M55M1 Gesture-Controlled Smart Fan

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code implements machine vision-based image recognition on the M55M1 development platform. It can detect and recognize user hand gestures and human body positions to enable intelligent fan control, allowing remote fan operation and automatic human tracking functionality.
BSP Version	M55M1_Series_BSP_CMSIS_V3.01.004
Hardware	NuMaker-GestureAI-M55M1 V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example code demonstrates how to implement image recognition on the NuMaker-GestureAI-M55M1 development platform by combining machine vision algorithms to detect user gestures and human body position information, thereby enabling intelligent fan control. Through real-time image analysis, the system interprets gesture commands to control fan operations such as start, stop, and operating modes. At the same time, it utilizes a human-tracking mechanism to automatically adjust the fan's direction and follow the target's position. This approach enables intuitive long-distance control without requiring physical contact or a conventional remote controller, while enhancing user convenience and interaction experience. It serves as a practical reference for integrating embedded AI technologies into smart home appliance applications.

1.1 NuMaker-GestureAI-M55M1 Development Platform

The M55M1 series microcontrollers are built on the Arm® Cortex®-M55 core, operating at frequencies of up to 220 MHz. They incorporate Arm® Helium™ DSP technology to significantly enhance digital signal processing capabilities, and integrate the Arm® Ethos™-U55 microNPU, providing strong performance for both DSP tasks and machine learning inference. This architecture enables efficient acceleration of machine learning workloads on edge devices.

This example uses the NuMaker-GestureAI-M55M1 module, which integrates a camera sensor along with communication interfaces such as I²C, UART, and USB, as shown in Figure 1-1. The module fully leverages the high-performance processing power and dedicated AI features of the M55M1 series, helping developers implement sophisticated artificial intelligence applications on edge devices. To simplify AI model deployment, the module is equipped with a Micro SD card slot for storing AI model files. In addition, the onboard camera enables seamless image capture, video processing, and computer vision recognition, allowing AI applications to efficiently perform visual sensing and analysis tasks.

