LAB 4 – MOTOR CONTROLLER & ENVIROMENTAL MONTIOR

TABLE OF CONTENTS

INTRODUCTION

Congratulations! If you're reading this, it means you've reached the final laboratory exercise in CEG3136.

In Lab 4, you'll develop code to control the speed and direction of the **onboard DC motor** using the slide potentiometer and numeric touchpad. You'll also use the onboard **environmental sensor** to monitor temperature and humidity and display their values. You'll have the opportunity use the **real-time clock** in a bonus activity.

OBJECTIVES

After successfully completing Lab 4, you'll be able to:

1. Use an analog-to-digital converter (ADC) to monitor the level of an external voltage.
2. Generate a PWM output signal to control a DC motor, using a timer in compare mode.
3. Monitor the speed of a motor via a rotary encoder, using a timer in capture mode.
4. Build a closed-loop digital control system using a proportional-integral (PI) controller and tune the system to achieve stability and minimize steady state error.
5. Communicate with an external device using the SPI protocol.

Groups who choose to complete the bonus activity, will be able to:

6. Develop a new device driver for a MCU peripheral, using a similar driver as a model.
7. Configure a real-time clock (RTC) and use it to set and track date and time.

ACADEMIC INTEGRITY

1. Both group members must be involved from start to finish. A student who did not actively participate in the lab work cannot claim credit towards their course grade. Absence for medical or other valid reasons shall be communicated with the course instructor.
2. All lab deliverables must be your own original work. Copying between groups and the use of generative AI are prohibited. This means for example, that you're **not allowed to use Chat GPT** to write any code for you or to write any part of your lab report.

PROJECT SETUP

In the STM32CubeIDE, follow these steps to set up a new project for Lab 4, using your Lab 3 project as a starting point. Detailed instructions and screenshots for steps 1, 3, and 4 can be found in your Lab 0 manual.

1. Create a new STM32 project for the STM32L552ZET6Q MCU. Name your project `CEG3136_Lab4` and create a New Folder in a safe location (e.g. H: or OneDrive). Select the 'Empty' project type.
2. Copy your `Src`, `Inc`, and `Drivers` folders from your Lab 3 project to your Lab 4 project. You can do this within the IDE by dragging and dropping while holding the Ctrl key. If you copy the folders outside of the IDE, you'll need to Refresh your project to make them show up.
3. In your project Properties, find your compiler build settings, add the following preprocessor define:
   ```
   STM32L552xx
   ```
4. Add the following include paths. Keep the default path `../Inc` in the list.
   ```
   ../Drivers/CMSIS/Include
   ../Drivers/CMSIS/Device/ST/STM32L5xx/Include
   ```

PROTIP: REOPENING A PROJECT

If you're working on a lab PC, your workspace will be empty each time you sign in (even if you create it on your own drive). Follow these steps to import an existing project into a workspace:

1. Under the File menu, select Import.
2. Expand the General category and select Existing Projects into Workspace. Click Next.
3. Beside the box for Select Root Directory, click Browse.
4. Navigate to and select or enter the existing project folder, then click Select Folder.
5. Back in the Import dialog box, click Finish.

TROUBLESHOOTING TIPS

Unable to find program for Run or Debug:

1. Right click your project, select Clean Project, then Build Project.
2. Resolve any compile or link errors.

GDB server error or unknown MCU type:

1. Power off your board by pressing the red glowing button.
2. Unplug the USB cable from the host side (i.e. lab PC or laptop).
3. Wait 5 seconds.
4. Restore power and reattach the USB cable.

Undefined reference to main; folder and file icons look different:

1. Rename folders `Inc` and `Src` to `IncX` and `SrcX` respectively.
2. Create new folders `Inc` and `Src`.
3. Drag and drop the header/source files from the old folders to the corresponding new folders.
4. Delete the old folders.

Nonsensical compile errors; code looks totally fine:

Review the code from the top of the file to the first error, checking for an imbalance of curly brackets or round brackets that should be curly brackets.

Your program is not running:

- Suspend and Resume the program a few times without breakpoints:
  - `WaitForSysTick()` is normal. Your program does this every 1ms after executing its tasks.
  - Being stuck in the `InfiniteLoop` means an unhandled exception occurred. Comment out code to avoid enabling interrupts (or registering callbacks). If it still crashes with all interrupts disabled, comment out more code or use breakpoints to isolate the problem.
- Use the SFRs tab to check your peripheral hardware registers:
  - You can save specific registers or fields as favourites.
  - Watch them while you step through your code or click RD to read inputs while suspended.
  - Manually edit the values of fields you suspect are incorrect to see if it resolves the issue.

PERFORMANCE BOOST

In this section, we'll configure a *phase-locked loop* (PLL) to increase the system clock rate, greatly improving CPU performance. We'll also enable optimization in our compiler settings to generate smaller, more efficient assembly code.

SYSTEM CLOCK CONFIGURATION

For the labs so far, we've been using the default System Clock (SYSCLK) frequency of 4.0MHz. We will now increase the frequency to 48.0MHz, using the MCU on-chip PLL as a clock multiplier.

SYSCLK.H

Under the `Inc` folder, add a new header file named `sysclk.h`. Replace its entire contents by copying and pasting the text shown (upper right).

SYSCLK.C

Under the `Src` folder, add a new source file named `sysclk.c`. Replace its entire contents by copying and pasting the mini-text shown (lower right).

sysclk.h

```c
#ifndef SYSCLK_H_
#define SYSCLK_H_

#include "stm32l5xx.h"

// System Clock frequency in Hz
#define SYSCLK_FREQ  48e6

void ConfigureSystemClock(void);

#endif /* SYSCLK_H_ */
```

sysclk.c

```c
// System Clock (SYSCLK) configuration

#include <stdbool.h>
#include "sysclk.h"

static bool done = 0;

void ConfigureSystemClock (void) {
    if (done)
        return;  // SYSCLK already configured

    // Multi-speed oscillator (MSI)
    RCC->CR |= RCC_CR_MSION | RCC_CR_MSIRDY | RCC_CR_MSIRGSEL;
    RCC->CR = (RCC->CR & ~RCC_CR_MSIRANGE_Msk) | 0x6 << RCC_CR_MSIRANGE_Pos;

    // Phase-locked loop (PLL)
    RCC->CR &= ~RCC_CR_PLLON;   // Disable PLL
    RCC->PLLCFGR =   24 << RCC_PLLCFGR_PLLN_Pos     // 2MHz x 24 = 48MHz
                 | 0b01 << RCC_PLLCFGR_PLLSRC_Pos;  // Select MSI clock input
    RCC->CR |= RCC_CR_PLLON;    // Enable PLL
    RCC->PLLCFGR |= RCC_PLLCFGR_PLLREN;  // Enable PLL clock output

    // Add flash read wait states to account for faster SYSCLK
    FLASH->ACR |= 0x2 << FLASH_ACR_LATENCY_Pos;

    // Select PLL as System Clock source
    RCC->CFGR |= 0x3 << RCC_CFGR_SW_Pos;

    done = true;
}
```

DRIVER UPDATES

Two drivers need to be updated to account for the System Clock increase from 4.0MHz to 48.0MHz.

