GENERAL

atmel microcontroller and c programming simon led game

Published: September 19, 2026 | Index ID: #-

Unlocking the Classic Simon Game with an Atmel Microcontroller and C Programming

For hobbyists and seasoned embedded engineers alike, recreating the iconic **Simon LED game** is a perfect blend of hardware tinkering and software logic. The original game, first released in the early 80s, challenges players to remember an ever growing sequence of colored lights and button presses. By harnessing an **Atmel microcontroller**—such as the popular ATmega328P or ATmega2560—and writing firmware in C, you can build a faithful replica that runs on a custom PCB or an Arduino board. This article walks you through every step: from selecting components and wiring the LEDs to structuring your C code, handling timing, and adding fun features like difficulty levels and sound output.

Why an Atmel Microcontroller is Ideal for a Simon LED Game

Atmel's AVR family of microcontrollers (now part of Microchip) offers a balanced mix of low power consumption, ample GPIO pins, and a robust C compiler ecosystem. Key reasons to choose an Atmel device for this project include:

- Rich Pin Count: Devices such as the ATmega328P provide 23 I/O pins, enough to drive four LEDs, four push buttons, a speaker, and optional peripherals.
- Integrated Timers and PWM: Precise LED dimming, button debouncing, and sound generation become straightforward.
- Low Cost & Availability: These chips are inexpensive, widely stocked, and supported by the Arduino ecosystem.
- Strong Community: A wealth of libraries, tutorials, and forums means you'll rarely hit a wall.

Gathering the Parts: A Component Checklist

Below is a practical list of components you'll need to build a complete Simon LED game. Feel free to adjust quantities or add extras if you want to experiment with more colors or a larger display.

- Atmel ATmega328P (or ATmega2560 for a larger board)
- Four 220 /:' 7W'&VçB Æ–Ö—F–ær &W6—7F÷'0
- Four LEDs (red, green, blue, yellow)
- Four tactile push buttons (or a 4 way switch)
- One 8 bit PWM speaker or buzzer
- One 16 MHz crystal oscillator with two 22 /pF capacitors
- 10 /k:' VÆÂ W &W6—7F÷'2 `or the buttons
- One 0.1 /µF ceramic capacitor (for the crystal)
- One 10 /µF electrolytic capacitor (for power smoothing)
- Power supply (5 /V regulator or USB via Arduino)
- Perfboard or custom PCB, wires, and a prototyping breadboard if preferred
- Optional: an LCD or OLED display for scores and instructions

Hardware Setup: Wiring the LEDs, Buttons, and Speaker

LED Connections

Each LED is connected from a digital pin to ground through a 220 /:’ &W6—7F÷ . For example, on an ATmega328P:

- Red LED !’ Pin D2
- Green LED !’ Pin D3
- Blue LED !’ Pin D4
- Yellow LED !’ Pin D5

When the microcontroller sets a pin HIGH, the LED lights up. The resistor limits current to a safe 10–20 mA, protecting both the LED and the microcontroller.

Button Connections

Each button connects from a digital pin to ground. The pin’s internal pull up resistor is activated in software, simplifying wiring:

- Button 1 !’ Pin D6
- Button 2 !’ Pin D7
- Button 3 !’ Pin D8
- Button 4 !’ Pin D9

When a button is pressed, the pin reads LOW; otherwise, it reads HIGH.

Speaker Wiring

For simple sound output, connect the speaker to a PWM pin (e.g., Pin D10). Tie the speaker’s negative lead to ground and the positive lead to the PWM output through a 100 /:’ &W6—7F÷" Fò Æ–Ö—B 7W'&VçBà

Power and Oscillator

Use a 16 /MHz crystal with two 22 /pF capacitors for accurate timing. Supply the ATmega328P with 5 /V from a USB or a 5 /V regulator. Add a 10 /µF capacitor across the VCC and GND pins to filter noise.

Software Overview: Structuring Your C Firmware

The firmware can be organized into several logical modules, each handling a specific aspect of the game. Below is a high level view of the modules and their responsibilities.