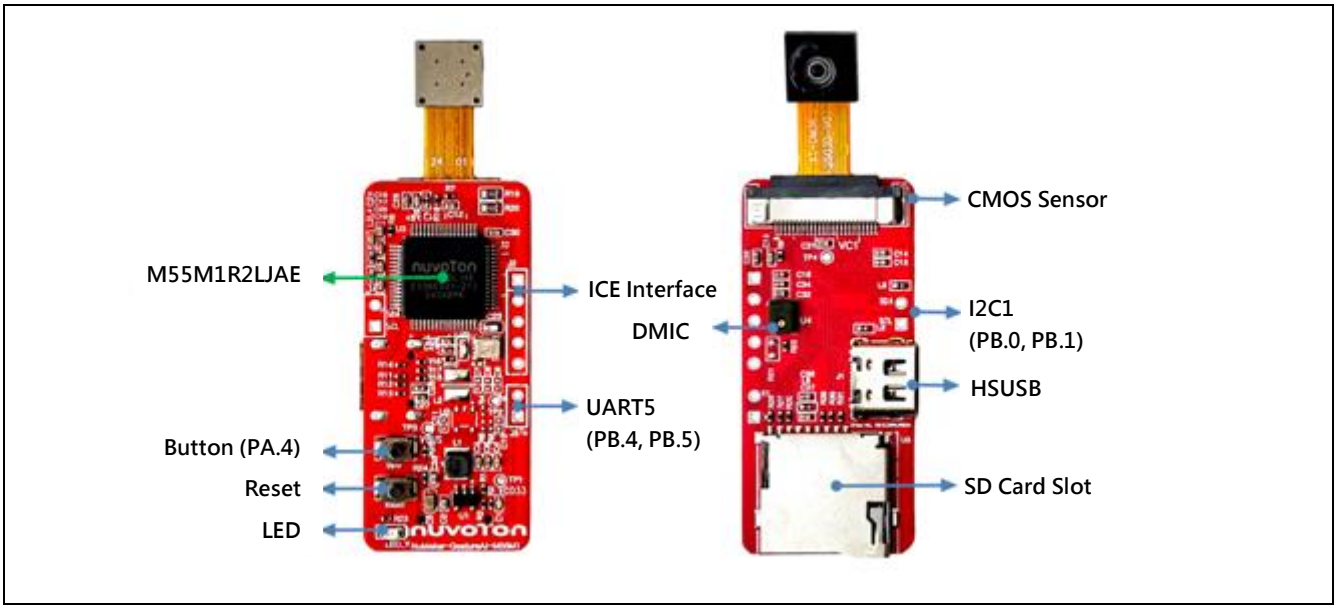


Figure 1-1 NuMaker-GestureAI-M55M1 Module

1.2 Principle

This example demonstrates how the M55M1 series microcontroller (MCU) can transform a conventional fan into a proactive, perception-enabled smart appliance. By performing high-performance object detection directly on the edge device, the system enables completely offline, low-latency, and privacy-preserving touchless control.

The gesture recognition system replaces traditional physical buttons with intuitive hand-gesture interactions. Users can show an “OK” gesture to adjust the fan speed, a “Fist” gesture to trigger smart power-off, or a “Peace” gesture to activate wide-angle oscillation, as shown in Figure 1-2. The key highlight is the Smart Tracking Mode, which can be enabled through gesture control. In this mode, the system continuously detects and tracks the user's position in real time, driving the fan’s oscillation motor to follow the target automatically. As a result, the fan can dynamically adjust its direction according to the user's movement, demonstrating a smart home appliance application that integrates computer vision AI and motor control on a single microcontroller platform.