SYSTICK.C

Open your existing `systick.c` file and make the following modifications:

IMPORTANT: You are not permitted to use machine assistance for this task.

1. Add include file `sysclk.h`.
2. Update the symbol definition SYSTICK and replace value `4000` with a new formula to calculate the SysTick period based on the System Clock frequency. Hint: Refer to `sysclk.h` and Exploration Question 2 of your Lab 1 Report to help you create the formula. Note that `SYSTICK` is the actual timer period, while the `SysTick_LOAD` register value is one less (`SYSTICK - 1`).
3. Locate the function `StartSysTick()`. Insert a call to `ConfigureSystemClock()` as the first statement.

I2C.C

Open your existing `i2c.c` file and locate the statement near the end of `I2C_Enable()` that configures the I^{2}C Timing Register (`TIMINGR`). Update the code to look like the following:

```
bus.iface->TIMINGR = 0x????????;  // 100kHz from 48MHz SYSCLK
```

Follow the process below to generate the correct 32-bit hexadecimal value to write to the register.

IMPORTANT: You are not permitted to use machine assistance for this task.

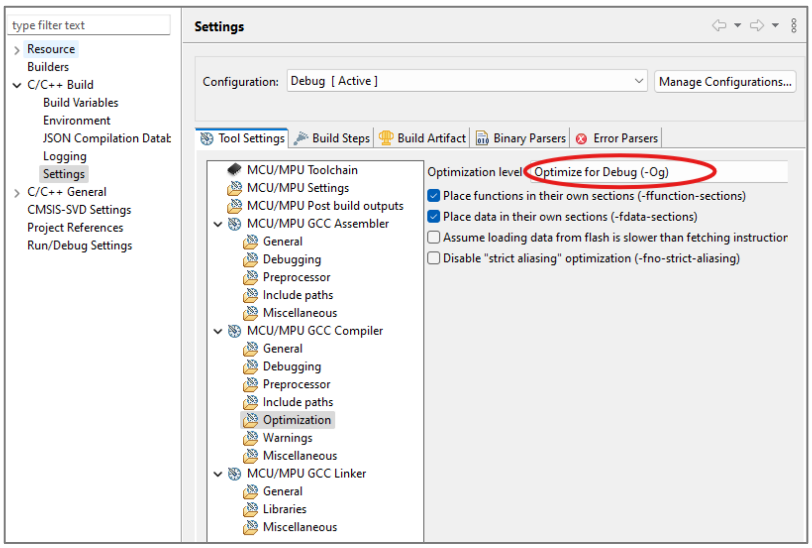
1. Read sections 43.4.10 and 43.9.5 of your MCU Reference Manual.
2. Using that information, determine the value of each field in `I2C_TIMINGR`.
3. Write out the complete register value, grouped in nibbles (4 bits or 1 hexadecimal digit). Note that the nibbles may not be listed in the same order as the reference table!

COMPILER OPTIMIZATION

Follow these steps to enable the C compiler to generate optimized source code:

1. In the Project Explorer, right click your project and open Properties.
2. Navigate to C/C++ Build, Settings, GCC Compiler, Optimization.
3. Set Optimization Level to Optimize for Debug (-Og).
4. Click Apply and Close. If prompted, allow it to Rebuild the Index.

~~For your lab report, screenshot disassembly of a C function of your choice with and without optimization.~~



TESTING THE CHANGES

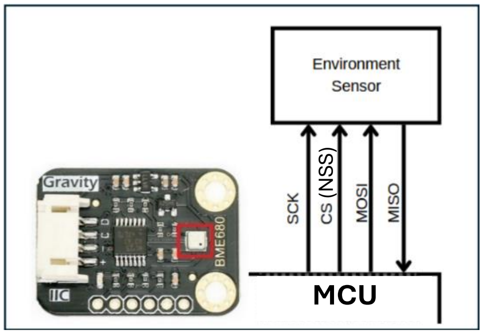
Build your program and correct any additional errors.

Run your program and test all your existing apps to ensure that everything is working as expected. Because the SysTick still runs at 1ms, the LED timing (should remain unchanged (e.g. in the Alarm System app, the blue/green LEDs should toggle at a period of 1 second each in ARMED state).

Update your Debug Configuration for the new System Clock frequency of 48.0MHz. Confirm that `printf()` is working in Debug mode.

ENVIRONMENTAL MONITOR

In this section, we'll use Serial Peripheral Interface (SPI) to obtain the temperature and humidity from the on-board Bosch BME680 environment sensor module.



SPI DRIVER

We'll first build a device driver for the MCU SPI controller. Like the I[2]C driver added in Lab 2, our SPI driver uses an object-oriented design, a linked-list request queue, and cooperative multitasking.

SPI.H

Under the `Inc` folder, add a new header file named `spi.h`. Replace its entire contents by copying and pasting the mini-text shown (below).

Compare `spi.h` to `i2c.h` and note the similarities and differences.

SPI.C

Under the `Src` folder, add a new source file named `spi.c`. Replace its entire contents by copying and pasting the mini-text shown (right).

