Using M2A23 to Control the Dashboard Pointer

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code uses the M2A23 series microcontroller (MCU) to control the pointer of the agricultural machinery dashboard.
BSP Version	M2A23_Series_BSP_CMSIS_V3.00.001
Hardware	NuMaker-M2A23SG V0.1

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 is based on the M2A23 platform and implements precise control of a dedicated two-phase, four-wire stepper motor. It is suitable for pointer drive applications on agricultural machinery dashboards and various other instruments. The system primarily includes two core functions:

Automatic Zeroing Calibration at Startup: When the system starts, the motor first locates the mechanical zero point to establish a physical reference. It then calibrates to the dashboard zero point, ensuring accurate pointer indication.

Periodic Reciprocating Motion: After initialization, the system controls the pointer to perform periodic reciprocating swings between the "zero point" and the "set target position" to verify the driving performance.

1.1 Principle

The system principle mainly includes two parts: mechanical transmission and electrical drive. The mechanical transmission structure is shown in Figure 1-1. The stepper motor rotor drives the pointer shaft to rotate through the gear set.

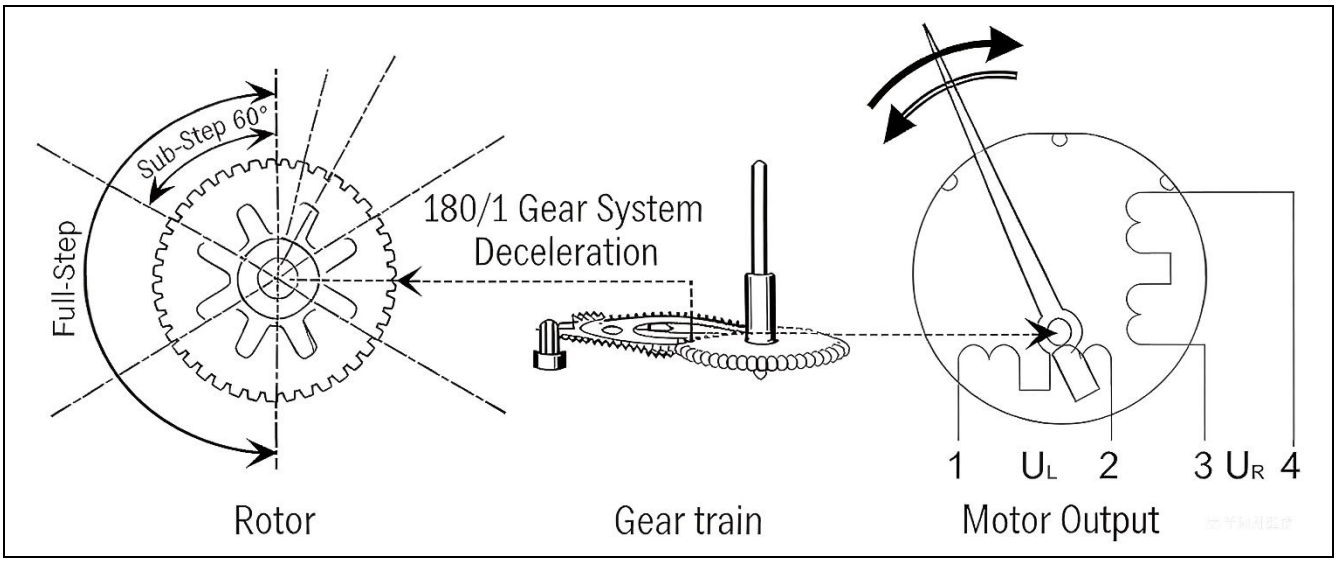

Figure 1-1 Stepper Motor Mechanical Transmission System

- **Reduction Ratio:** 180:1. When the rotor rotates 180°, the output shaft (pointer) only rotates 1°. The high reduction ratio design helps improve the positioning accuracy and holding torque of the pointer.
- **Step Angle Definition:**

  - Sub-Step: Rotor rotates 60°.

  - Full Step: Rotor rotates 180°

The electrical drive timing sequence is shown in Figure 1-2. The motor is driven using a six-step commutation method. By changing the output sequence of the GPIO pins, the rotation direction of the motor can be controlled.