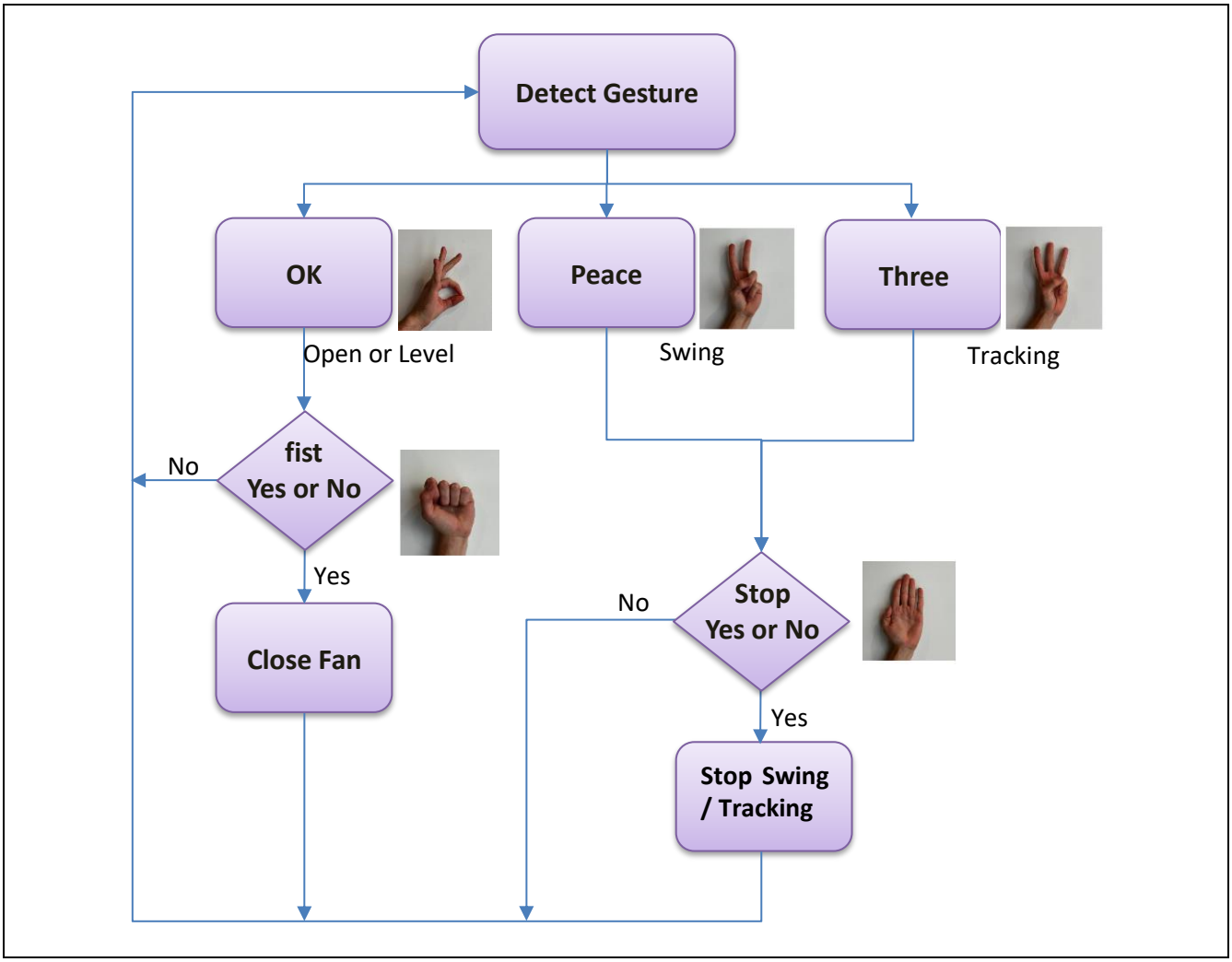


Figure 1-2 System Diagram

The vision AI system in this example is based on YOLOX Nano, a lightweight, anchor-free object detection model designed specifically for real-time inference and edge computing devices. Its architecture adopts a decoupled head, which separates classification and bounding box regression tasks, improving both training stability and detection accuracy.

Compared with traditional anchor-based approaches, YOLOX Nano directly predicts object locations and class probabilities on the feature maps, eliminating the need for complex anchor configurations and reducing the computational overhead of post-processing. Owing to its compact model size and low computational requirements, YOLOX Nano is particularly well suited for MCU-based AI applications, providing reliable object and gesture detection performance while maintaining real-time responsiveness on resource-constrained embedded systems.

1.3 Demo Result

The compilation and programming results of this example are shown in Figure 1-3. Through the USB interface on the NuMaker-GestureAI-M55M1 module, the device can operate as an external USB camera for a PC while simultaneously displaying the hand gestures detected by the onboard camera.

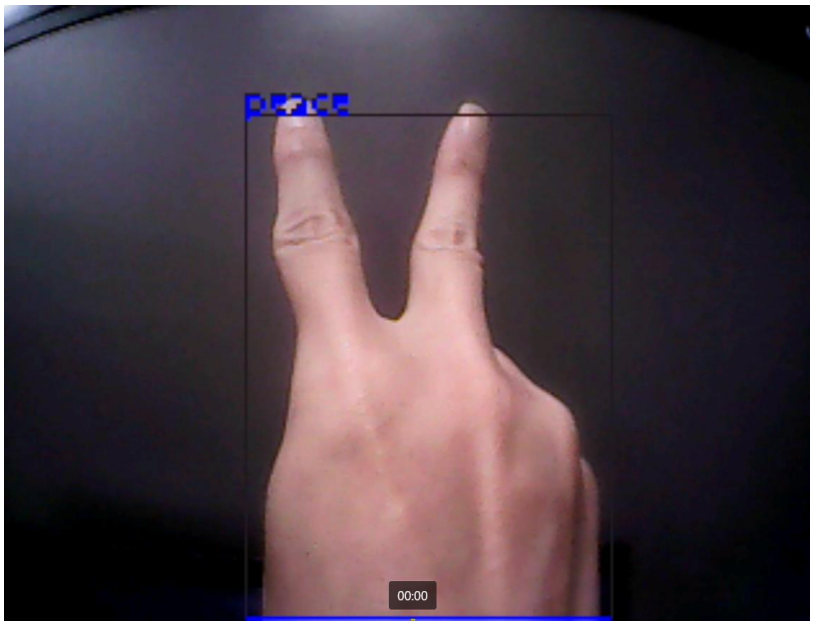


Figure 1-3 Operation Result

2. Code Description

This example performs overall system initialization in the main() function, including clock source configuration, initialization of the fan and stepper motor states, creation of the Fan Command Queue, and startup of the UVC device when required. It then creates two FreeRTOS tasks, main_task and vFanControlTask, and finally calls vTaskStartScheduler() to start the FreeRTOS scheduler, allowing the system to enter a multitasking execution environment.