Complete the highlighted parts of the code, referring to your Lab User's Guide, MCU Reference Manual (9.8.19–21), MCU Datasheet (Table 22), and `i2c.c`. Note that the **NSS pin is a regular GPIO output**, not an alternate function with the SPI controller.

spi.h

```c
#ifndef SPI_H_
#define SPI_H_

#include <stdbool.h>
#include "stm32l5xx.h"
#include "gpio.h"

// SPI bus connection
typedef struct {
    SPI_TypeDef *iface;    // Interface for SPI1-SPI3
    Pin_t        pinSCLK;  // MCU pin for SCLK/SCK
    Pin_t        pinMISO;  // MCU pin for MISO/SO
    Pin_t        pinMOSI;  // MCU pin for MOSI/SI
    Pin_t        pinNSS;   // MCU pin for NSS/CSB
} SPI_Bus_t;

extern SPI_Bus_t EnvSPI;  // SPI bus for Environmental Sensor

typedef enum {RX=1, TX=0} Direction_t;

// SPI transfer record
typedef struct SPI_Xfer_t {
    SPI_Bus_t    *bus;     // Pointer to SPI bus structure
    Direction_t   dir;     // Transfer direction
    uint8_t      *data;    // Pointer to data buffer
    int           size;    // Total number of bytes in transfer
    bool          last;    // Last transfer in combined sequence
    volatile bool busy;    // Busy indicator (queued or in progress)
    struct SPI_Xfer_t *next;  // Pointer to next transfer in queue
} SPI_Xfer_t;

void SPI_Enable(SPI_Bus_t bus);   // Enable SPI bus connection
void SPI_Request(SPI_Xfer_t *p);  // Request a new transfer

void ServiceSPIRequests(void);    // Called from main loop

#endif /* SPI_H_ */
```

spi.c

```c
// SPI controller driver

#include <stddef.h>
#include <stdio.h>
#include "spi.h"
#include "gpio.h"
#include "systick.h"

// SPI bus for the Environmental Sensor
SPI_Bus_t EnvSPI = {
    SPI1,          // SPI controller 1
    {GPIO?, ??},   // SCLK pin
    {GPIO?, ??},   // MISO pin
    {GPIO?, ??},   // MOSI pin
    {GPIO?, ??}    // NSS/CSB pin
};

// Pointers to head and tail of the transfer queue
static SPI_Xfer_t *head = NULL;
static SPI_Xfer_t *tail = NULL;

static int n = -1;  // Number of bytes transferred, -1 when idle

// Enable SPI controller and configure associated GPIO pins
void SPI_Enable (SPI_Bus_t bus) {
    if (bus.iface->CR1 & SPI_CR1_SPE)
        return;  // Already enabled

    // Enable clock to selected SPI controller
    // See MCU Reference Manual, Table 82
    RCC->APB??? |= bus.iface == SPI1 ? RCC_APB???_SPI1EN : 0;
    RCC->APB??? |= bus.iface == SPI2 ? RCC_APB???_SPI2EN :
                   bus.iface == SPI3 ? RCC_APB???_SPI3EN : 0;

    // Enable clocks to GPIO ports containing SPI pins
    // ...

    // Select alternate function as SPI (SCLK, MICO, MOSI only)
    // See MCU Datasheet, Table 22
    // ...

    // Alternate function mode (SCLK, MISO, MOSI only)
    // ...

    // NSS pin: GP output, HIGH state (inactive)
    // ...

    // Configure SPI peripheral
    bus.iface->CR1 &= ~SPI_CR1_SPE;
    bus.iface->CR1 = SPI_CR1_MSTR | (5<<3) | SPI_CR1_SSM | SPI_CR1_SSI;
    bus.iface->CR2 = SPI_CR2_FRXTH | (8 - 1) << SPI_CR2_DS_Pos;
    bus.iface->CR1 |= SPI_CR1_SPE;
}

// Add a transfer request to the queue
void SPI_Request (SPI_Xfer_t *p) {
    if (head == NULL)
        head = p;        // Add to empty queue
    else
        tail->next = p;  // Add to tail of non-empty queue
    tail = p;
    p->next = NULL;
    p->busy = true;      // Mark transfer as in-progress
}

// Polling implementation, called from main loop every tick
void ServiceSPIRequests (void) {
    if (head == NULL)
        return;  // Nothing to do right now

    SPI_Xfer_t *p = head;
    SPI_TypeDef *SPI = p->bus->iface;
    volatile uint8_t *DR = (volatile uint8_t *)&SPI->DR;  // Workaround

    if (n == -1) {
        // Begin a new transfer
        n = 0;
        GPIO_Output(p->bus->pinNSS, LOW);  // Assert select
        if (p->dir == RX) {
            *DR = 0;  // Dummy transmit
        }
    }
    else if (n < p->size) {
        if (p->dir == TX && SPI->SR & SPI_SR_TXE) {
            if (n > 0)
                *DR;  // Dummy receive
            // Copy transmit data from memory buffer to hardware buffer
            *DR = p->data[n++];
        }
        if (p->dir == RX && SPI->SR & SPI_SR_RXNE) {
            // Copy receive data from hardware buffer to memory buffer
            p->data[n++] = *DR;
            if (n < p->size)
                *DR = 0;  // Dummy transmit
        }
    }
    else {
        // Remove transfer from head of queue
        head = p->next;
        p->next = NULL;

        p->busy = 0;  // Mark transfer as complete
        n = -1;       // Prepare for next transfer

        if (p->dir == TX)
            // Drain the receive data buffer
            while (SPI->SR & SPI_SR_RXNE)
                *DR;  // Dummy receive

        if (p->last)
            GPIO_Output(p->bus->pinNSS, HIGH);     // De-assert select
    }
}
```

ENVIRO APP

In this section, you'll build an app that uses our new SPI driver to read raw data from the environmental sensor module, combines it to determine the ambient temperature and humidity—a process known as *sensor fusion*—and print the information on the alphanumeric display.

Throughout this section, you'll need to refer to the Environmental Sensor Datasheet. This document was explored in detail during Tutorial 9 in preparation for this lab.

ENVIRO.H

enviro.h

```
#ifndef ENVIRO_H_
#define ENVIRO_H_

void Init_Enviro(void);
void Task_Enviro(void);

#endif /* ENVIRO_H_ */
```

Under the `Inc` folder, add a new header file named `enviro.h`. Replace its entire contents by manually typing in the text shown.

DISPLAY.H

Open your existing `display.h` file. Add `ENVIRO` to the display page enumeration.

ENVIRO.C

Under the `Src` folder, add a new source file named `enviro.c`. Replace its entire contents by copying and pasting the skeleton code in the mini-text shown (right).

