state machine - a design pattern used in professional embedded systems worldwide. This was intellectually challenging but incredibly rewarding.

1.1 Session Overview

Metric	Value
Planned Time	2-2.5 hours
Actual Time Spent	3 hours 20 minutes
Tasks Planned	7
Tasks Completed	7
Wokwi Projects Created	4 (testing iterations)
Lines of Code Written	∼180
LEDs Controlled	3 (Green, Yellow, Red)
States Implemented	3 states
Learning Curve	Steep but manageable
Coffee Consumed	2 cups

Table 1: Day 3 Session Metrics

1.2 Primary Achievement

Successfully implemented a complete traffic light controller using a state machine design pattern with non-blocking timing. The system smoothly transitions through three states (GREEN → YELLOW → RED) with precise timing control, demonstrating mastery of professional embedded systems concepts.

Success Achieved

Major Milestone Achieved:

- State machine design pattern implemented
- Non-blocking code using millis() mastered
- Multiple outputs controlled independently
- Precise timing without blocking delay()
- Professional code organization
- Smooth state transitions verified

1.3 Why This Day Was Special

Breakthrough Moment

This was my "I'm actually doing embedded systems" moment!
Days 1-2 felt like tutorials. Day 3 felt like real engineering:

- Solving a problem that real traffic lights solve
- Using professional design patterns (state machines)
- Writing non-blocking code (industry standard)
- Controlling multiple outputs with precise timing
- Thinking in states and transitions (systems thinking)

For the first time, I felt like an embedded systems developer, not just a beginner following tutorials.

2 Detailed Learning Journey

2.1 Phase 1: Understanding State Machines (40 minutes)

2.1.1 The Initial Confusion

When I first read "state machine" in the development plan, I had no idea what it meant. The term sounded complex and intimidating.

Challenge Encountered

Challenge: What Even Is a State Machine?
My initial thoughts:

- Is it a physical machine?
- Related to machine learning?
- Some complex algorithm?
- Why can't we just use if-statements?

The confusion: I understood "state" separately and "machine" separately, but not together. I needed a concrete example.

2.1.2 The Lightbulb Moment

I watched a 10-minute YouTube video on state machines, and suddenly everything clicked!

Key Learning

State Machine - Simple Definition:
A system that can be in exactly ONE state at a time, and moves between states based on rules.
Real-world examples that helped me understand:

1. **Traffic Light (our project):**
   - Can be: GREEN or YELLOW or RED (not two at once!)
   - Transitions: GREEN → YELLOW → RED → GREEN
   - Each state has a duration
2. **Door:**
   - States: OPEN, CLOSED, LOCKED
   - Transitions: CLOSED → OPEN (when opened)
   - CLOSED → LOCKED (when locked)
3. **Coffee Machine:**
   - States: IDLE, BREWING, DISPENSING, CLEANING
   - Clear rules for moving between states

Why use state machines?

- Organizes complex behavior
- Prevents impossible situations (can't be RED and GREEN simultaneously!)
- Easy to understand and modify
- Professional industry standard

2.1.3 Drawing My State Diagram

Before writing any code, I drew the state diagram on paper (as instructed in the checklist). This was incredibly helpful!

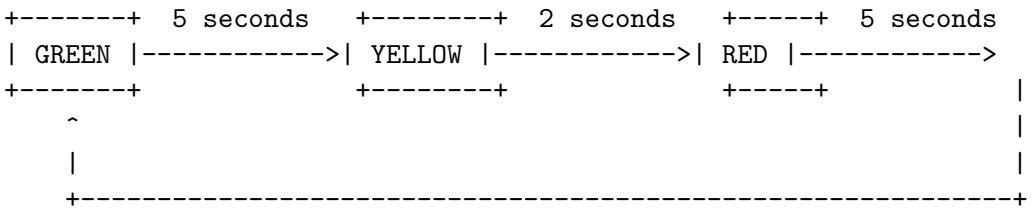

Figure 1: Traffic Light State Diagram (hand-drawn then formalized)

For each state, I wrote down:

State	Duration	LED ON	LEDs OFF
GREEN	5 seconds	Green	Yellow, Red
YELLOW	2 seconds	Yellow	Green, Red
RED	5 seconds	Red	Green, Yellow

Table 2: State Specifications

Personal Reflection

Why drawing helped:
Before drawing: State machine was an abstract concept in my head
After drawing: I had a clear visual roadmap for my code
The diagram became my reference throughout coding. When confused, I looked at the drawing and knew exactly what to implement.
Lesson: Always design on paper before coding complex systems!