Forward sequence: 1 → 2 → 3 → 4 → 5 → 6

Reverse sequence: 6 → 5 → 4 → 3 → 2 → 1

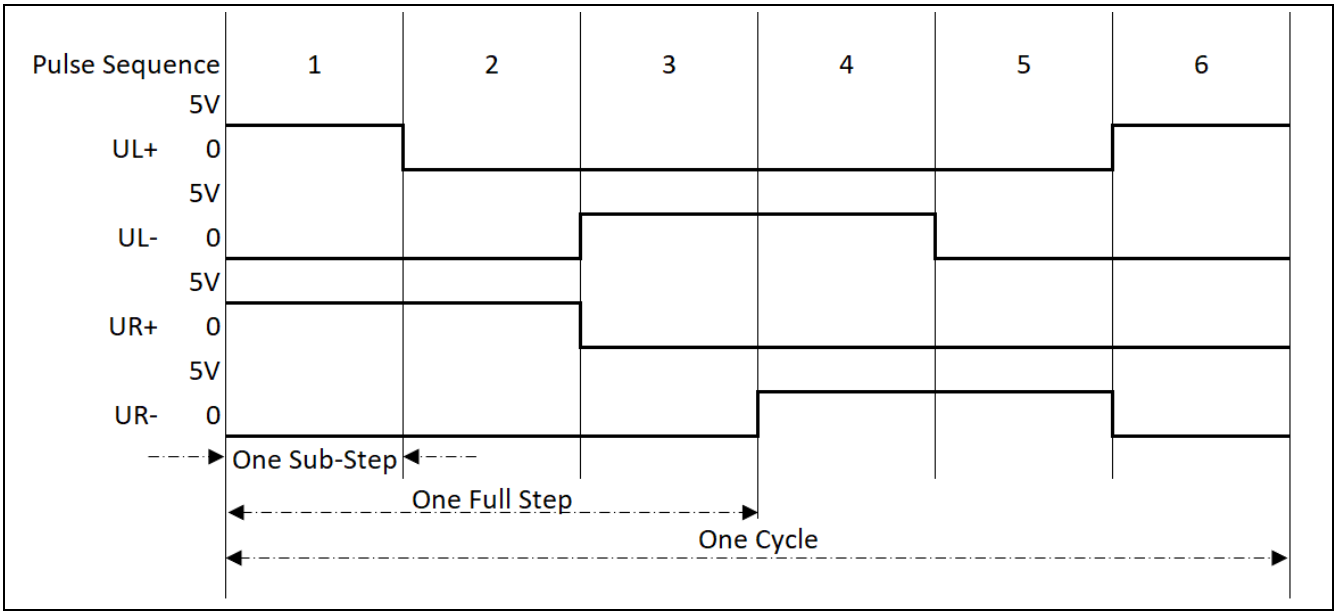

Figure 1-2 Stepper Motor Control Sequence

This example uses GPIO and TIMER to work together to control the stepper motor.

GPIO: Connects to the two-phase bipolar winding of the motor, driving the rotor's step motion by outputting a specific voltage level sequence.

TIMER: Provides a precise time base for controlling the frequency of step pulses, thereby adjusting the motor's rotation speed.

1.2 Demo Result

Open the UART tool on the PC and connect UART0 to print debug information, as shown in Figure 1-3.

