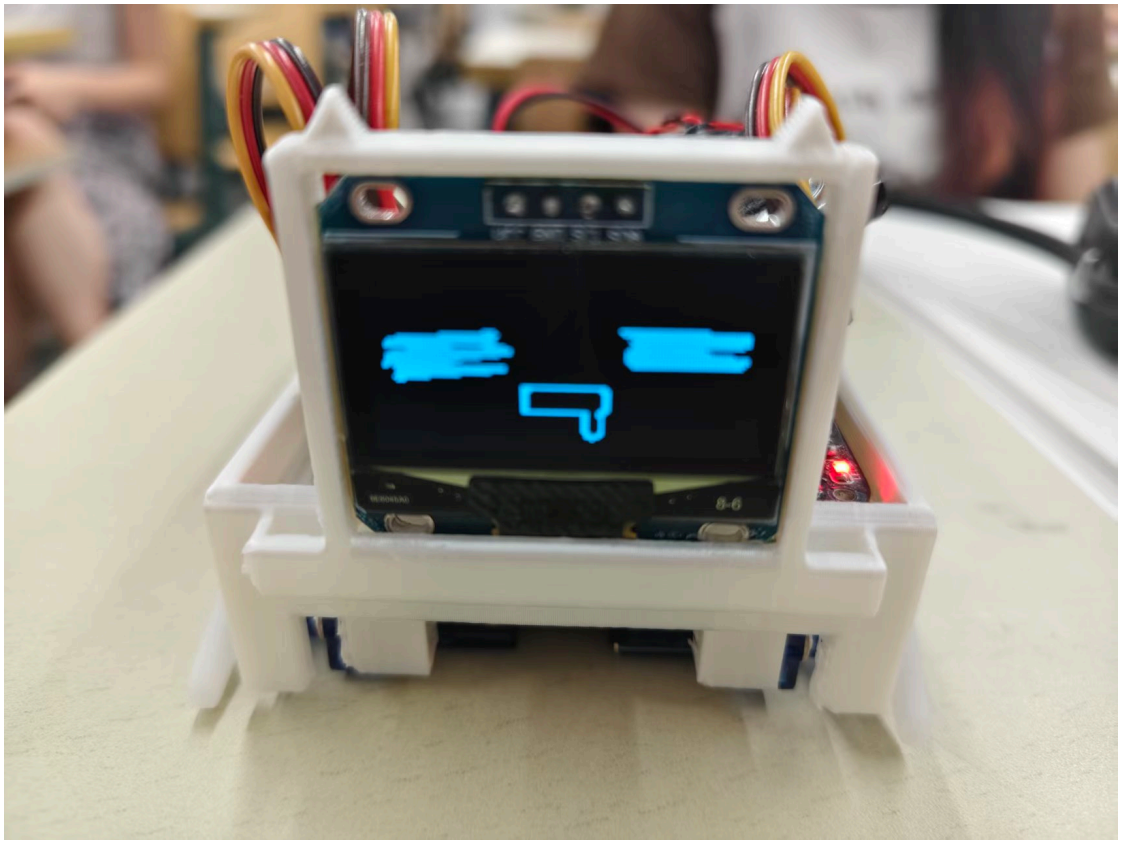


STM32 Voice-Controlled Electronic Pet Dog

Design & Implementation

June 2026

Contents

1. Project Overview

1.1. Project Basic Information

Project Name: STM32 Voice-Controlled Electronic Pet Dog

Platform: Built on the STM32F103C8T6 microcontroller platform. This project integrates mechanical design, PCB fabrication, embedded C programming, servo control, UART communication, and voice recognition technology into a complete interactive system.

Project Positioning: A voice-controlled bionic electronic pet dog designed as a portable, interactive embedded system. The robot can receive voice commands through a CT-18 voice recognition module and execute corresponding movements using five servo motors. It features an OLED status display for real-time feedback.

Core Platform: STM32F103C8T6 (Cortex-M3) development board, also known as the ”Blue Pill”, operating at 72 MHz with 64 KB Flash and 20 KB SRAM.

1.2. Project Objectives

The primary objectives of this project are:

1. **Mechanical Integration:** Design and 3D-print a compact quadruped body that accommodates five servo motors, the main control board, and peripheral modules.
2. **Hardware Interfacing:** Solder a custom PCB expansion board to route signals between the STM32 and all peripheral modules (voice recognition, OLED, servos, buzzer, LEDs).
3. **Embedded Firmware:** Develop firmware in C using STM32CubeMX and Keil MDK, implementing PWM servo control, UART command parsing, I2C display driving, and a multi-mode control state machine.
4. **Voice Interaction:** Integrate the CT-18 offline voice recognition module to support five command words (“Forward”, “Backward”, “Left”, “Right”, “Stop”).
5. **Safety:** Physical emergency-stop button immediately halts all servo motion and returns the dog to neutral stance.

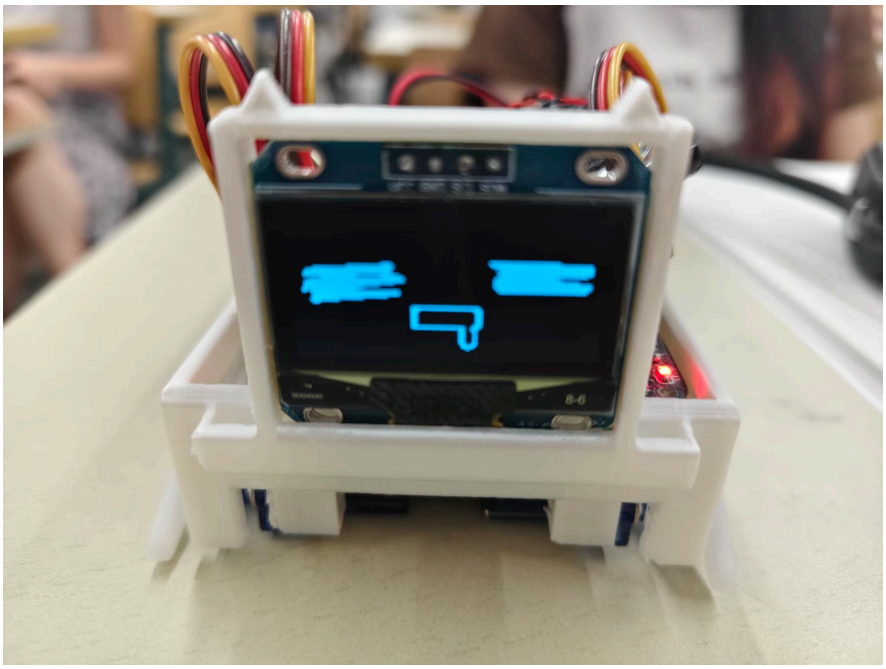


Figure 1: Assembled robot dog — front three-quarter view

2. Required Materials & Tools

This section lists all hardware components, software tools, and accounts required to reproduce the project.