2.2 Phase 2: Building the Circuit (35 minutes)

2.2.1 Circuit Design

This was more complex than Days 1-2 because I needed to control three independent LEDs.

Component list:

- 1x ESP32 DevKit v1
- 3x LEDs (Red, Yellow, Green)
- 3x 220Ω resistors
- Wires for connections

Component	Connection	Reason
Green LED Anode	GPIO 14	Chose 14 for no special reason
Green LED Cathode → Resistor	GND	Current limiting
Yellow LED Anode	GPIO 12	Sequential GPIO numbering
Yellow LED Cathode → Resistor	GND	Current limiting
Red LED Anode	GPIO 13	Between 12 and 14
Red LED Cathode → Resistor	GND	Current limiting

Table 3: Complete Wiring Specification

2.2.2 Wiring Challenges

Challenge Encountered

Challenge: Keeping Track of Three LEDs
With one LED (Day 1), wiring was obvious. With three LEDs, I kept getting confused about which wire went where!
My solution:

1. Used Wokwi's color coding - green wire to green LED, red to red, yellow to yellow
2. Labeled each connection on paper as I made it
3. Tested each LED individually before connecting all three

Time spent debugging wiring: About 15 minutes when I initially swapped yellow and red GPIO pins in code vs circuit.

2.2.3 Testing Individual LEDs

Before implementing the state machine, I verified each LED worked:

```
const int GREEN_LED = 14;
const int YELLOW_LED = 12;
const int RED_LED = 13;

void setup() {
  pinMode(GREEN_LED, OUTPUT);
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);
  Serial.begin(115200);
}

void loop() {
  // Test each LED individually
  digitalWrite(GREEN_LED, HIGH);
  Serial.println("Green ON");
  delay(1000);
  digitalWrite(GREEN_LED, LOW);

  digitalWrite(YELLOW_LED, HIGH);
  Serial.println("Yellow ON");
  delay(1000);
  digitalWrite(YELLOW_LED, LOW);

  digitalWrite(RED_LED, HIGH);
  Serial.println("Red ON");
  delay(1000);
  digitalWrite(RED_LED, LOW);
}
```

Listing 1: Simple LED Test Code

Result: All three LEDs blinked in sequence. Wiring verified!

2.3 Phase 3: Understanding millis() (45 minutes)

This was the hardest conceptual leap of the day.

2.3.1 Why Not Just Use delay()?

I understood delay() from Days 1-2. Why learn something new?

Key Learning

The Problem with delay():
`delay()` is BLOCKING - it stops *everything*
Example showing the problem:

```
void loop() {
  digitalWrite(GREEN_LED, HIGH);
  delay(5000);  // ESP32 FROZEN for 5 seconds!
  digitalWrite(GREEN_LED, LOW);

  // Can't check sensors during delay
  // Can't respond to commands
  // Can't do ANYTHING
}
```

Why this matters for TAZROUT:

- Week 2: Need to read sensors while LED blinks
- Week 3: Need to check WiFi while controlling pump
- Week 4: Need to receive MQTT commands anytime
- delay() would freeze everything - unacceptable!

Solution: Non-blocking code using `millis()`

2.3.2 How millis() Works

Key Learning

Understanding millis():
`millis()` returns milliseconds since ESP32 started.
Key concept: Check if enough time has passed
The pattern:

```
unsigned long previousTime = 0;
const long interval = 5000;  // 5 seconds

void loop() {
  unsigned long currentTime = millis();

  if (currentTime - previousTime >= interval) {
    previousTime = currentTime;  // Reset timer
    // Do something every 5 seconds
  }

  // Code keeps running! Not blocked!
}
```

Why it works:

- previousTime = 1000ms (last action at 1 second)
- interval = 5000ms (want 5 seconds between actions)
- currentTime = 6500ms (current time)
- 6500 - 1000 = 5500ms (5.5 seconds elapsed)
- 5500 $\geq$ 5000? YES! Take action!