The main_task handles the gesture control logic and creates a third FreeRTOS task, inferenceProcessTask, which is responsible for AI inference. The vFanControlTask handles fan operation control and stepper motor oscillation control. Commands are delivered through a queue mechanism, enabling vFanControlTask to receive instructions and perform the corresponding hardware operations.

```
int main()
{
    BaseType_t ret;

    FAN_GPIOInitIdleHigh();

    /* Initialize the UART module to allow printf related functions (if using retarget) */
    BoardInit();

    /* ------------------------------------------------------------------
     * Fan / stepper initialization
     *   - Do NOT start the fan on power-up.
     *   - Keep PA4 at idle HIGH to avoid an unintended High->Low edge.
     * ------------------------------------------------------------------ */
    FAN_Init();
    g_i32CurrentAngleX100 = kFanCenterAngleX100;
    g_i32TotalSteps = 0;
    g_u8MoveDone = 0;

    FAN_GPIOInitIdleHigh();

    g_fanCmdQueue = xQueueCreate(16, sizeof(FanCmdMsg));

    if (g_fanCmdQueue == nullptr)
    {
        info("FreeRTOS: Failed to create fan command queue\n");
        return -1;
    }

    #if defined(__USE_UVC__)
    UVC_Init();
    NVIC_SetPriority(HSUSBD_IRQn, 5);
    HSUSBD_Start();
    #endif

    /* Create main task. */
    ret = xTaskCreate(main_task, "main task", 2 * 1024, nullptr, MAINLOOP_TASK_PRIO,
nullptr);
```

```
    if (ret != pdPASS)
    {
        info("FreeRTOS: Failed to create main loop task \n");
        return -1;
    }

    ret = xTaskCreate(vFanControlTask, "fan_task", 1024, nullptr, 2, nullptr);

    if (ret != pdPASS)
    {
        info("FreeRTOS: Failed to create fan task \n");
        return -1;
    }

    // Start Scheduler
    vTaskStartScheduler();

    /* This is unreachable without errors. */
    info("program terminating...\n");
}
```

The main_task() function is responsible for creating and initializing the YOLOX Nano AI model, inference task, and image sensor, while managing image data through multiple frame buffers. Within its main loop, the program continuously captures image frames and sends them to inferenceProcessTask for AI inference.

After the inference results are obtained, the function calls HandleGestureAfterInference() to perform gesture recognition and debounce processing. Based on the recognized gestures, the system can control fan speed, turn the fan on or off, activate oscillation, and enable target-tracking functionality.