2.1. Hardware Components

Table 1: Complete hardware bill of materials

Component	Description	Qty.
STM32F103C8T6 Dev Board	"Blue Pill" board, Cortex-M3 @ 72 MHz	1
CT-18 Voice Recognition Module	Offline voice recognition, UART interface, 5 custom commands	1
180° Servo Motors	SG90 micro servos, 4.8–6 V operating voltage	5
OLED Display Module	0.96" SSD1306, 128×64, I2C interface	1
PCB Expansion Board	Custom-designed, single-layer, hand-soldered	1
3D Printed Dog Body	PLA filament, 0.2 mm layer height, snap-fit assembly	1 set
3.7 V Lithium Battery	Li-Po, 2000 mAh, with JST 1.25 mm connector	1
Charging Module	TP4056-based, micro-USB input, charging indicator LED	1
Boost Converter	5 V DC-DC step-up module for servo power rail	1
Speaker/Buzzer	Passive buzzer, 5 V, PWM-driven	1
Push Button	Tactile switch, 6×6 mm, for emergency stop	1
LEDs	5 mm, red/white, for eyes and status indication	10
10 μF Capacitors	Electrolytic, power decoupling	10
0.1 μF Capacitors	Ceramic, signal filtering	5
10 kΩ Resistors	Pull-up/pull-down, current limiting	5
2 kΩ Resistors	LED current limiting	10
Dupont Wires	Male-to-female, various lengths	Several
Breadboard Jump Wires	For prototyping	Several

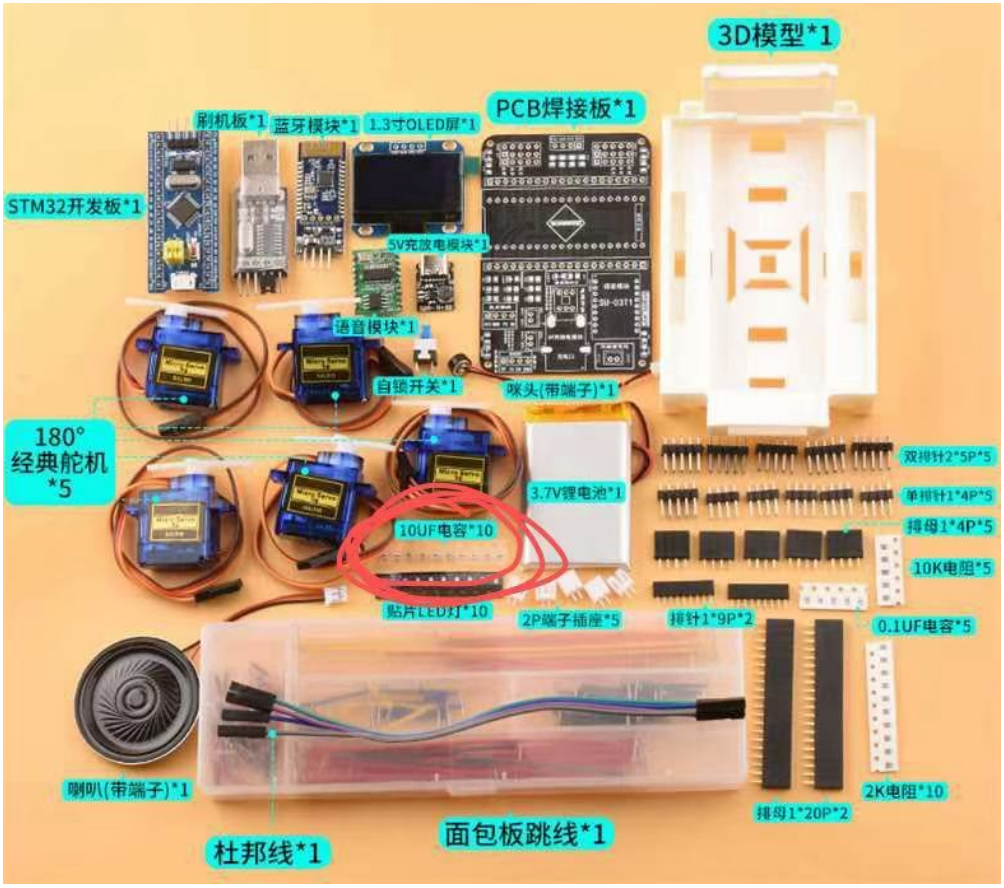


Figure 2: Hardware components laid out before assembly

2.2. Software & Tools

Table 2: Software environment and tools

Tool	Purpose
STM32CubeMX	Pin configuration, clock tree setup, peripheral initialization, code generation
Keil MDK (uVision 5)	C code editing, compilation, debugging, and firmware download
ST-Link Utility	Alternative firmware flashing tool
ST-Link V2 Programmer	Hardware debugger/programmer (SWD interface)
C Programming Language	All firmware development
3D Slicing Software	Preparing STL files for 3D printing (Cura / PrusaSlicer)

3. Step-by-Step Construction Guide

The construction of the robot dog was carried out in six sequential stages, each building upon the previous one. The following subsections detail each stage with corresponding image placeholders.

3.1. Step 1: Designing the Robot Structure

The first stage involved designing and fabricating the mechanical structure of the robot dog.

Mechanical Design

Five servo motors were positioned within the frame to control four legs and one neck (head rotation). Each leg uses one servo for forward/backward swing motion; the fifth servo rotates the head for expressive behavior. The design follows a four-bar linkage simplified to direct-drive joints.

Design Considerations

- **Lightweight structure:** The total assembled weight (excluding battery) was kept under 250 g to allow the SG90 servos to operate within their torque rating of 1.8 kg·cm.
- **Stable center of gravity:** The battery was placed at the bottom-rear to counterbalance the head servo. The STM32 board sits at the geometric center of the chassis.
- **Easy maintenance:** The body uses snap-fit connections rather than permanent adhesive, allowing quick access to internal electronics.
- **Sufficient clearance:** Internal cavity dimensions of 80 mm $\times$ 60 mm $\times$ 40 mm provide room for the PCB stack, battery, and wiring harness.

Post-Processing

After printing, support material was removed and mating surfaces were lightly sanded for smooth snap-fit assembly. The servo mounting holes were reamed to ensure precise alignment.

3.2. Step 2: Installing the Electronic Components

After completing the mechanical structure, all electronic components were installed and interconnected.

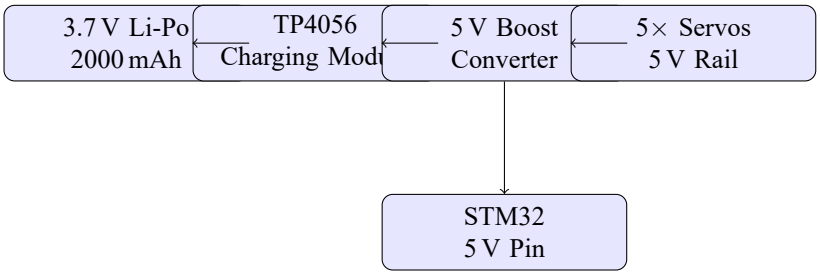
Component Placement

- **STM32 Board:** Mounted at chassis center using M2 nylon standoffs to prevent short circuits.
- **CT-18 Voice Module:** Placed at the front of the chassis with the microphone facing outward through a grille in the 3D-printed shell.
- **OLED Display:** Mounted on the dog's "back", visible from above, connected via I2C (4 wires: VCC, GND, SCL, SDA).
- **Power Supply:** TP4056 charging module $\rightarrow$ 5 V boost converter $\rightarrow$ common 5 V rail for servos and STM32 (via its 5 V input pin).
- **Servo Motors:** Five SG90s plugged into dedicated 3-pin headers on the custom PCB.