Complete the highlighted sections of code by referring to the datasheet. Hint: For burst reads of multiple bytes, always specify the lowest address, whether than be LSB or MSB. **Do not use generative AI.**

enviro.c Updated

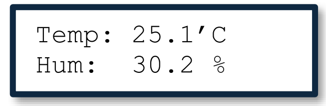
```
// Environmental monitor app

#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include "enviro.h"
#include "spi.h"
#include "display.h"
#include "touchpad.h"
#include "systick.h"

static enum {WAIT_INIT, GET_PARAMS, WAIT_PARAMS,
             TRIGGER_MEAS, WAIT_STATUS, MEAS_READY } state;

#define READ  0x80

// ----------------------------------------------------------
// SPI Transfers
// ----------------------------------------------------------

// Environmental sensor write
typedef struct {
    uint8_t   ctrl;  // Write command (RW=0) and 7-bit address
    uint8_t   data;  // Write data byte
} EnvWrite_t;

////////////////////////////////
// Initialization

// Select Page 1 (addr 0x00..0x7F)
static const EnvWrite_t txPage1 = {0x73, 0x10};
static SPI_Xfer_t Page1 = {&EnvSPI, TX, (void *)&txPage1, 2, 1};

// Select Page 0 (addr 0x80..0xFF)
static const EnvWrite_t txPage0 = {0x73, 0x00};
static SPI_Xfer_t Page0 = {&EnvSPI, TX, (void *)&txPage0, 2, 1};

// Reset sensor
// Refer to datasheet 5.3.1.5 and Table 20
static const EnvWrite_t txResetSensor = {0x60, 0xB6};  // Page 1
static SPI_Xfer_t ResetSensor = {&EnvSPI, TX, (void *)&txResetSensor, 2, 1};

// Read ID
// Refer to datasheet 5.3.1.6 and Table 20
static uint8_t rxId[1];
static const EnvWrite_t txIdAddr = {0x??|READ};  // Page 1
static SPI_Xfer_t ReadId1 = {&EnvSPI, TX, (void *)&txIdAddr, 1, 0};
static SPI_Xfer_t ReadId2 = {&EnvSPI, RX, (void *)&rxId[0], 1, 1};

// Configure oversampling (temperature and humidity)
// Refer to datasheet 3.2.1 and Table 20
static const EnvWrite_t txOvsp[2] = {{0x??, 0x??}, {0x??, 0x??}};  // Page 1
static SPI_Xfer_t SetOvsp = {&EnvSPI, TX, (void *)&txOvsp[0], 4, 1};

////////////////////////////////
// Get Calibration Parameters (temperature)
// Refer to datasheet Table 11

static uint16_t par_t[3];

static const EnvWrite_t txParTAddr1 = {0x??|READ};  // Page 0
static SPI_Xfer_t ParT1 = {&EnvSPI, TX, (void *)&txParTAddr1, 1, 0};
static SPI_Xfer_t ParT2 = {&EnvSPI, RX, (void *)&par_t[0], 2, 1};

static const EnvWrite_t txParTAddr2 = {0x??|READ};  // Page 0
static SPI_Xfer_t ParT3 = {&EnvSPI, TX, (void *)&txParTAddr2, 1, 0};
static SPI_Xfer_t ParT4 = {&EnvSPI, RX, (void *)&par_t[1], 3, 1};

////////////////////////////////
// Get Calibration Parameters (humidity)
// Refer to datasheet Table 13

static uint16_t par_h[8];

static uint8_t rx_buffer_hum[9];
static const EnvWrite_t txParHAddr = {0x??|READ};  // Page 0
static SPI_Xfer_t ParH1 = {&EnvSPI, TX, (void *)&txParHAddr, 1, 0};
static SPI_Xfer_t ParH2 = {&EnvSPI, RX, (void *)&rx_buffer_hum[0], 9, 1};

void ProcessHumidityParameters (void) {
    // Byte 1 split and combined with bytes 0 and 2
    // Refer to datasheet Table 13
    par_h[0] = (rx_buffer_hum[2] << 4) | (rx_buffer_hum[1] & 0x0F);
    par_h[1] = (rx_buffer_hum[0] << 4) | ((rx_buffer_hum[1] & 0xF0) >> 4);

    // Remaining bytes assigned 1:1
    for (uint8_t i=0; i<5; i++) {
        par_h[2+i] = rx_buffer_hum[3+i];
    }
}

////////////////////////////////
// Measurement

// Trigger measurement
// Refer to datasheet 3.2.1, step 8 (combine with existing settings!)
static const EnvWrite_t txTrigMeasAddr = {0x??, 0x41};  // Page 1
static SPI_Xfer_t TrigMeas = {&EnvSPI, TX, (void *)&txTrigMeasAddr, 2, 1};

// Check Status
// Refer to datasheet 5.3.5.1 and Table 20
static uint8_t status;
static const EnvWrite_t txStatusAddr = {0x??|READ};  // Page 1
static SPI_Xfer_t Status1 = {&EnvSPI, TX, (void *)&txStatusAddr, 1, 0};
static SPI_Xfer_t Status2 = {&EnvSPI, RX, (void *)&status, 1, 1};

// Read Humidity
// Refer to datasheet 5.3.4.3 and Table 20
static uint8_t humidityData[2];
static const EnvWrite_t txHumAddr = {0x??|READ};  // Page 1
static SPI_Xfer_t Hum1 = {&EnvSPI, TX, (void *)&txHumAddr, 1, 0};
static SPI_Xfer_t Hum2 = {&EnvSPI, RX, (void *)&humidityData, 2, 1};

// Read Temperature
// Refer to datasheet 5.3.4.2 and Table 20
static uint8_t tempData[3];
static const EnvWrite_t txTempAddr = {0x??|READ};  // Page 1
static SPI_Xfer_t Temp1 = {&EnvSPI, TX, (void *)&txTempAddr, 1, 0};
static SPI_Xfer_t Temp2 = {&EnvSPI, RX, (void *)&tempData, 3, 1};

// ----------------------------------------------------------
// Calculations and display
// ----------------------------------------------------------

static double CalcTemperature() {
    uint32_t temp_adc;
    double var1, var2, t_fine, temp_comp;

    // Refer to datasheet 3.3.1 and Table 11
    temp_adc = ...;

    ...

    return temp_comp;
}

static double CalcHumidity(double temp_comp) {
    uint16_t hum_adc;
    double var1, var2, var3, var4, hum_comp;

    // Refer to datasheet 3.3.3 and Table 13
    hum_adc = ...;

    ...

    return hum_comp;
}

static void ProcessEnvData (void) {
    double temperature;
    double humidity;
    temperature = CalcTemperature();
    humidity = CalcHumidity(temperature);

    // Display temperature and relative humidity to 1 decimal place with units
    ...
}

// ----------------------------------------------------------
// Initialization procedure
// ----------------------------------------------------------

void Init_Enviro (void) {

#if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
    // Enable access to Floating Point Unit (FPU)
    SCB->CPACR |= 0b11 << 10*2 | 0b11 << 11*2;
#endif

    SPI_Enable(EnvSPI);

    //Initialize Humidity/Temperature Sensor
    SPI_Request(&Page0);
    SPI_Request(&ResetSensor);
    SPI_Request(&ReadId1);
    SPI_Request(&ReadId2);
    SPI_Request(&SetOvsp);
    state = WAIT_INIT;
}

// ----------------------------------------------------------
// Task state machine
// ----------------------------------------------------------

void Task_Enviro (void) {
    switch (state) {

    case WAIT_INIT:
        // Wait for initialization to complete
        if (!SetOvsp.busy) {
            printf("Read ID: %x\n",rxId[0]);
            if (rxId[0] != 0x61) {
                printf("ERROR: Read ID incorrect!\n");
            }
            state = GET_PARAMS;
         }
         break;

    case GET_PARAMS:
        // Get calibration parameters for temperature and humidity
        // Refer to SPI transfers above
        SPI_Request(...)
        ...
        state = WAIT_PARAMS;
        break;

    case WAIT_PARAMS:
        // Wait for reads to complete
        if (!ParH2.busy) {
            // Temperature parameters saved in place
            // Humidity parameters require some processing
            ProcessHumidityParameters();
            state = TRIGGER_MEAS;
        }
        break;

    case TRIGGER_MEAS:
        // Trigger humidity and temperature measurement
        SPI_Request(&Page1);
        SPI_Request(&TrigMeas);
        SPI_Request(&Status1);
        SPI_Request(&Status2);
        state = WAIT_STATUS;
        break;

    case WAIT_STATUS:
        // Wait for status to show complete
        if (!(Status2.busy)) {
            if (status & 0x80) {
                // Read humidity and temperature
                // Refer to SPI transfers above
                SPI_Request(...)
                ...
                state = MEAS_READY;
                break;
            }
            else {
                // Read status again
                SPI_Request(...)
                ...
            }
        }
        break;

    case MEAS_READY:
        // Wait for reads to complete
        if (!Temp2.busy) {
            ProcessEnvData();      // Calculate temperature and humidity
            state = TRIGGER_MEAS;  // Start the next measurement
        }
        break;
    }
}
```