In addition, the program performs bounding-box rendering and UVC video streaming during runtime, forming a complete application workflow of “image capture → AI inference → gesture control → video output”.

```
static void main_task(void *pvParameters)
{
    BaseType_t ret;

    info("main task running \n");
    /* Model object creation and initialisation. */
    arm::app::YoloXnanoNu model;

    if (!model.Init(arm::app::tensorArena,
                    sizeof(arm::app::tensorArena),
                    arm::app::yoloxnanonu::GetModelPointer(),
                    arm::app::yoloxnanonu::GetModelLen()))
    {
        printf_err("Failed to initialise model\n");
        vTaskDelete(nullptr);
        return;
    }
…
    ret = xTaskCreate(inferenceProcessTask, "inference task", 1 * 1024, &taskParam,
INFERENCE_TASK_PRIO, nullptr);
```

```
…
    TfLiteTensor *inputTensor   = model.GetInputTensor(0);
    TfLiteTensor *outputTensor = model.GetOutputTensor(0);
…

#if defined (__USE_CCAP__)
    //Setup image sensor
    ImageSensor_Init();
    ImageSensor_Config(eIMAGE_FMT_RGB565, frameBuffer.w, frameBuffer.h, true);
#endif

    while (1)
    {
        infFramebuf = get_inf_framebuf();

        if (infFramebuf)
        {
            /* Detector post-processing*/
            inferenceJob->responseQueue = inferenceResponseQueue;
            inferenceJob->pPostProc = &postProcess;
            inferenceJob->modelCols = inputImgCols;
            inferenceJob->mode1Rows = inputImgRows;
            inferenceJob->srcImgWidth = infFramebuf->frameImage.w;
            inferenceJob->srcImgHeight = infFramebuf->frameImage.h;
            inferenceJob->results = &infFramebuf->results; //&results;

            xQueueReceive(inferenceResponseQueue, &inferenceJob, portMAX_DELAY);
        }

        fullFramebuf = get_full_framebuf();

        if (fullFramebuf)
        {
            //resize full image to input sensor
            image_t resizeImg;

            roi.x = 0;
            roi.y = 0;
            roi.w = fullFramebuf->frameImage.w;
            roi.h = fullFramebuf->frameImage.h;

            resizeImg.w = inputImgCols;
            resizeImg.h = inputImgRows;
            resizeImg.data = (uint8_t *)inputTensor->data.data; //direct resize to input
sensor buffer
            resizeImg.pixfmt = PIXFORMAT_RGB888;

            imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);

            /* If the data is signed. */
            if (model.IsDataSigned())
            {
                arm::app::image::ConvertImgToInt8(inputTensor->data.data, inputTensor-
>bytes);
            }
```

```
            //trigger inference
            inferenceJob->responseQueue = inferenceResponseQueue;
            inferenceJob->pPostProc = &postProcess;
            inferenceJob->modelCols = inputImgCols;
            inferenceJob->mode1Rows = inputImgRows;
            inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
            inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
            inferenceJob->results = &fullFramebuf->results;

            xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
            fullFramebuf->eState = eFRAMEBUF_INF;
        }

        if (infFramebuf)
        {

            /* Smart gesture fan control: run AFTER YOLO results are ready. */
            HandleGestureAfterInference(infFramebuf->results);

            // draw bbox and render
            DrawImageDetectionBoxes(infFramebuf->results, &infFramebuf->frameImage,
labels);
            TrackObject(infFramebuf->results);

#if defined (__USE_UVC__)
            if (UVC_IsConnect())
            {
                Rotate180InPlace_2Bpp((uint8_t*)infFramebuf->frameImage.data,
                                      infFramebuf->frameImage.w,
                                      infFramebuf->frameImage.h);

                SCB_CleanDCache_by_Addr((uint32_t*)infFramebuf->frameImage.data,
                                        infFramebuf->frameImage.w * infFramebuf-
>frameImage.h * 2U);
                __DSB();
                __ISB();

                UVC_SendImage((uint32_t)infFramebuf->frameImage.data,
                              infFramebuf->frameImage.w * infFramebuf->frameImage.h * 2U,
                              uvcStatus.StillImage);
            }
#endif
            PresentInferenceResult(infFramebuf->results, labels);
            infFramebuf->eState = eFRAMEBUF_EMPTY;
        }

        emptyFramebuf = get_empty_framebuf();

        if (emptyFramebuf)
        {

#if defined (__USE_CCAP__)
            //capture frame from CCAP
            ImageSensor_Capture((uint32_t)(emptyFramebuf->frameImage.data));
#endif
```

```
            emptyFramebuf->results.clear();
            emptyFramebuf->eState = eFRAMEBUF_FULL;
        }

        vTaskDelay(1);
    }

}
```

The inferenceProcessTask() function is a dedicated AI inference task that continuously waits for inference jobs submitted by main_task. Once a job is received, it calls InferenceProcess::RunJob() to perform model inference and post-processing. After the inference is completed, the results are returned to the main workflow through a response queue.

This architecture prevents computationally intensive inference operations from blocking the main task, allowing image acquisition, AI inference, and gesture control to run concurrently for improved system responsiveness and overall performance.

```
void inferenceProcessTask(void *pvParameters)
{
    struct ProcessTaskParams params = *reinterpret_cast<struct ProcessTaskParams
*>(pvParameters);

    InferenceProcess::InferenceProcess inferenceProcess(params.model);

    for (;;)
    {
        xInferenceJob *xJob;

        xQueueReceive(params.queueHandle, &xJob, portMAX_DELAY);

        inferenceProcess.RunJob(
            xJob->pPostProc,
            xJob->modelCols,
            xJob->mode1Rows,
            xJob->srcImgWidth,
            xJob->srcImgHeight,
            xJob->results
        );

        xQueueSend(xJob->responseQueue, &xJob, portMAX_DELAY);
    }

}
```

The vFanControlTask() function is responsible for fan and stepper motor control. After system startup, it performs an automatic calibration procedure for the stepper motor by first moving it to the right limit position and then returning it to the front reference position, thereby establishing a reliable angular reference.

The task receives fan control commands from gesture recognition or the application layer through g_fanCmdQueue. It generates control pulses on GPIO PA.4 at fixed time intervals to switch fan speed levels while continuously maintaining the actual fan operating state.