Cable Management

Wiring was routed through internal channels in the 3D-printed frame. Cable ties were used at junction points to prevent wires from interfering with leg movement. Signal wires (PWM, UART, I2C) were kept physically separated from the servo power rail to minimize electrical noise.

Power Distribution



Warning: Never power servo motors directly from the STM32's 3.3 V pin — they require 5 V and can draw peak currents exceeding 500 mA per servo. Always use a separate 5 V rail with common ground.

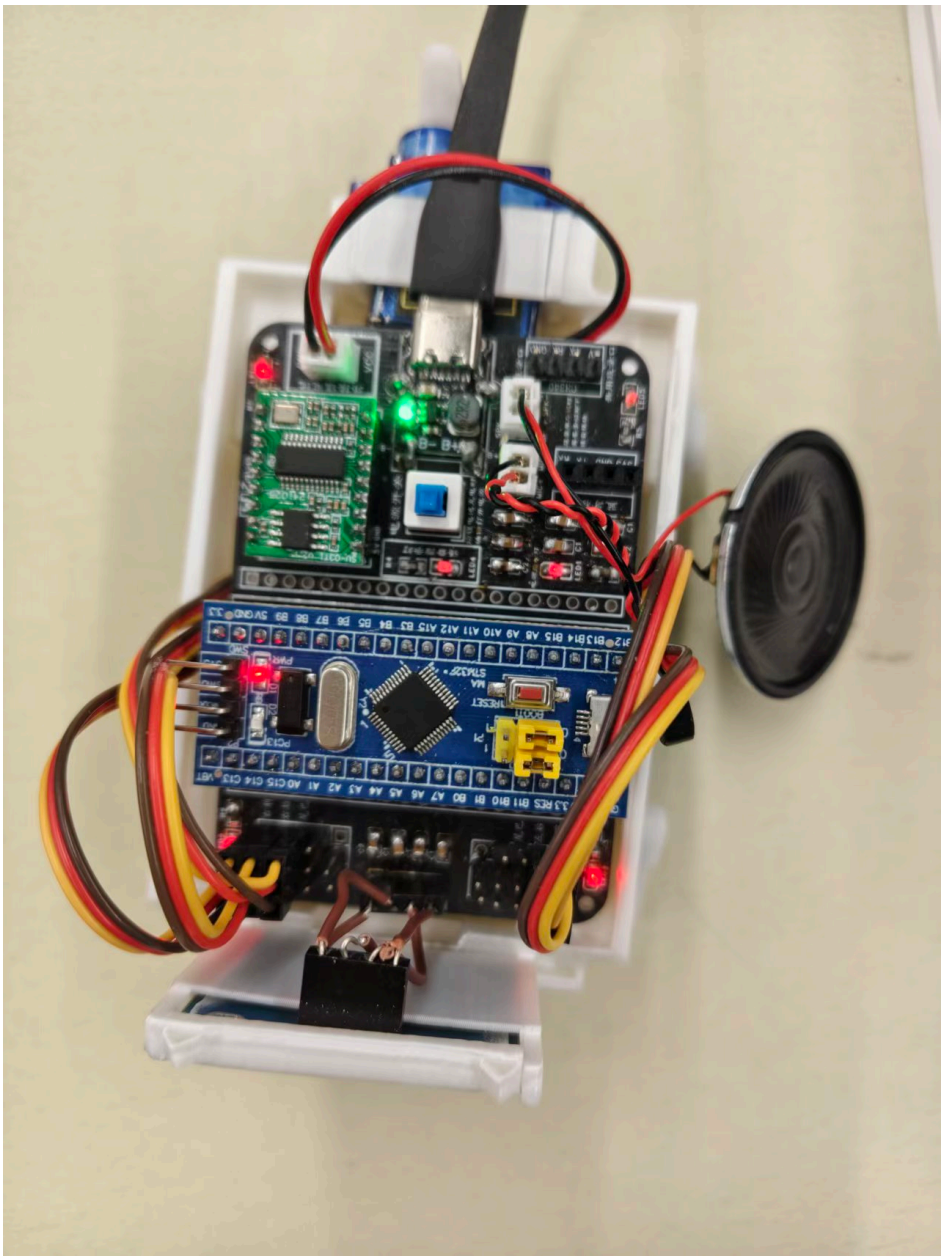


Figure 3: Internal wiring — PCB, battery, servo connectors visible with top shell removed

3.3. Step 3: Software Environment & Peripheral Configuration

This step covers the complete software toolchain setup and STM32 peripheral configuration.

Toolchain Installation

1. Download and install **STM32CubeMX** (v6.x or later) from STMicroelectronics.
2. Download and install **Keil MDK-ARM** (Community edition suffices; no license needed for STM32F103 with code size < 32 KB).
3. Install **ST-Link V2 drivers** for Windows (automatically detected on most systems).
4. Install the **STM32F1 HAL library** package via the CubeMX embedded software manager.

STM32CubeMX Pin Configuration

The following peripherals were configured in the CubeMX `.ioc` project file:

Table 3: STM32F103C8T6 pin assignment and peripheral mapping

Pin	Function	Peripheral	Mode	Purpose
PA9	TX	USART1	AF Push-Pull	CT-18 voice module (9600 baud)
PA10	RX	USART1	Input Floating	CT-18 voice module (9600 baud)
PB6	SCL	I2C1	AF Open-Drain	OLED SSD1306
PB7	SDA	I2C1	AF Open-Drain	OLED SSD1306
PA6	CH1	TIM3	AF Push-Pull	Servo 1 PWM (50 Hz)
PA7	CH2	TIM3	AF Push-Pull	Servo 2 PWM (50 Hz)
PB0	CH3	TIM3	AF Push-Pull	Servo 3 PWM (50 Hz)
PB1	CH4	TIM3	AF Push-Pull	Servo 4 PWM (50 Hz)
PB8	CH1	TIM4	AF Push-Pull	Servo 5 (neck) PWM (50 Hz)
PA0	–	GPIO	Output Push-Pull	Buzzer
PA1	–	GPIO	Input Pull-up	Emergency stop button
PC13	–	GPIO	Output Push-Pull	On-board LED (heartbeat)

Clock Configuration

- HSE (High-Speed External): 8 MHz crystal $\rightarrow$ PLL $\times 9$ $\rightarrow$ SYSCLK = 72 MHz
- APB1 Timer Clocks: 72 MHz (TIM2/3/4 — note: APB1 prescaler $\neq 1$ doubles the timer clock)
- APB2: 72 MHz (USART1)
- APB1: 36 MHz (I2C1)

PWM Timer Configuration for Servos

Servo motors require a 50 Hz PWM signal (period = 20 ms). The pulse width determines the angular position:

- 0.5 ms pulse $\rightarrow$ 0° position
- 1.5 ms pulse $\rightarrow$ 90° position (center)

- 2.5 ms pulse → 180° position

Timer configuration for TIM3 (servos 1–4):

- Prescaler = 72 – 1 = 71 → timer clock = 1 MHz (1 μs per tick)
- Auto-reload (ARR) = 20000 – 1 = 19999 → period = 20 ms (50 Hz)
- Compare values: 500 → 0°, 1500 → 90°, 2500 → 180°

Code Generation & Project Structure

After configuring all peripherals, the code was generated via CubeMX with the following settings:

- Toolchain / IDE: MDK-ARM V5
- Stack / Heap: Stack = 0x400, Heap = 0x200
- HAL library: included as source files (not static library)
- Generate peripheral initialization as a pair of `.c/.h` files per peripheral: disabled (all in `main.c` for simplicity)

The generated project was opened in Keil MDK, and the following custom source files were added to the project tree:

Table 4: Custom source files added to the Keil project

File	Responsibility
`oled_i2c.c / .h`	SSD1306 OLED driver: initialization, framebuffer, character rendering, display update over I2C1
`servo_pwm.c / .h`	Servo angle setting via TIM3/TIM4 CCR register writes; gait sequence definitions
`ct18_parser.c / .h`	CT-18 UART command parsing: receive byte, map to command enum, debounce
`motion_ctrl.c / .h`	High-level motion state machine: command dispatch, gait generation, servo sequencing
`main.c`	System initialization, main loop, mode switching

3.4. Step 4: Firmware Implementation — Voice Commands & Servo Control

This section presents the core firmware that enables voice-controlled movement. The STM32 listens to the CT-18 voice module over USART1. When a command word is recognized, the CT-18 sends a specific byte; the firmware maps that byte to a gait sequence executed by the five servo motors.

CT-18 Voice Module Communication Protocol

The CT-18 module is pre-trained with five custom command words. Upon recognizing a command, it transmits a single-byte identifier over UART at 9600 baud, 8N1:

Table 5: CT-18 command byte mapping

Voice Command	Description	Byte
“Forward”	Move forward	0x11
“Backward”	Move backward	0x12
“Left”	Turn left	0x13
“Right”	Turn right	0x14
“Stop”	Halt all motion	0x15

Main Program Structure

```c
/**
 * @file main.c
 * @brief STM32 Voice-Controlled Electronic Pet Dog
 * @date 2026-06-15
 */

#include "main.h"
#include "oled_i2c.h"
#include "servo_pwm.h"
#include "ct18_parser.h"
#include "motion_ctrl.h"

/* Global handles (generated by CubeMX) */
extern UART_HandleTypeDef huart1; /* CT-18 voice module */
extern I2C_HandleTypeDef hi2c1; /* OLED SSD1306 */
extern TIM_HandleTypeDef htim3; /* Servo PWM 1-4 */
extern TIM_HandleTypeDef htim4; /* Servo PWM 5 (neck) */

/* System state */
typedef enum {
    MODE_VOICE = 0  /* Default: voice control via CT-18 */
} SystemMode;

SystemMode g_mode = MODE_VOICE;
CmdType    g_cmd    = CMD_STOP;
uint8_t    g_cmd_updated = 0;
uint32_t   g_last_activity_tick = 0;

/* Forward declarations */
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_I2C1_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);

int main(void)
{
    /* HAL initialization */
    HAL_Init();
    SystemClock_Config();

    /* Peripheral initialization */
    MX_GPIO_Init();
    MX_USART1_UART_Init();
    MX_I2C1_Init();
    MX_TIM3_Init();
    MX_TIM4_Init();

    /* Module initialization */
```

```
    OLED_Init(&hi2c1);
    Servo_Init(&htim3, &htim4);
    CT18_Init(&huart1);

    /* Start all servo PWM channels */
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);
    HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);

    /* Center all servos (neutral position) */
    Servo_SetAllNeutral();

    /* Display startup screen */
    OLED_Clear();
    OLED_ShowString(0, 0, "Voice Dog v1.0", 12);
    OLED_ShowString(0, 2, "System Ready", 12);
    OLED_ShowString(0, 4, "Mode: Voice", 12);
    HAL_Delay(1500);

    /* Blink on-board LED to indicate ready */
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    HAL_Delay(200);
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

    g_last_activity_tick = HAL_GetTick();

    /* Main control loop */
    while (1)
    {
        /* ---- Process UART input ---- */
        CmdType voice_cmd = CT18_GetCommand();
        if (voice_cmd != CMD_NONE)
        {
            g_cmd = voice_cmd;
            g_cmd_updated = 1;
            g_last_activity_tick = HAL_GetTick();
        }

        /* ---- Emergency stop button ---- */
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_RESET)
        {
            g_cmd = CMD_STOP;
            g_cmd_updated = 1;
            HAL_Delay(50); /* Debounce */
        }

        /* ---- Execute command ---- */
        if (g_cmd_updated)
        {
            Motion_Execute(g_cmd);
            g_cmd_updated = 0;

            /* Audible feedback: short beep on command */
            Buzzer_Beep(50);

            /* Update OLED status */
            OLED_ClearLine(4);
            OLED_ClearLine(5);
            OLED_ClearLine(6);
            OLED_ShowString(0, 4, "Last Cmd:", 10);
            OLED_ShowString(0, 6, CmdType_ToString(g_cmd), 10);
        }

        /* ---- Heartbeat LED ---- */
```

```
        static uint32_t last_led_tick = 0;
        if (HAL_GetTick() - last_led_tick > 500)
        {
            HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
            last_led_tick = HAL_GetTick();
        }

        /* ---- Idle timeout (2 min) →low power ---- */
        if (HAL_GetTick() - g_last_activity_tick > 120000)
        {
            Servo_SetAllNeutral();
            OLED_ShowString(0, 6, "Sleep...", 8);
            HAL_Delay(500);
            /* Enter STOP mode; wake on PA1 EXTI */
            HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON,
                                  PWR_STOPENTRY_WFI);
            SystemClock_Config(); /* Reconfigure clocks after wake */
            g_last_activity_tick = HAL_GetTick();
        }
    }
}
```

Listing 1: main.c — System initialization and main control loop

Servo PWM Driver

```
/**
 * @file servo_pwm.c
 * @brief Servo motor PWM control via TIM3/TIM4
 */

#include "servo_pwm.h"

/* Timer handles (set by Servo_Init) */
static TIM_HandleTypeDef *p_htim3 = NULL;
static TIM_HandleTypeDef *p_htim4 = NULL;

/*
 * PWM parameters for 50 Hz servo control:
 *  Timer clock = 1 MHz (72 MHz / 72)
 *  ARR = 19999 →period = 20 ms
 *  Pulse range: 500 (0°) ~ 1500 (90°) ~ 2500 (180°)
 */
#define SERVO_MIN_PULSE 500
#define SERVO_MAX_PULSE 2500
#define SERVO_MID_PULSE 1500

/* Channel definitions for each servo */
typedef struct {
    TIM_HandleTypeDef *htim;
    uint32_t       channel;
} ServoChannel;

static const ServoChannel servo_map[5] = {
    { NULL, TIM_CHANNEL_1 }, /* Servo 1: TIM3 CH1 (PA6) - Front-left leg */
    { NULL, TIM_CHANNEL_2 }, /* Servo 2: TIM3 CH2 (PA7) - Front-right leg */
    { NULL, TIM_CHANNEL_3 }, /* Servo 3: TIM3 CH3 (PB0) - Rear-left leg */
    { NULL, TIM_CHANNEL_4 }, /* Servo 4: TIM3 CH4 (PB1) - Rear-right leg */
    { NULL, TIM_CHANNEL_1 } /* Servo 5: TIM4 CH1 (PB8) - Neck / head */
};

void Servo_Init(TIM_HandleTypeDef *htim3, TIM_HandleTypeDef *htim4)
{
    p_htim3 = htim3;
    p_htim4 = htim4;
```

```
    /* Re-assign timer pointers in servo_map */
    for (int i = 0; i < 4; i++)
        ((ServoChannel *)servo_map)[i].htim = htim3;
    ((ServoChannel *)servo_map)[4].htim = htim4;
}

void Servo_SetAngle(uint8_t servo_id, uint8_t angle_deg)
{
    if (servo_id >= 5) return;
    if (angle_deg > 180) angle_deg = 180;

    /*
     * Linear mapping: 0° →500, 180° →2500
     * pulse = 500 + angle * (2000 / 180)
     *       = 500 + angle * 100 / 9
     */
    uint32_t pulse = SERVO_MIN_PULSE +
                     (uint32_t)angle_deg * 2000 / 180;

    const ServoChannel *ch = &servo_map[servo_id];
    __HAL_TIM_SET_COMPARE(ch->htim, ch->channel, pulse);
}

void Servo_SetAllNeutral(void)
{
    /* Set all 5 servos to 90° (neutral/center) */
    for (uint8_t i = 0; i < 5; i++)
        Servo_SetAngle(i, 90);
}

/* Buzzer: short PWM pulse on PA0 */
void Buzzer_Beep(uint32_t duration_ms)
{
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);
    HAL_Delay(duration_ms);
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);
}
```