1. Hardware Abstraction Layer (HAL): Functions to initialize GPIO, timers, and PWM.
2. Input Handling: Debounce logic for buttons and reading their states.
3. Output Control: Functions to turn LEDs on/off and produce tones.
4. Game Logic: Sequence generation, user input validation, and difficulty scaling.
5. Utility Functions: Random number generation, delay routines, and error handling.

Initializing the Microcontroller in C

Below is a concise snippet that sets up the ATmega328P for the Simon game. It configures the I/O pins, enables pull ups, and prepares the PWM channel for sound.

```
/* Include AVR libraries */
#include
#include
```

```
#include
#include

/* Define pin macros for readability */
#define LED_RED    PB0
#define LED_GREEN  PB1
#define LED_BLUE   PB2
#define LED_YELLOW PB3
#define BTN_RED    PD2
#define BTN_GREEN  PD3
#define BTN_BLUE   PD4
#define BTN_YELLOW PD5
#define SPEAKER    PB4

/* Function prototypes */
void init_ports(void);
void play_tone(uint16_t freq, uint16_t duration);
void flash_led(uint8_t led, uint16_t duration);

/* Main function */
int main(void)
{
    init_ports();
    /* Main loop */
    while (1)
    {
        /* Game loop will go here */
    }
    return 0;
}

/* Initialize ports and PWM */
void init_ports(void)
{
    /* Set LED pins as outputs */
    DDRB |= (1
```

With this foundation, you can now build the game logic on top of the HAL.

Generating the Simon Sequence

The core of the game is a random sequence of colors that grows each round. The sequence can be stored in an array of bytes, where each byte represents one LED/button combination. For example, 0 = red, 1 = green, 2 = blue, 3 /= /yellow.

```
#define MAX_SEQUENCE 32
```

```
uint8_t sequence[MAX_SEQUENCE];
uint8_t sequence_length = 0;

/* Add a new random step to the sequence */
void add_random_step(void)
{
    sequence[sequence_length++] = rand() & 0x03; /* 0–3 */
}
```

By limiting MAX_SEQUENCE to 32, you avoid overflow while still allowing a challenging game.

Displaying the Sequence to the Player

Once the sequence is built, you must flash the LEDs in order. A simple function can handle this:

```
void play_sequence(void)
{
    for (uint8_t i = 0; i
```

Adjust the duration and pause to tweak the game's pacing. Adding a subtle sound cue for each flash enhances the user experience.

Reading Player Input with Debouncing

Mechanical buttons produce noisy signals that can cause false triggers. A simple software debounce routine checks the button state multiple times with a short delay. Here’s a minimal example that returns the pressed color or 0xFF if no button is pressed.

```
uint8_t read_button(void)
{
    static uint8_t last_state = 0xFF;
    static uint8_t counter = 0;

    uint8_t state = (PIND & ((1 = 5) /* 5 * 10ms = 50ms debounce */
        {
            counter = 0;
            if (state & (1
```

Integrate this routine into the main game loop to capture the user's sequence.

Validating the Player’s Sequence

After the player has pressed a series of buttons, compare their inputs to the stored sequence. If a mismatch occurs, trigger a “game over” routine; otherwise, continue to the next round.

```
uint8_t validate_input(void)
{
    for (uint8_t i = 0; i
```

Adding a short delay after each button press improves user readability.

Scoring, Difficulty, and Game Over

To make the game engaging, track the current round number and display it on an optional LCD. You can also increase difficulty by shortening the LED flash duration or speeding up the input window.

- Score Tracking: Increment a uint8_t score each time the player successfully reproduces the sequence.
- Difficulty Scaling: Reduce duration by 10 /ms every 5 rounds.
- Game Over Animation: Flash all LEDs in a pattern and play a buzzer tone.

Adding Sound: Using PWM for Tones

Sound enhances the tactile feedback of the Simon game. By configuring Timer1 in CTC mode and toggling the