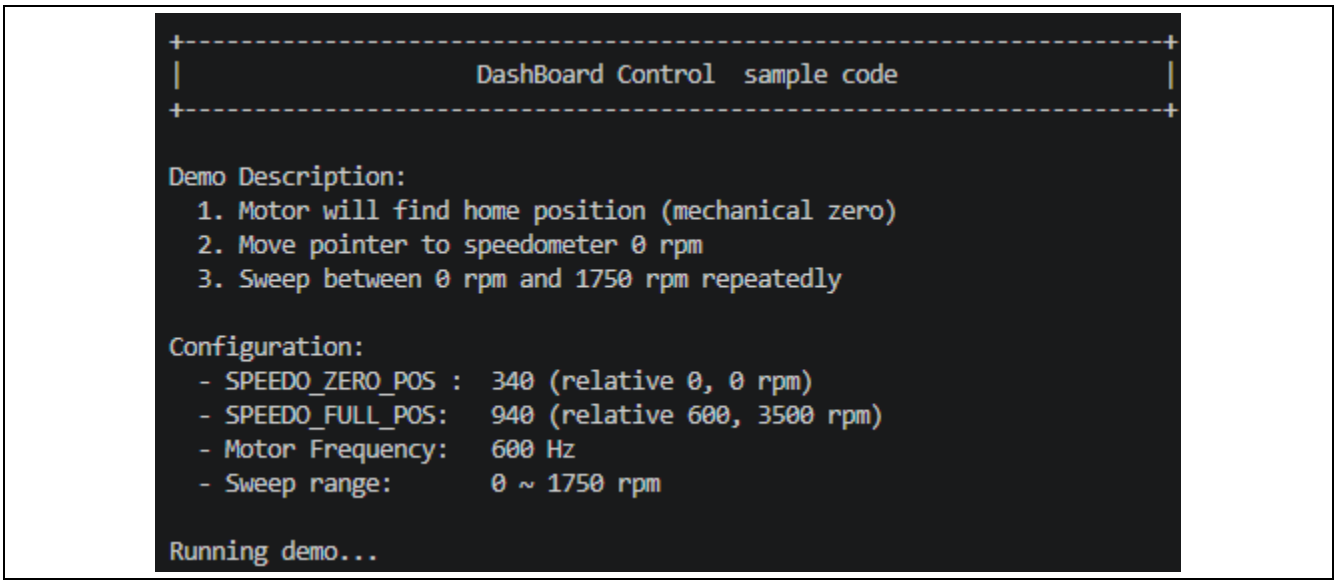

Figure 1-3 Execution Results

2. Code Description

The code is located in *main.c*.

Macro Definition

Define the motor control pins, maximum steps, the zero point and end point of the speedometer, motor rotation speed, etc.

```
/*******************************************************************************
 * Motor Pin Definitions
 * GPIO pins used for stepper motor coil control
 ******************************************************************************/
#define UA_POS PB5    // Motor A phase positive (GPIO PB5)
#define UA_NEG PB4    // Motor A phase negative (GPIO PB4)
#define UB_POS PB3    // Motor B phase positive (GPIO PB3)
#define UB_NEG PB2    // Motor B phase negative (GPIO PB2)

/*******************************************************************************
 * Motor Position Definitions
 * All positions are absolute steps from mechanical zero point
 ******************************************************************************/
#define STEP_MAX 1080             // Maximum steps for homing (mechanical zero)
#define SPEEDO_ZERO_POS 340       // Speedometer zero position (absolute, 0 rpm reference)
#define SPEEDO_FULL_POS 940       // Speedometer full scale position (absolute, 3500 rpm)

/*******************************************************************************
 * Motor Control Configuration
 ******************************************************************************/
#define MOTOR_FREQ 600            // Timer interrupt frequency (Hz)
                                  // Controls motor stepping speed
                                  // 600 Hz = 600 steps/second max
#define FULL_STEP_COUNT 6         // Number of steps in one full electrical cycle

#define SPEED_MAX 3500            // Maximum speed on speedometer
                                  // Range: 0 ~ 3500 rpm
```

Motor Control Parameters

Six-step commutation sequence, state machine, motor steps, pointer position, direction, etc.

```
const MotorStep full_step_sequence[] =
{
    {1, 0, 1, 0},  // Step 1: A +5V, B +5V
    {0, 0, 1, 0},  // Step 2: A 0V,  B +5V
    {0, 1, 0, 0},  // Step 3: A -5V, B 0V
    {0, 1, 0, 1},  // Step 4: A -5V, B +5V
    {0, 0, 0, 1},  // Step 5: A 0V,  B -5V
    {1, 0, 0, 0}   // Step 6: A +5V, B 0V
};

......

volatile uint8_t machine_state = MOTOR_STATE_IDLE;    // Current motor state
volatile uint32_t total_step = 0;                     // Remaining steps to move
volatile uint32_t delay = 0;                          // Software delay counter
```

```
......

volatile int16_t speedo_ptr_pos = 0;   // Speedometer pointer position (0 to 600)
volatile uint8_t direction = MOTOR_DIR_FORWARD;  // Current motor direction
```