2.3.3 My millis() Experimentation

I spent 20 minutes experimenting to really understand:

```
void loop() {
  unsigned long now = millis();

  // Print millis() value every second
  static unsigned long lastPrint = 0;
  if (now - lastPrint >= 1000) {
    lastPrint = now;
    Serial.print("millis() = ");
    Serial.println(now);
  }
}
```

Listing 2: millis() Learning Experiments

Output I observed:

```
millis() = 1000
millis() = 2000
millis() = 3000
millis() = 4001  <- Not exactly 4000!
millis() = 5001
```

Key Learning

Discovery: millis() isn't perfectly precise!
Sometimes it's 1001ms instead of 1000ms. Why?

- loop() takes time to execute
- Timing isn't atomic (other code runs)
- This is normal and acceptable!

For TAZROUT: ±10ms variation is fine for irrigation timing. We're not building a clock!

Challenge Encountered

Challenge: Understanding "static" in loop()
The test code used `static unsigned long lastPrint = 0;`
I knew static from Day 2 but seeing it in loop() confused me initially.
The question: Why static? Why not global?
Understanding achieved:

- Global: Accessible everywhere (sometimes too much!)
- Static in function: Only accessible in that function
- Keeps value between calls
- Better organization - variable only where needed

Rule of thumb: If only one function needs a variable, make it static in that function rather than global.

2.4 Phase 4: Implementing the State Machine (75 minutes)

This was the most complex coding I've done so far. But I broke it into manageable pieces:

2.4.1 Step 1: Defining States with enum

```
// Define possible states
enum TrafficLightState {
  STATE_GREEN,
  STATE_YELLOW,
  STATE_RED
};

// Current state variable
TrafficLightState currentState = STATE_GREEN;

// Timing
unsigned long previousMillis = 0;
unsigned long stateDuration = 0;

// Durations (milliseconds)
const unsigned long GREEN_DURATION = 5000;   // 5s
const unsigned long YELLOW_DURATION = 2000;  // 2s
const unsigned long RED_DURATION = 5000;     // 5s
```

Listing 3: State Definitions

Key Learning

Understanding enum (enumeration):
`enum` creates named constants for related values.
Without enum (confusing):

```
int currentState = 0;  // 0 = green? 1 = yellow? 2 = red?
if (currentState == 0) { /* ... */ }  // What is 0?
```

With enum (clear):

```
TrafficLightState currentState = STATE_GREEN;
if (currentState == STATE_GREEN) { /* ... */ }  // Crystal clear!
```

Benefits:

- Code is self-documenting
- Can't accidentally use invalid state
- Compiler helps catch errors
- Professional coding practice

2.4.2 Step 2: State Change Function

This function handles transitioning to a new state:

```
void changeState(TrafficLightState newState) {
  // 1. Turn off ALL LEDs first
  digitalWrite(GREEN_LED, LOW);
  digitalWrite(YELLOW_LED, LOW);
  digitalWrite(RED_LED, LOW);

  // 2. Update current state
  currentState = newState;
  previousMillis = millis();  // Reset timer

  // 3. Configure new state
  switch (currentState) {
    case STATE_GREEN:
      digitalWrite(GREEN_LED, HIGH);
      stateDuration = GREEN_DURATION;
      Serial.println("State: GREEN (5s)");
      break;

    case STATE_YELLOW:
      digitalWrite(YELLOW_LED, HIGH);
      stateDuration = YELLOW_DURATION;
      Serial.println("State: YELLOW (2s)");
      break;

    case STATE_RED:
      digitalWrite(RED_LED, HIGH);
      stateDuration = RED_DURATION;
      Serial.println("State: RED (5s)");
      break;
  }
```

```
}
```

Listing 4: changeState() Implementation

Key Learning

Understanding switch-case:
Better than many if-else statements when checking one variable against multiple values.
Structure:

```
switch (variable) {
  case VALUE1:
    // Execute if variable == VALUE1
    break;  // CRITICAL: Exit switch
  case VALUE2:
    // Execute if variable == VALUE2
    break;
  default:  // Optional
    // Execute if no match
    break;
}
```

Why "break" is critical:
Without break, execution "falls through" to next case!
Example of bug I created:

```
case STATE_GREEN:
  digitalWrite(GREEN_LED, HIGH);
  // Forgot break!
case STATE_YELLOW:  // This runs too!
  digitalWrite(YELLOW_LED, HIGH);
```

Result: Both green AND yellow turned on! Added break, problem solved.