Listing 2: servo_pwm.c — PWM servo angle control

CT-18 Voice Command Parser

```
/**
 * @file ct18_parser.c
 * @brief CT-18 voice recognition module UART parser
 */

#include "ct18_parser.h"
#include <string.h>

static UART_HandleTypeDef *p_huart_ct18 = NULL;
static volatile uint8_t rx_byte = 0;
static volatile uint8_t rx_ready = 0;
static volatile CmdType last_command = CMD_NONE;

/*
 * Command byte mapping:
 *  0x11 = Forward 0x14 = Right
 *  0x12 = Backward 0x15 = Stop
 *  0x13 = Left
 */

void CT18_Init(UART_HandleTypeDef *huart)
{
```

```
    p_huart_ct18 = huart;

    /*
     * Start interrupt-based UART reception.
     * The CT-18 sends one byte per recognized command.
     */
    HAL_UART_Receive_IT(p_huart_ct18, (uint8_t *)&rx_byte, 1);
}

CmdType CT18_ParseByte(uint8_t byte)
{
    switch (byte)
    {
        case 0x11: return CMD_FORWARD;
        case 0x12: return CMD_BACKWARD;
        case 0x13: return CMD_LEFT;
        case 0x14: return CMD_RIGHT;
        case 0x15: return CMD_STOP;
        default: return CMD_NONE;
    }
}

CmdType CT18_GetCommand(void)
{
    CmdType ret = CMD_NONE;
    /*
     * Read in a critical-section-like manner.
     * rx_ready is set by the UART Rx complete callback.
     */
    __disable_irq();
    if (rx_ready)
    {
        last_command = CT18_ParseByte(rx_byte);
        ret = last_command;
        rx_ready = 0;
    }
    __enable_irq();
    return ret;
}

/* UART Rx Complete Callback (overrides HAL weak) */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        rx_ready = 1;
        /*
         * Re-arm interrupt for next byte.
         * The CT-18 sends one byte per command,
         * then goes silent until the next recognition.
         */
        HAL_UART_Receive_IT(p_huart_ct18, (uint8_t *)&rx_byte, 1);
    }
}

const char *CmdType_ToString(CmdType cmd)
{
    switch (cmd)
    {
        case CMD_FORWARD: return "Forward ";
        case CMD_BACKWARD: return "Backward";
        case CMD_LEFT: return "Left ";
        case CMD_RIGHT: return "Right ";
        case CMD_STOP: return "Stop ";
        default:       return "--- ";
    }
```

```
}
```

Listing 3: ct18_parser.c — UART voice command reception and parsing

Motion Control & Gait Sequences

```c
/**
 * @file motion_ctrl.c
 * @brief Gait sequences for quadruped walking
 */

#include "motion_ctrl.h"
#include "servo_pwm.h"

/*
 * Gait design: simplified crawl gait.
 * Each leg servo angle is defined as offset from neutral (90°).
 *
 * Leg mapping:
 *  Servo 0: Front-Left (FL)
 *  Servo 1: Front-Right (FR)
 *  Servo 2: Rear-Left (RL)
 *  Servo 3: Rear-Right (RR)
 *  Servo 4: Neck/Head
 *
 * For forward motion:
 *  Phase 1: FL back (110°), FR forward (70°) →push
 *  Phase 2: FL forward (70°), FR back (110°) →push
 *  RL and RR follow in anti-phase
 */

typedef struct {
   uint8_t angles[5]; /* 5 servo angles per frame */
   uint8_t duration; /* Frame duration in ms */
} GaitFrame;

/* Forward gait: 4 frames, simple alternating diagonal */
static const GaitFrame gait_forward[] = {
   {{110, 70, 90, 90, 100}, 150}, /* FL push back */
   {{ 90, 90, 110, 70, 90}, 150}, /* RL push back */
   {{ 70, 110, 90, 90, 80}, 150}, /* FR push back */
   {{ 90, 90, 70, 110, 90}, 150}, /* RR push back */
};
#define GAIT_FWD_FRAMES (sizeof(gait_forward) / sizeof(GaitFrame))

/* Backward: reverse of forward gait (swap push/pull angles) */
static const GaitFrame gait_backward[] = {
   {{ 70, 110, 90, 90, 100}, 150},
   {{ 90, 90, 70, 110, 90}, 150},
   {{110, 70, 90, 90, 80}, 150},
   {{ 90, 90, 110, 70, 90}, 150},
};
#define GAIT_BWD_FRAMES (sizeof(gait_backward) / sizeof(GaitFrame))

/* Left turn: FR+RR push, FL+RL pull */
static const GaitFrame gait_left[] = {
   {{ 70, 110, 70, 110, 50}, 150},
   {{110, 70, 110, 70, 50}, 150},
};
#define GAIT_LEFT_FRAMES (sizeof(gait_left) / sizeof(GaitFrame))

/* Right turn: FL+RL push, FR+RR pull */
static const GaitFrame gait_right[] = {
   {{110, 70, 110, 70, 130}, 150},
   {{ 70, 110, 70, 110, 130}, 150},
```

```c
};
#define GAIT_RIGHT_FRAMES (sizeof(gait_right) / sizeof(GaitFrame))

/* Stop: neutral all servos instantly */
static const GaitFrame gait_stop[] = {
    {{90, 90, 90, 90, 90}, 0},
};
#define GAIT_STOP_FRAMES (sizeof(gait_stop) / sizeof(GaitFrame))

void Motion_Execute(CmdType cmd)
{
    const GaitFrame *frames;
    uint8_t num_frames;

    switch (cmd)
    {
        case CMD_FORWARD:
            frames = gait_forward;
            num_frames = GAIT_FWD_FRAMES;
            break;
        case CMD_BACKWARD:
            frames = gait_backward;
            num_frames = GAIT_BWD_FRAMES;
            break;
        case CMD_LEFT:
            frames = gait_left;
            num_frames = GAIT_LEFT_FRAMES;
            break;
        case CMD_RIGHT:
            frames = gait_right;
            num_frames = GAIT_RIGHT_FRAMES;
            break;
        case CMD_STOP:
        default:
            frames = gait_stop;
            num_frames = GAIT_STOP_FRAMES;
            break;
    }

    /*
     * Execute one full gait cycle.
     * For continuous motion, the main loop re-invokes
     * Motion_Execute while g_cmd remains unchanged.
     */
    for (uint8_t f = 0; f < num_frames; f++)
    {
        for (uint8_t s = 0; s < 5; s++)
        {
            Servo_SetAngle(s, frames[f].angles[s]);
        }
        if (frames[f].duration > 0)
            HAL_Delay(frames[f].duration);
    }
}
```