System Initialization

Enable the timer clock and set the motor control pins.

```
void SYS_Init(void)
{
    ......
    /* Enable TIMER module clock */
    CLK_EnableModuleClock(TMR0_MODULE);

    /* Select TIMER clock source */
    CLK_SetModuleClock(TMR0_MODULE, CLK_CLKSEL1_TMR0SEL_PCLK0, 0);

    ......

    /* Set multi-function pins for UART0 RXD and TXD */
    GPIO_SetMode(PB, BIT5, GPIO_MODE_OUTPUT);
    GPIO_SetMode(PB, BIT4, GPIO_MODE_OUTPUT);
    GPIO_SetMode(PB, BIT3, GPIO_MODE_OUTPUT);
    GPIO_SetMode(PB, BIT2, GPIO_MODE_OUTPUT);
}
```

Motor Initialization

Initialize the motor output and related global variables.

```
void Motor_Init(void)
{
    // Disable all motor coils (set all pins to LOW)
    UA_POS = 0;
    UA_NEG = 0;
    UB_POS = 0;
    UB_NEG = 0;

    // Reset step counter (no pending movement)
    total_step = 0;

    // Set default motor direction
    direction = MOTOR_DIR_FORWARD;

    // Set motor state to idle
    machine_state = MOTOR_STATE_IDLE;

    // Reset speedometer pointer position to zero (0 rpm)
    speedo_ptr_pos = 0;
}
```

Motor Single Step Drive

This function performs single-step control of the stepper motor. It drives the motor using a six-step lookup table method.

```
void Motor_Step(uint8_t direction)
{
    static uint32_t current_phase = 0;      // Internal phase state tracking
    static uint8_t last_direction = MOTOR_DIR_FORWARD;    // Track last direction
                                                          // for smooth reversal

    // Handle direction change - adjust phase to avoid repetition
    if (direction != last_direction)
    {
        // Adjust phase index for smooth direction transition
        if (direction == MOTOR_DIR_FORWARD) // Switch to Forward
        {
            current_phase = (current_phase + 2) % FULL_STEP_COUNT;
        }
        else  // Switch to Reverse
        {
            current_phase = (current_phase + 4) % FULL_STEP_COUNT;
        }

        last_direction = direction;
    }

    // Directly assign values from full_step_sequence to GPIO pins
    UA_POS = full_step_sequence[current_phase].A1;
    UA_NEG = full_step_sequence[current_phase].A2;
    UB_POS = full_step_sequence[current_phase].B1;
    UB_NEG = full_step_sequence[current_phase].B2;

    // Update phase for next call (AFTER)
    if (direction == MOTOR_DIR_FORWARD) // Forward
    {
        current_phase = (current_phase + 1) % FULL_STEP_COUNT;
    }
    else  // Reverse
    {
        current_phase = (current_phase - 1 + FULL_STEP_COUNT) % FULL_STEP_COUNT;
    }
}
```

Motor Start & Stop Control

The motor's phase switching is automatically handled by the timer interrupt. These two functions control the motor's start and stop by modifying shared variables to decouple the main program from the interrupt.

```
void Motor_Start(uint8_t dir, uint32_t steps)
{
    if (steps == 0)
        return;  // No movement needed

    direction = dir;
    machine_state = (dir == MOTOR_DIR_FORWARD) ? MOTOR_STATE_FORWARD :
                                                 MOTOR_STATE_REVERSE;
    total_step = steps;
}
/**
```

```
 * @brief Stop motor immediately
 */
void Motor_Stop(void)
{
    total_step = 0;
    machine_state = MOTOR_STATE_IDLE;

    UA_POS = 0;
    UA_NEG = 0;
    UB_POS = 0;
    UB_NEG = 0;
}
```

Speedometer Pointer Zero Calibration

To ensure the accuracy of the pointer indication, the system must perform an automatic zero calibration process when starting up.

- Find the Mechanical Zero Point: Rotate the motor in reverse to the mechanical zero position.
- Calibrate the Speedometer Zero Point: Rotate forward from the mechanical zero position to the speedometer zero point.

```
void Speedo_ZeroCalibration(void)
{
    // Step 1: Reverse to mechanical zero point (Home Position)
    direction = MOTOR_DIR_REVERSE;
    total_step = STEP_MAX;

    while (total_step > 0); // Wait until homing complete (step counts down to 0)

    // Step 2: Forward to speedometer working zero position (0 RPM reference)
    direction = MOTOR_DIR_FORWARD;
    total_step = SPEEDO_ZERO_POS;

    while (total_step);

    // Step 3: Reset relative position counter to 0
    speedo_ptr_pos = 0;
}
```