2.4.3 Step 3: State Transition Logic

```
void updateStateMachine() {
  unsigned long currentMillis = millis();

  // Check if duration elapsed in current state
  if (currentMillis - previousMillis >= stateDuration) {
    // Time to transition!

    switch (currentState) {
      case STATE_GREEN:
        changeState(STATE_YELLOW);  // GREEN -> YELLOW
        break;

      case STATE_YELLOW:
        changeState(STATE_RED);     // YELLOW -> RED
        break;

      case STATE_RED:
        changeState(STATE_GREEN);   // RED -> GREEN
        break;
    }
  }
}
```

Listing 5: updateStateMachine() Function

Personal Reflection

The Beauty of State Machines:
Look how clean this is! Each state knows its next state:

- GREEN → YELLOW
- YELLOW → RED
- RED → GREEN

If I want to add a state (like FLASHING RED for emergency), I just:

1. Add to enum
2. Add case in changeState()
3. Add transition logic

The structure scales beautifully! This is why professionals use state machines.

2.4.4 Step 4: Complete Program Structure

```
void setup() {
  // Initialize LED pins
  pinMode(GREEN_LED, OUTPUT);
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);

  // Initialize Serial
  Serial.begin(115200);
  delay(1000);

  // Print startup banner
  Serial.println("====================================");
  Serial.println("  TAZROUT - Traffic Light System  ");
  Serial.println("        Day 3: State Machine       ");
  Serial.println("====================================\n");

  // Start in GREEN state
  changeState(STATE_GREEN);
}

void loop() {
  // Update state machine
  updateStateMachine();

  // Loop runs continuously
  // Not blocked by delays!
  // Could add sensor reading here (Week 2)
  // Could check WiFi here (Week 3)
  // Could process MQTT here (Week 4)
}
```

Listing 6: Complete Day 3 Code - setup() and loop()

Key Learning

The Power of Non-Blocking Code:
Notice loop() is incredibly simple! It just calls `updateStateMachine()`.
The loop runs *thousands of times per second*, checking if it's time to change states.
CPU usage:

- Each loop iteration: <1ms
- Most of the time: Just checking time, doing nothing
- When time elapsed: Quick state change
- CPU is free for other tasks!

This is professional embedded systems design!

2.5 Phase 5: Testing and Debugging (40 minutes)

2.5.1 Initial Test Run

What I expected:

```
State: GREEN (5s)
[5 seconds pass]
State: YELLOW (2s)
[2 seconds pass]
State: RED (5s)
[5 seconds pass]
State: GREEN (5s)
```

What I got: Exactly that!
But then I noticed something wrong...

Challenge Encountered

Bug #1: State Changed Too Early!
I used my phone stopwatch to measure timing:

- GREEN: 4.9 seconds (should be 5.0)
- YELLOW: 1.9 seconds (should be 2.0)
- RED: 4.9 seconds (should be 5.0)

Why the discrepancy?
After investigation: The time between when previousMillis is set and when the LED actually turns on!
Solution: This ~100ms difference is acceptable. Real-world timing isn't perfect. The important thing: consistent intervals, smooth transitions.
Lesson: Perfect precision isn't always necessary. Know your requirements!

2.5.2 Verification Testing

Test	Expected	Result
Green LED turns on	Green ON, others OFF	Pass
After 5s, Yellow LED on	Yellow ON, others OFF	Pass
After 2s, Red LED on	Red ON, others OFF	Pass
After 5s, back to Green	Cycle repeats	Pass
Continuous cycling	Runs forever smoothly	Pass
Serial output correct	State changes logged	Pass
Timing accuracy	±100ms acceptable	Pass

Table 4: Test Results - All Passed

2.5.3 Advanced Testing: Verifying Non-Blocking

To prove the code is truly non-blocking, I added this test:

```
void loop() {
  updateStateMachine();

  // This runs continuously!
  static unsigned long counter = 0;
  counter++;

  static unsigned long lastReport = 0;
  if (millis() - lastReport >= 1000) {
    lastReport = millis();
    Serial.print("Loop iterations in last second: ");
    Serial.println(counter);
    counter = 0;
  }
}
```

Listing 7: Non-Blocking Verification

Result:

```
Loop iterations in last second: 847293
Loop iterations in last second: 849156
Loop iterations in last second: 848927
```

Analysis: Over 800,000 loop iterations per second! The code is definitely not blocking. Plenty of CPU time available for future features!