MAIN.C

Open your existing `main.c` file. Add include directives for `spi.h` and `enviro.h`. In the Housekeeping section, add a call to `ServiceSPIRequests()` immediately after the I^{2}C equivalent. Also add calls to `Init_Enviro()` and `Task_Enviro()` in the appropriate places.

USER INTERFACE

DISPLAY FORMAT



Sample Enviro app display

The figure to the right shows a sample format for the Enviro app display of temperature and relative humidity, using floating point numbers.

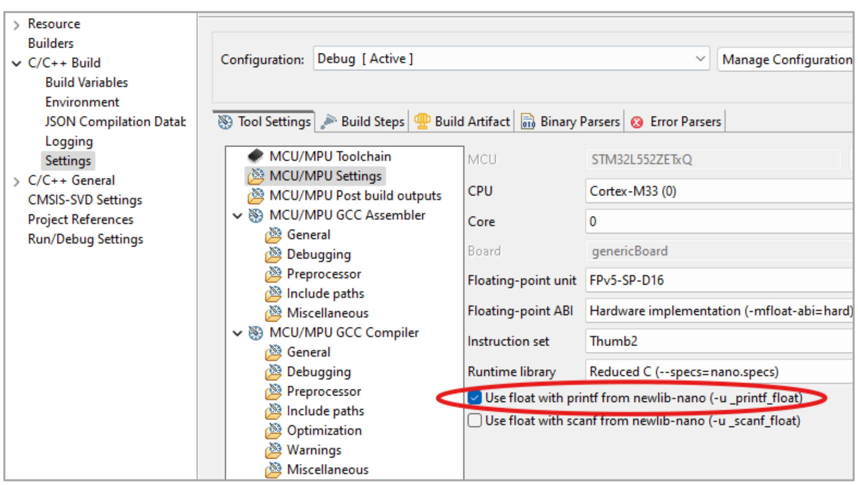
Note: Enviro app does not need to process any Touchpad input.

FORMATTING FLOATING-POINT VALUES

If you'd like to use `%f` floating-point format specifiers in `printf()` and `DisplayPrint()`, you'll first need to change a setting for your project, as shown in the screenshot below:

1. In Project Explorer, right click your project and select Preferences.
2. Under C/C++ Build, open Settings.
3. Under MCU/CPU Settings, check the box to **Use float with printf from newlib-nano**.
4. Click Apply and Close. If asked, confirm rebuilding the index.

PROTIP: The format specifier `%n.mf` (where `n` and `m` are integers) guarantees a fixed spacing/digits on the left/right side of the decimal point respectively. For example, `%2.2f` will format `9.2` as " 9.20" (one leading space) and `17.3333` as "17.33". This is quite handy when you want the field width and precision to stay consistent over a range of values.

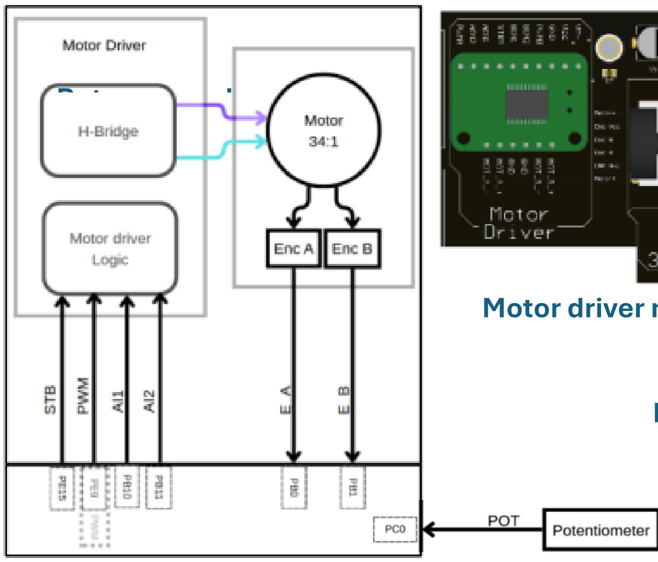


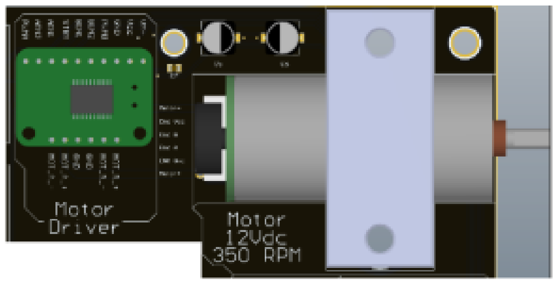
Required project setting for printing floating point numbers

MOTOR CONTROLLER

In the section, we'll build an app to control the on-board DC motor via its driver IC with a PWM signal generated by a timer. We'll use an ADC to measure the position of the on-board slider potentiometer, acting as a speed control, and GPIO interrupts to measure the speed from the motor's built-in rotary encoders.