Speedometer Pointer Control

This function maps speed values to motor control parameters (direction and steps) to achieve precise position control of the speedometer pointer.

```
uint8_t Speedo_SetSpeed(uint32_t speed)
{
    uint32_t target_position;
    int32_t steps_to_move;
    uint8_t dir;

    // Validate speed range
    if (speed > SPEED_MAX)
    {
```

```
        return 2;  // Invalid speed value
    }

    // Convert speed to speedometer pointer position
    // Linear mapping: speed 0~3500 → position 0~600
    if (speed > SPEED_MAX)
        speed = SPEED_MAX;

    target_position = (speed * (SPEEDO_FULL_POS - SPEEDO_ZERO_POS)) / SPEED_MAX;

    // Validate position range
    if (target_position < 0 || target_position > (SPEEDO_FULL_POS - SPEEDO_ZERO_POS))
    {
        return 2;  // Invalid position
    }

    // Calculate steps needed to reach target speed
    if (target_position > speedo_ptr_pos)
    {
        steps_to_move = target_position - speedo_ptr_pos;
        dir = MOTOR_DIR_FORWARD;   // Move pointer toward higher speed
    }
    else if (target_position < speedo_ptr_pos)
    {
        steps_to_move = speedo_ptr_pos - target_position;
        dir = MOTOR_DIR_REVERSE;   // Move pointer toward lower speed
    }
    else
    {
        return 1;  // Already at target speed
    }

    // Start motor movement to update speedometer
    Motor_Start(dir, steps_to_move);

    return 0;  // Success
}
```

Timer Initialization

This function configures and starts Timer0, providing an accurate time base for motor control and software delays. The frequency is determined by the macro definition MOTOR_FREQ.

```
void TIMER0_Init(void)
{
    // Configure Timer0 in periodic mode
    // Interrupt frequency: MOTOR_FREQ Hz (600 Hz = 600 steps/second)
    TIMER_Open(TIMER0, TIMER_PERIODIC_MODE, MOTOR_FREQ);

    // Enable Timer0 interrupt
    TIMER_EnableInt(TIMER0);

    // Enable NVIC interrupt for TMR0
    NVIC_EnableIRQ(TMR0_IRQn);

    // Start Timer0 counting
```

```
    TIMER_Start(TIMER0);
}
```

Timer Interrupt Service Function

This function handles motor stepping, position updates, and software delay counting.

```
void TMR0_IRQHandler(void)
{
    // Check Timer0 interrupt flag
    if (TIMER_GetIntFlag(TIMER0) == 1)
    {
        /* Clear Timer0 time-out interrupt flag */
        TIMER_ClearIntFlag(TIMER0);

        // Execute motor step if remaining steps exist
        if (total_step > 0)
        {
            // Execute one motor step
            Motor_Step(direction);

            // Decrement remaining step count
            total_step--;

            // Update speedometer pointer position
            if (machine_state == MOTOR_STATE_FORWARD)
                speedo_ptr_pos++;      // Forward: pointer moves toward higher speed
            else if (machine_state == MOTOR_STATE_REVERSE)
                speedo_ptr_pos--;      // Reverse: pointer moves toward lower speed
        }
        else
        {
            // Steps complete, set motor state to idle
            machine_state = MOTOR_STATE_IDLE;
        }

        // Decrement delay counter (if delay in progress)
        if (delay > 0)
            delay--;
    }
}
```