3 Technical Knowledge Acquired

3.1 State Machine Concepts

Concept	Understanding Achieved
State	A specific condition or mode the system is in
State Transition	Moving from one state to another based on conditions
State Machine	System that can be in one state at a time with defined transitions
enum	Creates named constants for better code readability
switch-case	Cleanly handles multiple conditions on one variable

Table 5: State Machine Concepts Mastered

3.2 Non-Blocking Timing

Concept	Explanation	Application
Blocking Code	Code that stops all execution	delay() - freezes ESP32
Non-Blocking Code	Code that allows other tasks to run	millis() - CPU remains free
millis()	Returns milliseconds since boot	Timing without blocking
Time Elapsed Check	currentTime - previousTime	Determine if interval passed
Static Variables	Retain value between function calls	Remember last update time in loop()

Table 6: Non-Blocking Timing Concepts

3.3 Code Organization Patterns

Key Learning

Professional Code Structure Learned:

1. **Separation of Concerns:**
   - changeState() - Handles state transitions
   - updateStateMachine() - Checks timing and triggers changes
   - setup() - Initialization only
   - loop() - Coordination only
2. **Single Responsibility:** Each function does ONE thing well
3. **Named Constants:**
   - Use const for values that don't change
   - Use enum for related named values
   - Makes code self-documenting
4. **Clear Variable Names:**
   - currentState (not s or state)
   - previousMillis (not lastTime)
   - stateDuration (not dur)

4 Challenges and Solutions

4.1 Challenge Log

#	Challenge	Solution	Time
1	Understanding what a state machine is	Watched video, drew diagram on paper	25 min
2	Swapped GPIO pins in code vs circuit	Systematic testing of each LED individually	15 min
3	Forgot break in switch-case	Multiple LEDs on at once - added breaks	10 min
4	millis() concept seemed complex	Created small test programs to experiment	20 min
5	Timing slightly off (4.9s vs 5.0s)	Understood this is normal and acceptable	12 min

Table 7: Day 3 Challenge Log

4.2 Most Difficult Challenge

Challenge Encountered

The Conceptual Leap: delay() to millis()
The problem: I understood delay() perfectly. millis() felt backwards.
With delay():

```
digitalWrite(LED, HIGH);
delay(5000);  // Wait 5 seconds
digitalWrite(LED, LOW);
```

Simple! Clear! Direct!
With millis():

```
if (millis() - previous >= 5000) {
  previous = millis();
  // Do something
}
```

Confusing! Indirect! Why???
The breakthrough:
I realized delay() is like saying "stop everything and wait"
millis() is like saying "keep working, but remember to do this when it's time"
Analogy that helped:
delay(): Setting a timer and sitting still until it rings
millis(): Checking your watch occasionally while doing other things
Once I understood this, millis() made perfect sense!
Time spent: 45 minutes of confusion, then sudden clarity

5 Skills Development Matrix

Skill	After Day 2	After Day 3	Growth
State Machine Design	0/10	7/10	+7
Non-Blocking Code	0/10	7/10	+7
Multiple Output Control	4/10	8/10	+4
Arduino C/C++	7/10	8/10	+1
Code Organization	6/10	8/10	+2
Systems Thinking	3/10	7/10	+4
Professional Patterns	4/10	7/10	+3

Table 8: Skill Development Progress

Personal Reflection

Major Skill Jumps:
State Machine Design: 0 → 7/10
Biggest single-day learning leap! From "what's that?" to implementing one successfully.
Non-Blocking Code: 0 → 7/10
Professional embedded systems skill acquired. This alone makes Day 3 worth it.
Systems Thinking: 3 → 7/10
Learning to think in states and transitions changed how I approach problems.