The task also supports a high-priority FORCE_OFF command, which can clear any pending control pulses and calculate the required number of pulses needed to turn off the fan. This mechanism ensures that fan control and stepper motor operations can continue running in the background without interfering with AI inference and other real-time processing tasks.

```
void vFanControlTask(void *pvParameters)
{
    (void)pvParameters;

    /*
     * Fan pulse task
     *   - Receives fan commands from gesture logic (main_task)
     *   - Generates PA4 pulses with a minimum inter-pulse gap
     *
     * IMPORTANT: This task MUST NOT start the fan on power-up.
     */

    /* Ensure PA.4 stays idle HIGH (avoid unintended falling edge at boot). */
    FAN_GPIOInitIdleHigh();

    /* ------------------------------------------------------------------
     * Stepper boot calibration (restore original behavior)
     *   Step1: CW 1030 steps (go to right end)
     *   Step2: reset angle to 0, then CCW 560 steps (go to front)
     *
     * NOTE: We DO NOT start the fan here anymore.
     * ------------------------------------------------------------------ */
    static bool s_calStarted = false;

    if (!s_calStarted)
    {
        g_stepperCalibrated = false;
        g_u8AutoTestState = 1;
        g_u8MoveDone = 0;
        STEPPER_MoveSteps(1030, 1);
        s_calStarted = true;
        info("[INIT] stepper auto-calibration start\n");
    }

    /* Fan speed state:
     *   0=OFF, 1=Speed1, 2=Speed2, 3=Speed3, 4=Speed4, 5=Speed5
     * NOTE: This variable MUST represent the *actual physical* level.
     *       Therefore we only update it AFTER a pulse is generated.
     */
    static uint8_t s_fanLevel = 0;

    uint8_t pendingPulses = 0;
    TickType_t nextPulseTick = 0;

    const TickType_t kPulseGapTicks = pdMS_TO_TICKS(350);
    const uint8_t kMaxPendingPulses = 20;

    info("Fan pulse task started (gesture-controlled)\n  ");

// --- Boot auto force-off (one-shot) ----------------------------------------
```

```
// Assumption: fan controller powers up at Speed1.
// States: 0(off) + 5 speeds => 6 states total.
// To go from 1 -> 0 needs 5 pulses: 1->2->3->4->5->0
    static bool s_bootForceOffDone = false;
    if (!s_bootForceOffDone)
    {
        // Sync SW level to known HW boot state.
        s_fanLevel = 1U;

        // Schedule pulses immediately; gap is enforced by kPulseGapTicks in the task
loop.
        pendingPulses = 5U;
        nextPulseTick = 0;

        s_bootForceOffDone = true;
        info("[INIT] boot auto force-off: 5 pulses scheduled\n");
    }

    for (;;)
    {
…

        /* Stepper calibration state machine: advance when move completes. */
        if (g_u8MoveDone)
        {
            g_u8MoveDone = 0;

            if (g_u8AutoTestState == 1)
            {
                /* Reached right end. Define this as angle 0.00 deg. */
                g_i32TotalSteps = 0;
                g_i32CurrentAngleX100 = 0;

                g_u8AutoTestState = 2;
                STEPPER_MoveSteps(560, -1);
                info("[INIT] stepper auto-calibration step2 (to front)\n");
            }
            else if (g_u8AutoTestState == 2)
            {
                g_u8AutoTestState = 0;
                g_stepperCalibrated = true;
                info("[INIT] stepper auto-calibration complete\n");
            }
        }

        bool forceOffRequested = false;

        /* Drain all pending commands (non-blocking). */
        FanCmdMsg msg;

        while ((g_fanCmdQueue != nullptr) && (xQueueReceive(g_fanCmdQueue, &msg, 0) ==
pdTRUE))
        {
            if (msg.cmd == FAN_CMD_FORCE_OFF)
            {
                forceOffRequested = true;
```

```
                /* Keep draining to clear the queue. */
            }
            else if (msg.cmd == FAN_CMD_NEXT_LEVEL)
            {
                /* If FORCE_OFF is requested in the same burst, ignore NEXT_LEVEL. */
                if (!forceOffRequested)
                {
                    if (pendingPulses < kMaxPendingPulses)
                    {
                        pendingPulses++;
                    }
                }
            }
        }

        if (forceOffRequested)
        {
            /*
             * Priority command: FORCE_OFF
             *   - discard any queued NEXT_LEVEL pulses
             *   - compute pulses needed from the *actual* current level
             */
            pendingPulses = 0;

            const uint8_t pulsesNeeded = (s_fanLevel == 0U) ? 0U : (uint8_t)(6U -
s_fanLevel);

            pendingPulses = pulsesNeeded;

            /* If other tasks send during this moment, they will be handled next loop. */
        }

        const TickType_t now = xTaskGetTickCount();

        if ((pendingPulses > 0U) && (nextPulseTick == 0 || now >= nextPulseTick))
        {
            generate_pulse();
            pendingPulses--;

            /* Update actual fan level AFTER pulse. */
            s_fanLevel++;

            if (s_fanLevel >= 6U)
            {
                s_fanLevel = 0U;
            }

            nextPulseTick = now + kPulseGapTicks;
        }

        vTaskDelay(pdMS_TO_TICKS(20));
    }
}
```