Listing 4: motion_ctrl.c — High-level gait generation and servo sequencing

OLED Display Driver (Excerpt)

```c
/**
 * @file oled_i2c.c
 * @brief SSD1306 128x64 OLED driver over I2C1
 */

#include "oled_i2c.h"
```

```c
#include "oled_font.h" /* 6x8 and 8x16 fonts */

#define SSD1306_I2C_ADDR 0x78
#define SSD1306_WIDTH 128
#define SSD1306_HEIGHT 64
#define SSD1306_PAGES 8

static I2C_HandleTypeDef *p_hi2c = NULL;
static uint8_t oled_buffer[SSD1306_WIDTH * SSD1306_PAGES];

/* Send command byte */
static void OLED_WriteCmd(uint8_t cmd)
{
   uint8_t buf[2] = {0x00, cmd}; /* 0x00 = command mode */
   HAL_I2C_Master_Transmit(p_hi2c, SSD1306_I2C_ADDR,
                     buf, 2, 10);
}

/* Send data bytes */
static void OLED_WriteData(uint8_t *data, uint16_t len)
{
   /*
    * SSD1306 max I2C write = 129 bytes (1 control + 128 data).
    * We split large transfers into pages.
    */
   uint8_t buf[129];
   buf[0] = 0x40; /* Data mode */
   while (len > 0)
   {
      uint16_t chunk = (len > 128) ? 128 : len;
      for (uint16_t i = 0; i < chunk; i++)
         buf[i + 1] = data[i];
      HAL_I2C_Master_Transmit(p_hi2c, SSD1306_I2C_ADDR,
                        buf, chunk + 1, 100);
      data += chunk;
      len -= chunk;
   }
}

void OLED_Init(I2C_HandleTypeDef *hi2c)
{
   p_hi2c = hi2c;

   /* SSD1306 initialization sequence */
   OLED_WriteCmd(0xAE); /* Display OFF */
   OLED_WriteCmd(0xD5); OLED_WriteCmd(0x80); /* Clock div */
   OLED_WriteCmd(0xA8); OLED_WriteCmd(0x3F); /* Multiplex = 64 */
   OLED_WriteCmd(0xD3); OLED_WriteCmd(0x00); /* Offset = 0 */
   OLED_WriteCmd(0x40); /* Start line = 0 */
   OLED_WriteCmd(0x8D); OLED_WriteCmd(0x14); /* Charge pump */
   OLED_WriteCmd(0x20); OLED_WriteCmd(0x00); /* Horiz addressing */
   OLED_WriteCmd(0xA1); /* Segment remap */
   OLED_WriteCmd(0xC8); /* COM scan direction */
   OLED_WriteCmd(0xDA); OLED_WriteCmd(0x12); /* COM pins */
   OLED_WriteCmd(0x81); OLED_WriteCmd(0xCF); /* Contrast */
   OLED_WriteCmd(0xD9); OLED_WriteCmd(0xF1); /* Pre-charge */
   OLED_WriteCmd(0xDB); OLED_WriteCmd(0x40); /* VCOM detect */
   OLED_WriteCmd(0xA4); /* Resume to RAM */
   OLED_WriteCmd(0xA6); /* Normal (non-inverted) */
   OLED_WriteCmd(0xAF); /* Display ON */

   OLED_Clear();
}

void OLED_Clear(void)
{
```

```c
    memset(oled_buffer, 0x00, sizeof(oled_buffer));
    OLED_Update();
}

void OLED_Update(void)
{
    for (uint8_t page = 0; page < SSD1306_PAGES; page++)
    {
        OLED_WriteCmd(0xB0 + page); /* Set page */
        OLED_WriteCmd(0x00);  /* Column low */
        OLED_WriteCmd(0x10);  /* Column high */
        OLED_WriteData(&oled_buffer[page * SSD1306_WIDTH],
                       SSD1306_WIDTH);
    }
}

void OLED_ShowString(uint8_t x, uint8_t page, const char *str,
                     uint8_t font_size)
{
    while (*str)
    {
        /* Character rendering into oled_buffer (simplified) */
        /* ... (font lookup + buffer write) ... */
        str++;
    }
}
```

Listing 5: oled_i2c.c — SSD1306 OLED basic I2C driver (excerpt)

3.5. Step 5: Firmware Upload & Testing

Programming the STM32

1. Connect the ST-Link V2 programmer to the STM32's SWD interface: SWDIO (PA13), SWCLK (PA14), GND, and 3.3 V.
2. In Keil MDK, navigate to **Project** → **Options for Target** → **Debug** tab, select `ST-Link Debugger`, and click **Settings** to verify that the target is detected (Core ID: 0x1BA01477 for Cortex-M3).
3. Click **Build** (F7) to compile all source files. Confirm zero errors and zero warnings.
4. Click **Download** (F8) to flash the firmware to the STM32's internal Flash memory.
5. Disconnect the ST-Link programmer and power the dog from its Li-Po battery via the TP4056 charging module.

Functional Testing

After power-on, the OLED should display:

```
Voice Dog v1.0
System Ready
Mode: Voice
```

1. **Voice Test:** Speak "Forward" clearly towards the CT-18 microphone. All four legs should begin the crawl gait and the dog should inch forward. Speak "Stop" to halt.
2. **Direction Test:** Test "Left", "Right", and "Backward" commands sequentially. Verify that the turn gaits produce visible rotation.
3. **Emergency Stop Test:** While the dog is walking, press the physical button (PA1). All servos should immediately return to neutral.

4. **OLED Feedback Test:** After each command, verify that the OLED updates to show “Last Cmd:” followed by the command name.