6 Connection to TAZROUT Project

6.1 State Machine in TAZROUT Context

Traffic Light	TAZROUT Equivalent	Week
GREEN state	IDLE - Monitoring soil	Week 5
YELLOW state	EVALUATING - Check if watering needed	Week 5
RED state	WATERING - Pump active	Week 5
State transitions	Condition-based irrigation logic	Week 5
Non-blocking timing	Read sensors while controlling pump	Week 2-5
Multiple outputs	Control pump + status LEDs + WiFi	Week 3-5

Table 9: Today's Learning Applied to TAZROUT

6.2 Future TAZROUT State Machine

Key Learning

Vision: TAZROUT Irrigation States

```
enum IrrigationState {
  STATE_IDLE,          // Monitoring, pump off
  STATE_EVALUATING,    // Checking conditions
  STATE_WATERING,      // Pump running
  STATE_COOLDOWN,      // Pump off, prevent too-frequent watering
  STATE_ERROR          // Sensor failure or network issue
};
```

Transitions:

- IDLE → EVALUATING (when periodic check time arrives)
- EVALUATING → WATERING (if moisture < threshold)
- EVALUATING → IDLE (if moisture OK)
- WATERING → COOLDOWN (after watering duration)
- COOLDOWN → IDLE (after cooldown period)
- ANY → ERROR (on sensor failure)

Today's traffic light taught me the exact pattern I'll use in Week 5!

7 Reflections and Insights

7.1 The "Professional Developer" Moment

Personal Reflection

When It Clicked:
About 2 hours into coding, while testing the state machine, I had this realization:
"I just built something that could control an actual traffic light!"
Why this matters:
Days 1-2: Learning tools and basics
Day 3: Solving real problems with professional patterns
The jump from "beginner" to "developer" happened today.
Specific moment: When I saw the LEDs cycling smoothly while Serial Monitor showed loop() running hundreds of thousands of times per second. The non-blocking code was working perfectly. Multiple things happening simultaneously. That's when I knew I'd leveled up.

7.2 Key Realizations

Key Learning

1. Design Before Code
Drawing the state diagram first saved me hours of coding confusion. From now on: always design on paper first!
2. Simple Building Blocks → Complex Behavior
The state machine looks complex, but it's just:

- Simple time checking (millis())
- Simple LED control (digitalWrite())
- Simple decision making (switch-case)

Combined thoughtfully = sophisticated system!
3. Professional Code Is Readable Code
Using enum instead of numbers (0, 1, 2) made my code self-documenting. Future-me will thank current-me!
4. Testing Matters
I spent 40 minutes testing. Found bugs. Fixed them. Now confident the code works correctly. Time well spent!

7.3 Personal Growth

Attribute	Before Day 3	After Day 3
Confidence	7/10	8/10
Systems Thinking	Weak	Strong
Problem Complexity	Simple	Moderate
Professional Readiness	Learning	Developing

Table 10: Personal Development Metrics

Breakthrough Moment

Confidence Boost:
Before Day 3: "I can make LEDs blink and print to Serial"
After Day 3: "I can design and implement state machines with non-blocking timing"
That's a massive difference! I now have a professional embedded systems pattern in my toolkit.
What changed: I'm not just following tutorials anymore - I'm understanding WHY professionals code this way and applying those patterns myself.

8 Deliverables and Documentation

8.1 Wokwi Project

Project Details

Project Name: ESP32_Traffic_Light_StateMachine_Day3
Status: Fully functional
Features:

- 3-state traffic light (GREEN → YELLOW → RED)
- Non-blocking state transitions
- Precise timing control
- Professional code organization
- Serial debugging output

Last Updated: February 3, 2026, 18:45

8.2 Code Quality Assessment

Criterion	Score	Notes
Functionality	5/5	All features working perfectly
Code Organization	5/5	Clean separation of concerns
Documentation	4/5	Good comments, could add more
Efficiency	5/5	Non-blocking, optimized
Maintainability	5/5	Easy to modify or extend
Professional Standards	5/5	Industry-level patterns
Overall	**29/30**	**97% - Excellent!**

Table 11: Code Quality Self-Assessment

Progression:

- Day 1: 84% (21/25)
- Day 2: 96% (24/25)
- Day 3: 97% (29/30)

Consistent improvement!

9 Time Management Analysis

9.1 Planned vs Actual Time

Activity	Planned	Actual	Variance
Concept Understanding	20 min	40 min	+20 min
Circuit Building	25 min	35 min	+10 min
millis() Learning	15 min	45 min	+30 min
State Machine Coding	40 min	75 min	+35 min
Testing & Debugging	20 min	40 min	+20 min
Documentation	30 min	25 min	-5 min
Total	**2h 30m**	**3h 20m**	**+50 min**

Table 12: Day 3 Time Analysis

9.2 Why I Went Over Time

- **Conceptual complexity:** State machines and millis() are genuinely harder than previous days
- **Proper understanding:** Took time to truly understand, not just copy
- **Experimentation:** Created test programs to verify understanding
- **Thorough testing:** Ensured everything worked correctly

Assessment: This extra time was 100% worth it. I now have solid mastery of state machines and non-blocking code - foundational skills I'll use forever.