The HandleGestureAfterInference() function is responsible for converting the object detection results generated by YOLOX Nano into actual control commands. It first selects the most

suitable gesture candidate from each frame and applies cross-frame debounce and cooldown mechanisms to improve recognition stability, preventing transient misdetections from triggering unintended actions.

Once a gesture is confirmed, the function executes the corresponding control behavior based on the recognized gesture class. These actions include fan speed adjustment, fan shutdown, activation or deactivation of stepper motor oscillation, and human-tracking functionality. The resulting fan control commands are then sent to g_fanCmdQueue through SendFanCmd(), where they are processed by vFanControlTask. This example provides a robust and reliable gesture-driven control framework.

```
static void HandleGestureAfterInference(const
std::vector<arm::app::object_detection::DetectionResult> &results)
{
    double score = 0.0;
    int area = 0;

    /* 1) Pick a raw gesture candidate from this frame (or none). */
    const int candidateCls = FindBestGestureCandidate(results, score, area);

    /* 2) Debounce across frames: only fire on stable gesture-change edge. */
    const int firedCls = GestureFilterUpdate(candidateCls);

    if (firedCls < 0)
    {
        return;
    }

    const TickType_t now = xTaskGetTickCount();

    /* 3) Additional cooldown (protect against brief flicker changing stable state). */
    if (firedCls >= 0 && firedCls < 12)
    {
        const TickType_t cooldown = GestureCooldownTicks(firedCls);
        const TickType_t last = g_lastGestureFireTick[firedCls];

        if (last != 0 && (now - last) < cooldown)
        {
            return;
        }

        g_lastGestureFireTick[firedCls] = now;
    }

    switch (firedCls)
    {
        case kClsOk:
            /* Fan: cycle speed/off by one step. */
            SendFanCmd(FAN_CMD_NEXT_LEVEL);
            info("[GESTURE] ok -> fan next level (score=%.2f area=%d)\n", score, area);
            break;

        case kClsFist:
            /* Fan: force off by sending remaining pulses. */
            SendFanCmd(FAN_CMD_FORCE_OFF);
```

```
            info("[GESTURE] fist -> fan force off (score=%.2f area=%d)\n", score, area);
            break;

        case kClsPeace:
            /* Motor: start swing. */
            g_swingMode = true;
            STEPPER_Abort();
            info("[GESTURE] peace -> start swing (score=%.2f area=%d)\n", score, area);
            break;

        case kClsStop:
            /*
             * "stop" should stop swing & tracking:
             *   - stop swing
             *   - stop tracking
             */
            g_swingMode = false;
            g_trackingMode = false;
            g_cyclePersons = false;
            g_cycleIndex = 0;
            g_nextPersonSwitchTick = 0;
            STEPPER_Abort();
            info("[GESTURE] stop -> stop swing & tracking (score=%.2f area=%d)\n", score,
area);