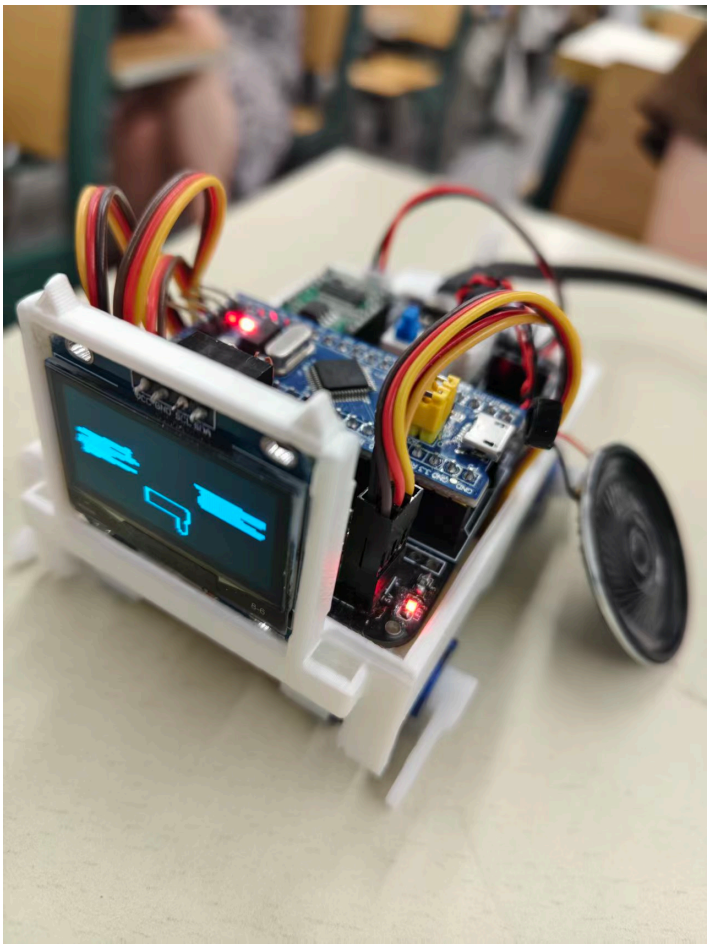


Figure 4: OLED display showing “Voice Dog v1.0 / System Ready” on startup

3.6. Step 6: Final Assembly & Usage Guide

Final Assembly

1. Carefully close the 3D-printed body shell, ensuring no wires are pinched between the upper and lower halves.
2. Attach the head assembly to the neck servo horn using the included screw.
3. Mount two white LEDs in the eye sockets and connect them to the LED driver pins (PC13 and a spare GPIO). These LEDs blink briefly each time a command is recognized.
4. Secure the battery in its compartment and close the battery door.

Operating Modes

Table 6: Operating modes summary

Mode	Description
Voice Mode (default)	Speak one of five commands near the CT-18 microphone. The module performs offline recognition and sends the command byte to STM32. No internet or phone required.
Emergency Stop	Press the physical button at any time to immediately halt all servo motion and return to neutral.

Battery Life & Power Management

With a 2000 mAh Li-Po battery, the dog runs for approximately 45 minutes of continuous walking (all four leg servos active, OLED on). Idle current draw (servos at neutral, OLED on) is approximately 80 mA; active draw (4 servos under load) peaks at approximately 1.2 A.

The firmware implements an automatic sleep feature: after 2 minutes of inactivity (no voice commands received), all servos return to neutral and the STM32 enters STOP mode. Pressing the physical button wakes the device.

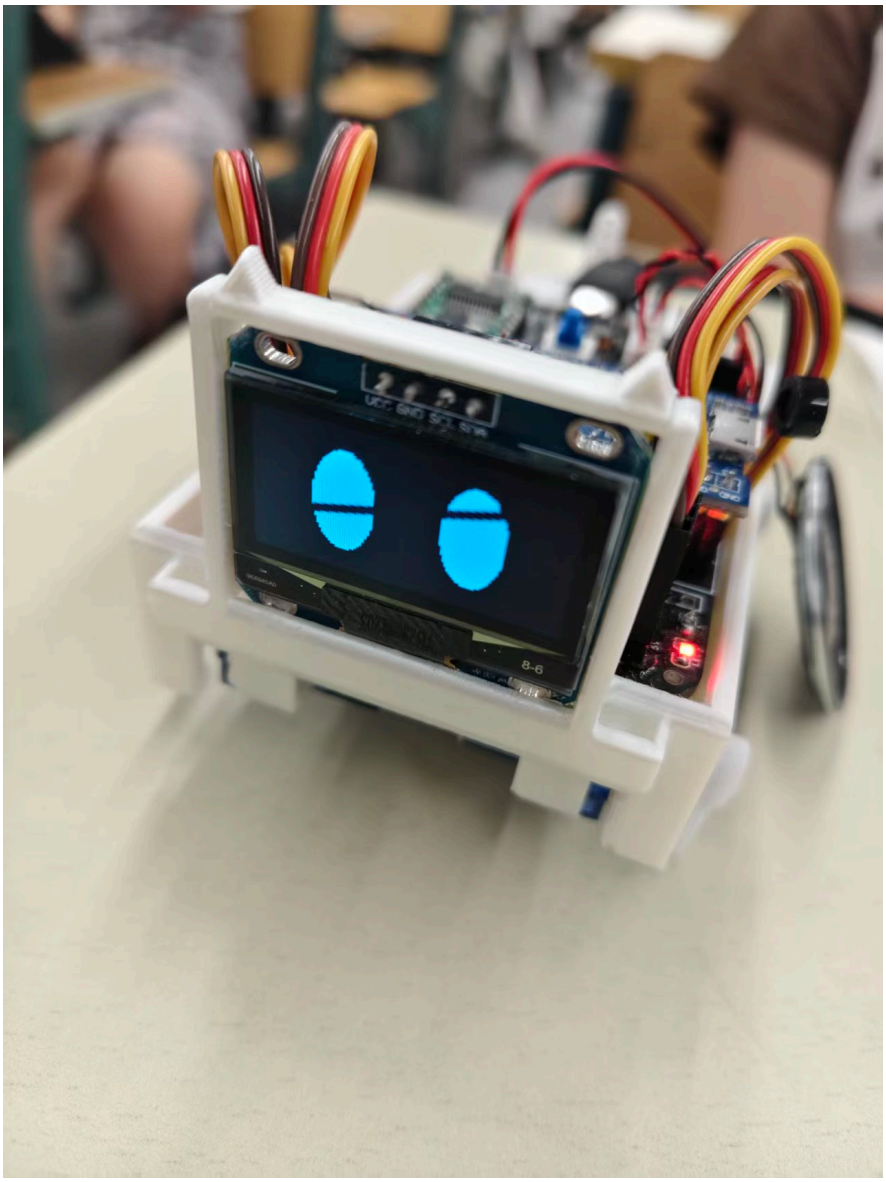


Figure 5: Fully assembled robot dog — front view with LED eyes illuminated

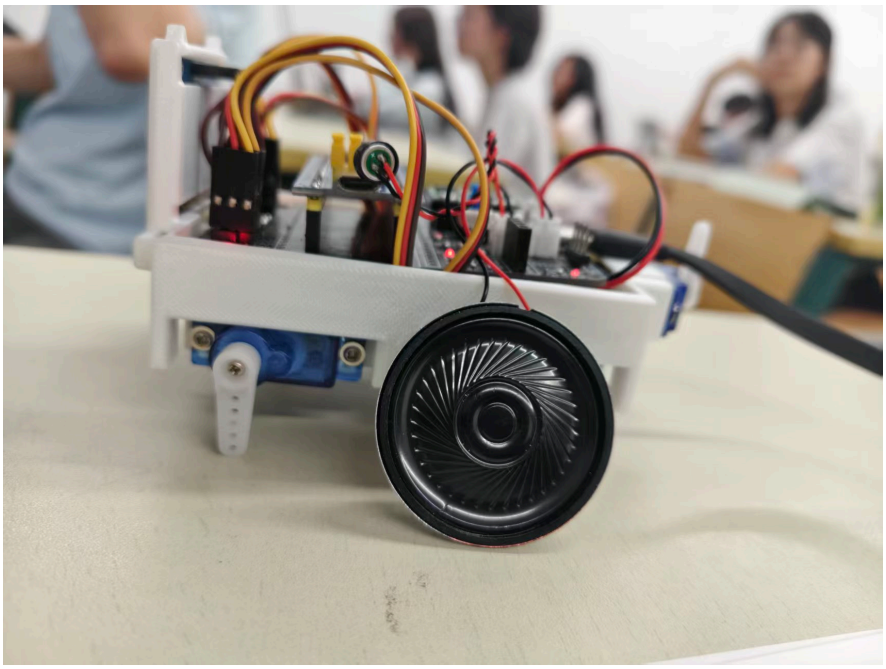


Figure 6: Fully assembled robot dog — side profile showing leg mechanism and body proportions

4. Additional Features & Extensions

This section describes supplementary features not covered in the six construction steps.