Main

This function sequentially performs hardware initialization, motor zero calibration, and enters a loop to demonstrate the reciprocating motion of the pointer.

```
int32_t main(void)
{
    ……

    // Initialize motor variables and pins
    Motor_Init();

    // Initialize Timer0 for motor stepping control
    TIMER0_Init();
```

```
    // Perform speedometer zero calibration (find mechanical zero, then move to 0 rpm)
    Speedo_ZeroCalibration();

    // Main loop - continuously sweep speedometer pointer
    while (1)
    {
        // Short delay at current position (1 second)
        Delay(MOTOR_FREQ * 1);

        // Wait for any pending movement to complete
        while (Motor_IsBusy());

        // Set speedometer to mid speed 1750 rpm
        Speedo_SetSpeed(SPEED_MAX/2);

        // Wait for movement to complete
        while (Motor_IsBusy());

        // Short delay at mid speed position (1 second)
        Delay(MOTOR_FREQ * 1);

        // Wait for any pending movement to complete
        while (Motor_IsBusy());

        // Set speedometer to zero (0 rpm)
        Speedo_SetSpeed(0);

        // Wait for movement to complete
        while (Motor_IsBusy());
    }
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M2A23_Series_BSP_CMSIS_V3.00.001
- IDE version
  - Keil uVision 5.43
  - VS Code 1.115.3

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M2A23SG V0.1
- Hardware Connection
  - Connect MCU pins PB.2, PB.3, PB.5 and PB.4 to the stepper motor pins 1, 2, 3 and 4 respectively.
  - Connect the UART0 TX (PB.13) pin to the PC via a UART-to-USB tool to view the execution results of this example code.

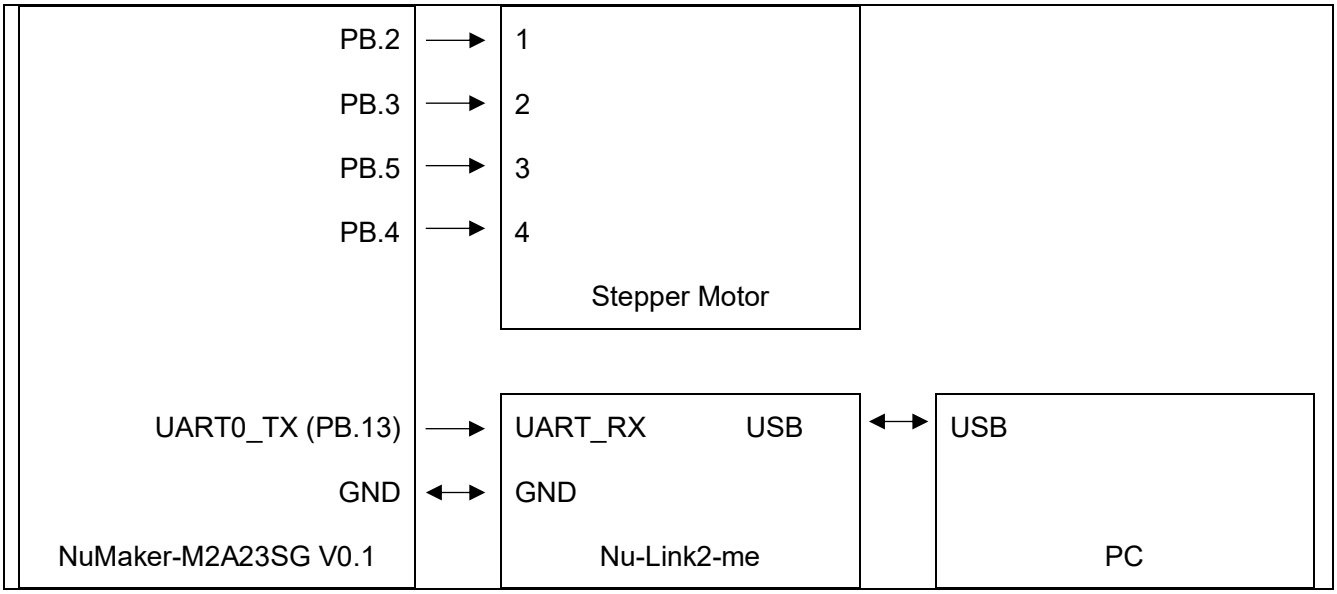

Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M2A23_DashBoard_Control_V1.00	
Library	Sample code header and source 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
SampleCode	
ExampleCode	
Project	Source file of example code

Figure 4-1 Directory Structure

5. Example Code Execution

5.1 Keil Platform

1. Browse the sample code folder as described in the Directory Information section and double-click *M2A23_DashBoard_Control.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

5.2 VS Code Platform

1. Choose “File”
2. Choose “Open Folder”
3. Choose the “VSCode” folder in the “ExampleCode” path
4. Choose “CMSIS”
   - Click “Build solution” to compile
   - Click “Load & Run application” to download the code into memory
5. Enter debug mode.
   - Click “Load & Debug application” to execute the code

6. Revision History

Date	Revision	Description
2026.04.15	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.