            break;

        case kClsThree:
            /* Enter tracking mode. */
            g_trackingMode = true;
            g_swingMode = false;
            g_cyclePersons = false;
            g_cycleIndex = 0;
            g_nextPersonSwitchTick = 0;
            STEPPER_Abort();
            info("[GESTURE] three -> enter tracking mode (score=%.2f area=%d)\n", score,
area);
            break;

        case kClsPalm:
            g_trackingMode = true;
            /* In tracking mode: toggle cycling among detected persons. */
            if (g_trackingMode)
            {
                g_cyclePersons = !g_cyclePersons;
                g_cycleIndex = 0;
                g_nextPersonSwitchTick = now + pdMS_TO_TICKS(5000);
                info("[GESTURE] palm -> cycle persons %s (score=%.2f area=%d)\n",
                     g_cyclePersons ? "ON" : "OFF", score, area);
            }

            break;

        default:
            /* Other gesture classes are ignored in this spec. */
            break;
    }
```

```
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M55M1_Series_BSP_CMSIS_V3.01.004
- IDE version
  - Keil uVision 5.43
  - VS Code V1.112.0
    - CMSIS Solution extension
    - Arm Tools Environment Manager
    - Arm Compiler (ARMClang 6.x)

3.2 Hardware Requirements

- Circuit components
  - NuMaker-GestureAI-M55M1
- Pin Connect
  - Connect the NuMaker-GestureAI-M55M1 to the fan control board (integrated in the fan body) via wires, as shown in Figure 3-1.

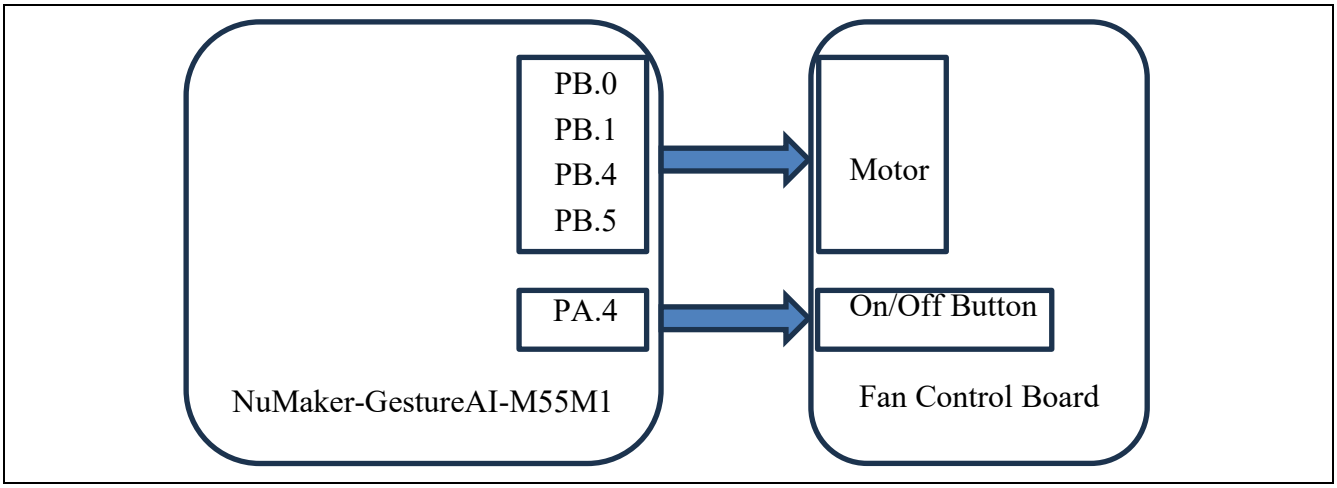


Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M55M1_Gesture_Smart_Fan_Control_V1.00	
🗁 Library	Core software libraries and device support files
🗁 CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
🗁 Device	CMSIS compliant device header file
🗁 StdDriver	All peripheral driver header and source files
🗁 ThirdParty	
🗁 FreeRTOS	Real-time operating system kernel for embedded devices
🗁 ml-embedded-evaluation-kit	Machine learning embedded evaluation examples and reference applications
🗁 openmv	MicroPython-based machine vision framework for embedded cameras
🗁 tflite_micro	TensorFlow Lite for Microcontrollers inference library for tiny ML on MCU
🗁 SampleCode	
🗁 ExampleCode	
🗁 GestureFanControl	Example source code directory
🗀 Keil	Source code of Keil project
🗀 VSCode	Source code of VS Code project

Figure 4-1 Directory Structure

5. Example Code Execution

5.1 Keil MDK

1. Navigate to the **Keil** project folder in Windows Explorer and open the *GestureFanControl.uvprojx* project file.
2. Build
3. Download
4. Enter debug mode.
   - Run

5.2 VS Code

1. Select **File** in **VS Code** IDE
   - Click **Open Folder**
   - Select the **VSCode** folder under the project directory
2. Select **CMSIS**
   - Click **Build solution** to compile the project
   - Click **Load & Run application** to program the target chip
   - Click **Load & Debug application** to enter debug mode

6. Revision History

Date	Revision	Description
2026.06.24	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.