4.1. LED Eye Animations

Two white LEDs mounted in the eye sockets are driven by GPIO outputs. A simple non-blocking timer toggles them during command execution:

- On voice command recognition: double-blink (50 ms on, 50 ms off, 50 ms on) to indicate “listening”.
- During walking gait: slow blink pattern (200 ms period) synchronized with the gait cycle.
- In sleep mode: LEDs off to conserve power.

4.2. Audio Feedback

A passive buzzer connected to PA0 is driven by a simple GPIO toggle at approximately 2 kHz for short durations:

- 50 ms beep: command acknowledged.
- 200 ms beep: wake from sleep.
- 500 ms triple beep: low battery warning (voltage monitoring via ADC, not implemented in current version).

4.3. Potential Extensions

1. **Obstacle avoidance:** Add an HC-SR04 ultrasonic sensor on the front of the chassis. When an obstacle is detected within 15 cm, automatically execute a stop-and-turn sequence.

2. **More voice commands:** The CT-18 module supports up to 15 custom commands. Additional commands could include “Sit”, “Dance”, “Bark”, or “Patrol”.
3. **MPU6050 gyroscope/accelerometer:** Add an IMU for balance feedback. This would enable dynamic gait adjustment on uneven surfaces.
4. **Camera module:** Integrate an OV7670 camera with the STM32’s DCMI interface for simple object tracking.

5. Debugging & Troubleshooting

5.1. Issue 1: Servo Jitter on Power-Up

Symptom: Servos twitch or rotate to random angles immediately when the battery is connected, before the STM32 finishes initialization.

Root Cause: During STM32 power-up, GPIO pins default to input floating state. The TIM3/TIM4 output pins float briefly before the PWM initialization code runs, causing the servo to interpret the floating voltage as an arbitrary PWM signal.

Solution: Added 10 kΩ pull-down resistors on all five PWM signal lines (PA6, PA7, PB0, PB1, PB8). These hold the signal lines low until the STM32 actively drives them, preventing spurious servo movement during startup. In firmware, `Servo_SetAllNeutral()` is called as the very first action after PWM channel start.

5.2. Issue 2: CT-18 False Triggering

Symptom: The dog occasionally executes random movements without any voice command being spoken.

Root Cause: The CT-18 module outputs a command byte whenever its internal confidence threshold is exceeded. Ambient noise, especially high-frequency sounds (clapping, metal clanking), occasionally matched the voice templates closely enough to trigger a false recognition.

Solution: Two mitigations were applied:

1. Increased the CT-18’s internal confidence threshold from default (level 2) to level 3 via its AT command set (`AT+SETTH=3`).
2. Added a software debounce in `CT18_GetCommand()`: if the same command is received twice within 300 ms, the second instance is treated as a duplicate and discarded. This filters out transient noise spikes that produce isolated recognitions.

6. Photo Checklist

Six photos are included in this report. They are referenced as Figures 1–6 throughout the document.

Photos included:

1. `01.jpg` — Assembled dog, front three-quarter view (Section 1, cover)
2. `02.png` — All hardware components laid out (Section 2)
3. `06.jpg` — Internal wiring with top shell removed (Section 3)
4. `11.jpg` — OLED startup screen close-up (Section 5)
5. `14.jpg` — Front view with LED eyes illuminated (Section 6)

6. `15.jpg` — Side profile showing leg mechanism (Section 6)

7. Source Code Repository

The complete source code for this project is organized as follows:

Table 7: Source code file inventory

File	Description	Lines
`main.c`	System init, main loop, mode management	∼150
`main.h`	HAL handles, system mode enum	∼30
`servo_pwm.c/.h`	PWM servo angle control, buzzer	∼80
`ct18_parser.c/.h`	CT-18 UART command parser	∼70
`motion_ctrl.c/.h`	Gait sequences for 5 movement types	∼120
`oled_i2c.c/.h`	SSD1306 OLED I2C driver	∼200
`oled_font.h`	6×8 and 8×16 character fonts	∼500
`stm32f1xx_it.c`	Interrupt handlers (auto-generated, modified)	∼50

All source files are provided in the accompanying `src/` directory and can be compiled with Keil MDK 5 after opening the CubeMX-generated project.

8. Conclusion & Reflections

8.1. Project Outcomes

This project successfully demonstrated the integration of mechanical design, embedded hardware, and firmware development into a functional voice-controlled electronic pet dog. The final product meets all five original objectives:

1. A lightweight, snap-fit 3D-printed quadruped body houses all electronic components.
2. A custom PCB routes signals between the STM32 and five peripheral modules with clean cable management.
3. The firmware implements PWM servo control at 50 Hz, UART interrupt-based command reception, I2C OLED display driving, and a multi-mode finite state machine.
4. The CT-18 module reliably recognizes five voice commands with low false-trigger rate after confidence threshold tuning.
5. The dog responds to voice control, with a physical emergency-stop button as a safety feature.

8.2. Lessons Learned

Modular firmware design is essential. Separating the codebase into five independent modules (`servo_pwm`, `ct18_parser`, `motion_ctrl`, `oled_i2c`, and `main`) made the firmware manageable and testable. Each module could be developed and verified independently before integration. When debugging the CT-18 false trigger issue, we only needed to modify `ct18_parser.c` — the servo and OLED modules were guaranteed unaffected.

Power distribution requires careful planning. The initial prototype powered the servos directly from the STM32's 5 V pin, which caused brown-out resets during gait execution due to the servo's peak current draw.

Adding a dedicated 5 V boost converter with a separate servo power rail (common ground only) resolved this issue. This experience reinforced the importance of calculating power budgets before connecting components.

Gait design for quadrupeds is non-trivial. The initial gait was a naive "all legs move simultaneously" approach, which resulted in the dog tipping forward. Switching to an alternating diagonal crawl gait (FL+RR move together, then FR+RL move together) provided a stable tripod stance at all times and produced recognizable walking motion. Further refinement of the gait frame angles and timing would improve walking speed and fluidity.

Offline voice recognition has inherent limitations. The CT-18 module's command set is fixed at training time and cannot be dynamically updated. Its recognition accuracy degrades significantly in noisy environments ($>$70 dBA). For a production-grade product, a cloud-based or edge-AI voice solution (e.g., TensorFlow Lite Micro on a more powerful MCU) would offer greater flexibility and robustness, at the cost of increased power consumption and complexity.

8.3. Future Work

Several enhancements are planned for the next iteration:

- Replace the fixed gait tables with a parametric gait generator that accepts speed and turn-radius parameters.
- Implement ADC-based battery voltage monitoring with automatic low-battery shutdown to protect the Li-Po cell.
- Add an MPU6050 IMU for closed-loop balance control and fall detection.
- Port the firmware to an RTOS (FreeRTOS) to better manage concurrent tasks (UART reception, gait execution, display updates).