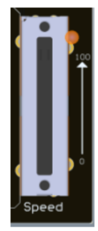
MCU pin connections for each component are shown in the diagram below:





Motor driver module and motor unit

Rotary encoders to measure speed



Potentiometer for speed control

ADC DRIVER

We will start by creating a simple driver for our MCU's analog-to-digital converter (ADC) peripheral so we can detect the position of the slider potentiometer.

The driver supports one channel in discontinuous mode, with conversions performed on-demand. However, it could be expanded to support multiple channels and continuous mode conversions.

ADC.H

Under the `Inc` folder, add a new header file named `adc.h`. Replace its entire contents by copying and pasting the mini-text shown (right above).

ADC.C

Under the `Src` folder, add a new source file named `adc.c`. Replace its entire contents by copying and pasting the mini-text shown (right).

adc.h

```
#ifndef ADC_H_
#define ADC_H_

#include "stm32l5xx.h"
#include "gpio.h"

// ADC input structure
typedef struct {
    ADC_TypeDef  *iface;  // ADC peripheral
    int           chan;   // Channel number
    Pin_t         pin;    // Input/output pin
} ADCInput_t;

void ADC_Enable(ADCInput_t ai);
uint32_t ADC_Read(ADCInput_t ai);

#endif /* ADC_H_ */
```

adc.c

```
// ADC driver

#include "adc.h"
#include "gpio.h"

void ADC_Enable (ADCInput_t ai) {
    ADC_TypeDef *ADC = ai.iface;

    // Configure GPIO pin for analog
    GPIO_Enable(ai.pin);
    GPIO_Mode(ai.pin, ANALOG);

    RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;           // Enable ADC clock
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;        // Enable SYSCFG clock
    RCC->CCIPR1 |= 1 << RCC_CCIPR1_ADCSEL_Pos;  // Select PLLADC1CLK

    ADC->CR &= ~ADC_CR_ADEN;     // Disable ADC
    ADC->CR &= ~ADC_CR_DEEPPWD;  // Disable deep power down

    ADC12_COMMON_NS->CCR &= ~ADC_CCR_PRESC;    // Clear prescaler
    ADC12_COMMON_NS->CCR |= ADC_CCR_CKMODE_1;  // Set clock to HCLK/1

    // 1 conversion, selected channel, discontinuous mode
    ADC->SQR1 = 0 << ADC_SQR1_L;
    ADC->SQR1 |= (ai.chan << ADC_SQR1_SQ1_Pos);
    ADC->CFGR |= ADC_CFGR_DISCEN;

    ADC->CR &= ~(ADC_CR_JADSTP | ADC_CR_ADSTP);
    ADC->CR &= ~(ADC_CR_JADSTART | ADC_CR_ADSTART | ADC_CR_ADDIS);
    ADC->CR |= ADC_CR_ADVREGEN;  // Enable ADC voltage regulator
    ADC->CR |= ADC_CR_ADEN;      // Enable ADC
}

// Read ADC
uint32_t ADC_Read(ADCInput_t ai) {
    ADC_TypeDef *ADC = ai.iface;

    ADC->CR |= ADC_CR_ADSTART;   // Start first ADC conversion, software triggered
    while (!(ADC->ISR & ADC_ISR_EOC)) {}  // Wait for conversion to complete

    return ADC1->DR;  // Conversion result
}
```

TIMER DRIVER

In this section, we'll build a driver for the general-purpose timer peripheral provided by the MCU. Our driver supports the following subset of the timer features:

- Configuring the timer period in terms of pre-scaler and automatic reload values.
- *Output Compare* mode to toggle an output or generate a pulse-width modulated (PWM) signal.
- *Input Capture* mode to timestamp rising or falling edge transitions observed on a pin.
- Interrupt callbacks for timer update and capture/compare events.

TIMER.H

Under the `Inc` folder, add a new header file named `timer.h`. Replace its entire contents by copying and pasting the mini-text shown (right).

TIMER.C

Under the `Src` folder, add a new source file named `timer.c`. Replace its entire contents by copying and pasting the mini-text shown (right below).

Complete the highlighted code to configure the GPIO pin associated with the timer capture/compare channel. Hint: Check your SPI and I²C drivers for examples and refer to the `TimerIO_t` structure definition in `timer.h`.

timer.h

```c
#ifndef TIMER_H_
#define TIMER_H_

#include <stdint.h>
#include "stm32l5xx.h"
#include "gpio.h"

// This preprocessor macro converts a Timer register base address
// (TIM1, TIM2, TIM3, etc.) to a zero-based index (0, 1, 2, etc.)
// The address pattern can be identified from the device header file
#define TIMER_NUM(addr) ((int)addr == 0x40012C00 ? 1 : \
                         (int)addr == 0x40013400 ? 8 : \
                        (((int)addr & 0x0FC00) / 0x400 + 2))

// Timer input/output structure
typedef struct {
    TIM_TypeDef  *iface;  // Timer peripheral
    int           chan;   // Capture/compare channel number
    Pin_t         pin;    // Input/output pin
    int           af;     // Alternate function number
} TimerIO_t;

typedef enum {INCAP=1, OUTCMP=0} TimerMode_t;
typedef enum {TIPRI=0, TISEC=1, TOG=0, PWM1=1, PWM2=2} TimerSelect_t;
typedef enum {UP=0, CC1=1, CC2=2, CC3=3, CC4=4} TimerFlag_t;

void TimerEnable(TimerIO_t timer);
void TimerMode(TimerIO_t timer, TimerMode_t mode, TimerSelect_t sel);
void TimerPeriod(TimerIO_t tio, uint16_t psc, uint16_t arr, uint16_t rcr);
void TimerOutput(TimerIO_t timer, uint16_t thresh);
uint16_t TimerInput(TimerIO_t timer);
void TimerCallback(TimerIO_t timer, void (*func)(void), TimerFlag_t flag);
void TimerStart(TimerIO_t timer, TimerMode_t mode);

#endif /* TIMER_H_ */
```