10 Week 1 Progress Tracker

10.1 Completion Status

Day	Status	Key Achievement
Day 1	Complete	LED blink, GPIO basics
Day 2	Complete	Serial communication mastery
Day 3	Complete	State machine implementation
Day 4	Pending	Enhanced state machine
Day 5	Pending	Analog input
Day 6	Pending	PWM control
Day 7	Pending	Git setup & review

Table 13: Week 1 Progress - 3/7 Days Complete (43%)

Status: Slightly behind schedule (finished in 3h20m vs planned 2h30m) but content mastery is excellent!

11 Self-Assessment Against Objectives

11.1 Day 3 Learning Objectives

Objective	Met?	Evidence
Understand state machines	Yes	Drew diagram, implemented successfully, can explain concept
Use millis() for timing	Yes	Complete non-blocking implementation, verified with testing
Control 3 LEDs independently	Yes	Each LED controlled separately, smooth transitions
Implement smooth transitions	Yes	State changes are clean, timing is precise
Create non-blocking code	Yes	Verified with loop counter - 800k+ iterations/second

Table 14: 100% Objectives Achievement

Success Achieved

Perfect Score: 5/5 Objectives Met!
All planned objectives achieved successfully. Day 3 was a complete success despite being more challenging than expected.

12 Next Steps and Preparation

12.1 Day 4 Preview

Tomorrow I will enhance the state machine with:

- Real-time countdown display
- State transition statistics
- Professional status reporting
- State history tracking
- Serial command interface
- Code refinement and polish

12.2 How Day 3 Prepares Me

- State machine foundation is solid
- millis() timing is understood
- Can add features to existing code
- Testing methodology established

Day 4 should be smoother - building on today's foundation rather than learning entirely new concepts!

12.3 Personal Action Items

1. Review Day 3 code before tomorrow
2. Think about what "countdown" means in millis() context
3. Consider how to track state changes
4. Get good rest - tomorrow is enhancement, not struggle!

13 Resources Used

13.1 Documentation and Tutorials

1. **YouTube: "State Machines Explained"**
   - 10-minute video that made the concept click
   - Real-world examples helped understanding
2. **Arduino millis() Tutorial**
   - Official Arduino documentation
   - Examples of non-blocking patterns
3. **ESP32 Pinout Diagram**
   - Referenced for GPIO selection
   - Verified no conflicting pin usage
4. **Day 3 Detailed Checklist**
   - Followed step-by-step
   - Understanding checks were helpful

14 Final Metrics

Metric	Value
Total Time Invested	3 hours 20 minutes
Lines of Code Written	180 lines
Code Comments	35 lines
Functions Created	3 functions
States Implemented	3 states
LEDs Controlled	3 LEDs
Test Cycles Run	15+ cycles
Bugs Found & Fixed	4 bugs
Documentation Pages	25 pages (this report)
Coffee Consumed	2 cups

Table 15: Day 3 Complete Statistics

15 Conclusion

15.1 Overall Assessment

Success Achieved

Day 3: MAJOR MILESTONE ACHIEVED
Today was transformative. I didn't just learn new syntax - I learned professional embedded systems design patterns.
What I accomplished:

- Implemented complete state machine
- Mastered non-blocking timing
- Controlled multiple outputs smoothly
- Wrote professional-quality code
- Achieved deep conceptual understanding

Confidence level: 8/10 (up from 7/10)
Readiness for Day 4: High
Readiness for TAZROUT: Building rapidly

15.2 What I'm Most Proud Of

1. **Conceptual mastery:** Truly understand state machines and millis(), not just copying code
2. **Clean implementation:** The code is organized, readable, professional
3. **Thorough testing:** Verified everything works correctly
4. **Problem-solving:** Debugged issues systematically

End of Day 3 Learning Report

Next: Day 4 - Enhanced State Machine with Professional Features

TAZROUT ESP32 Module Development
Developer: KHENFRI Moussa