timer.c Updated

```c
// Timer peripheral driver

#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>

#include "timer.h"
#include "gpio.h"
#include "sysclk.h"

// -------------------------------------------------------
// Initialization
// -------------------------------------------------------

// Force an update of timer registers after configuration changes
static void UpdateTimerRegisters (TIM_TypeDef *TIM) {
    TIM->EGR |= TIM_EGR_UG;       // Update registers
    while (!(TIM->SR & 0x1)) {}   // Wait for update interrupt flag
    TIM->SR &= ~TIM_SR_UIF;       // Clear update interrupt flag
}

// Enable the timer peripheral clock
void TimerEnable (TimerIO_t timer) {
    TIM_TypeDef *TIM = timer.iface;

    // Enable timer peripheral clock
    if      (TIM == TIM1) RCC->APB2ENR  |= RCC_APB2ENR_TIM1EN;
    else if (TIM == TIM8) RCC->APB2ENR  |= RCC_APB2ENR_TIM8EN;
    else                  RCC->APB1ENR1 |= 1 << (TIMER_NUM(TIM) - 2);

    // Enable SYSCFG clock
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

    // Enable preload
    TIM->CR1 |= TIM_CR1_ARPE;

    UpdateTimerRegisters(TIM);

    // Configure the GPIO pin using parameters from timer structure
    ...
}

// Set periods for the 3 cascaded counters
void TimerPeriod(TimerIO_t tio, uint16_t psc, uint16_t arr, uint16_t rcr) {
    TIM_TypeDef *TIM = tio.iface;

    TIM->PSC = psc;  // Prescaler register
    TIM->ARR = arr;  // Auto-reload register
    TIM->RCR = rcr;  // Repetition count register
    TIM->CNT = 0;    // Clear counter

    UpdateTimerRegisters(TIM);
}

// Set the operating mode of a timer channel:
// Input Capture (INCAP) or Output Compare (OUTCMP)
// Timer Output: Toggle (TOG), PWM Mode 1 (PWM1), or PWM Mode 2 (PWM2)
// Timer Input: Primary (TIPRI) or Secondary (TISEC) for given channel
void TimerMode (TimerIO_t tio, TimerMode_t mode, TimerSelect_t sel) {
    TIM_TypeDef *TIM = tio.iface;
    volatile uint32_t *CCMR = &TIM->CCMR1 + (tio.chan - 1) / 2;

    if (mode == OUTCMP) {
        // Capture/compare mode register
        // Select PWM Mode 1 or 2, preload enable
        *CCMR |= ( (sel == PWM1 ? 0b0110 : 0b0111) << TIM_CCMR1_OC1M_Pos
                 | TIM_CCMR1_OC1PE ) << (tio.chan - 1) % 2 * 8;

        // Enable capture/compare
        TIM->CCER |= TIM_CCER_CC1E << (tio.chan - 1) * 4;
        TIM->CCER |= TIM_CCER_CC2E;  // Output compare mode

        TIM->BDTR |= TIM_BDTR_MOE;   // Main output enable
    }

    else if (mode == INCAP) {
        // Select timer input pin T1 or T2
        *CCMR |= (sel == TISEC ? 0b10 : 0b01) << TIM_CCMR1_CC1S_Pos
                  << (tio.chan - 1) % 2 * 8;

        // Enable capture/compare
        TIM->CCER |= TIM_CCER_CC1E << (tio.chan - 1) * 4;
    }
}

// Start the timer
void TimerStart (TimerIO_t tio, TimerMode_t mode) {
    TIM_TypeDef *TIM = tio.iface;

    // Toggle timer configuration bits
    TIM->CCER &= ~(TIM_CCER_CC1E << (tio.chan - 1) * 4);
    TIM->CCER |=  (TIM_CCER_CC1E << (tio.chan - 1) * 4);

    TIM->CR1  |= TIM_CR1_CEN;  // Counter enable
}

// -------------------------------------------------------
// Observation and control
// -------------------------------------------------------

// Output Compare mode: Write CCR threshold value
void TimerOutput(TimerIO_t timer, uint16_t thresh) {
    volatile uint32_t *CCR = (&timer.iface->CCR1 + timer.chan - 1);
    *CCR = thresh;
}

// Input Capture mode: Read
uint16_t TimerInput(TimerIO_t timer) {
    volatile uint32_t *CCR = (&timer.iface->CCR1 + timer.chan - 1);
    return *CCR;
}

// -------------------------------------------------------
// Interrupt handling
// -------------------------------------------------------

// Array of callback function pointers
// Timers 1 to 8, UP + 4 CC channels for each
static void (*callbacks[8][5]) (void);

// Register a function to be called when an interrupt occurs
void TimerCallback(TimerIO_t timer, void (*func)(void), TimerFlag_t flag) {
    TIM_TypeDef *TIM = timer.iface;

    int i = TIMER_NUM(TIM);
    int IRQn = i == 1 ? flag == UP ? TIM1_UP_IRQn : TIM1_CC_IRQn :
               i == 8 ? flag == UP ? TIM8_UP_IRQn : TIM8_CC_IRQn :
                                     TIM2_IRQn + (i - 2);

    // Set callback address
    callbacks[i-1][flag] = func;

    // Enable interrupt generation
    TIM->DIER |= (1 << flag);

    UpdateTimerRegisters(TIM);

    // Enable interrupt vector
    NVIC->IPR[IRQn] = 0;
    __COMPILER_BARRIER();
    NVIC->ISER[IRQn / 32] = 1 << (IRQn % 32);
    __COMPILER_BARRIER();
}

// Interrupt handler for all timers
static void TimerIRQHandler (TIM_TypeDef *TIM, int i, int IRQn) {
    // Callback function pointer
    void (*fp)(void);

    // Clear pending IRQ
    NVIC->ICPR[IRQn / 32] = 1 << (IRQn % 32);

    // Detect interrupt flags
    for (int j = 0; j <= 4; j++)
        if (TIM->SR & 1 << j) {
            fp = callbacks[i-1][j];
            if (fp != NULL) {
                UpdateTimerRegisters(TIM);
                fp();  // Invoke callback
            }
        }
}

// Dispatch all Timer IRQs to common handler function
void TIM1_UP_IRQHandler() { TimerIRQHandler(TIM1, 1, TIM1_UP_IRQn); }
void TIM1_CC_IRQHandler() { TimerIRQHandler(TIM1, 1, TIM1_CC_IRQn); }
void TIM2_IRQHandler()    { TimerIRQHandler(TIM2, 2, TIM2_IRQn); }
void TIM3_IRQHandler()    { TimerIRQHandler(TIM3, 3, TIM3_IRQn); }
void TIM4_IRQHandler()    { TimerIRQHandler(TIM4, 4, TIM4_IRQn); }
void TIM5_IRQHandler()    { TimerIRQHandler(TIM5, 5, TIM5_IRQn); }
void TIM6_IRQHandler()    { TimerIRQHandler(TIM6, 6, TIM6_IRQn); }
void TIM7_IRQHandler()    { TimerIRQHandler(TIM7, 7, TIM7_IRQn); }
void TIM8_UP_IRQHandler() { TimerIRQHandler(TIM8, 8, TIM8_UP_IRQn); }
void TIM8_CC_IRQHandler() { TimerIRQHandler(TIM8, 8, TIM8_CC_IRQn); }
```

MOTOR APP

We will now use our ADC and Timer drivers to create an app to control the motor.

MOTOR.H

Under the `Inc` folder, add a new header file named `motor.h`. Replace its entire contents by copying and pasting the text shown (right).

MOTOR.C

Under the `Src` folder, add a new source file named `motor.c`. Replace its entire contents by copying and pasting the skeleton code in the mini-text shown (right).

Read the next few sections of the lab manual before attempting to complete the code.

DISPLAY.H

Open your existing `display.h` file. Add `MOTOR` to the display page enumeration. If necessary, increase the number of display `PAGES`. You can set the default backlight colour for any added pages beyond the initial four in the `dispColor[]` array defined in `display.c`.

MAIN.C

Open your existing `main.c` file. Add an include directive `motor.h`. Also add calls to `Init_Motor()` and `Task_Motor()` in the appropriate places.

Note: Our ADC and Timer drivers do not require any housekeeping tasks to be run in the main loop.

motor.h

```c
#ifndef MOTOR_H_
#define MOTOR_H_

void Init_Motor(void);
void Task_Motor(void);

#endif /* MOTOR_H_ */
```

motor.c Updated

```c
// Motor controller app

#include <stdbool.h>
#include <math.h>
#include <stdio.h>

#include "motor.h"
#include "display.h"
#include "touchpad.h"
#include "gpio.h"
#include "timer.h"
#include "adc.h"
#include "sysclk.h"
#include "touchpad.h"

// GPIO pins
// Refer to Lab User's Guide
static const Pin_t AI1  = {GPIO?, ??};  // Pin P??? -> Motor driver
static const Pin_t AI2  = {GPIO?, ??};  // Pin P??? -> Motor driver
static const Pin_t STBY = {GPIO?, ??};  // Pin P??? -> Motor driver
static const Pin_t EncA = {GPIO?, ??};  // Pin P??? <- Rotary encoder A
static const Pin_t EncB = {GPIO?, ??};  // Pin P??? <- Rotary encoder B

// Analog-to-digital converter for potentiometer
// Refer to Lab User's Guide and MCU datasheet
static const ADCInput_t Pot = {ADC?, ?, {GPIO?, ??}};  // ADC1 Chan1 <- P???

// Timer channels
// Refer to Lab User's Guide and MCU datasheet
static const TimerIO_t Motor = {TIM?, ?, {GPIO?, ??}, ?};  // Timer? Chan? -> P??? AF?

// Timer period
#define PWM_PSC (100 - 1)  // Pre-scaler
#define PWM_ARR (240 - 1)  // Auto-reload
#define PWM_RCR ( 50 - 1)  // Repetition count

#define MAX_SPEED 350.0
static enum { CW=0, CCW=1 } direction = CCW;      // Clockwise or counter-clockwise
static enum { OL=0, CL=1, CLT=2 } loopMode = OL; // Open/closed loop (w/ tuning)

static float rpmScalingFactor;
static float desiredRPM = 0;
static float errorRpm;
static float error = 0;
static float errorRpmSum = 0;
static float errorSum = 0;

// PI controller parameters
// Note: Update defaults for Np and Ni after tuning
// Kp = Np * dp
int   Np = 5;
float dp = 0.1;
// Ki = Ni * di
int   Ni = 5;
float di = 0.001;

uint32_t pulsesA          = 0;
uint32_t pulsesB          = 0;
uint32_t timerCount     = 0;
uint32_t totalPulses    = 0;
uint32_t prevTimerCount = 0;

// Interrupt callback functions
static void CallbackMotor(void);
static void CallbackEncA(void);
static void CallbackEncB(void);

// Change motor direction
// Refer to motor driver datasheet and use your GPIO driver
void MotorDirection(int dir) {
    if (dir == CW) {
        // Clockwise
        ...
    }
    else {
        // Counter-clockwise
        ...
    }
}

// Initialize app
void Init_Motor (void) {

#if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
                        SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));
#endif

    ADC_Enable(Pot);

    // Configure GPIO pins
    // Refer to motor controller datasheet.  Start in short brake, regardless of PWM state.
    // Register callbacks for rotary encoder inputs

    ...

    TimerEnable(Motor);
    TimerPeriod(Motor, PWM_PSC, PWM_ARR, PWM_RCR);
    TimerMode(Motor, OUTCMP, PWM1);
    TimerCallback(Motor, CallbackMotor, UP);
    TimerStart(Motor, OUTCMP);

    MotorDirection(direction);

    // Refer to Lab Manual section on Speed Measurement
    rpmScalingFactor = ...
}

// Execute app
void Task_Motor (void) {

    float motorDrivePercentage = (float)ADC_Read(Pot) / 4096.0;
    desiredRPM = motorDrivePercentage * MAX_SPEED;

    if (timerCount-prevTimerCount > 0) {
        float measuredRPM = (float)totalPulses*rpmScalingFactor;

        if (loopMode == OL) {
            TimerOutput(Motor, (uint16_t) (motorDrivePercentage * (float)(PWM_ARR + 1)));

            // Display status for Open Loop mode
            DisplayPrint(MOTOR, 0, " OL     %3d %%", (int)(motorDrivePercentage*100));
            DisplayPrint(MOTOR, 1, "%cCW     %3d RPM", direction == CCW ? 'C' : ' ', (int)measuredRPM);
        }

        else {
            // Closed Loop control

            float Kp = (float)Np * dp;
            float Ki = (float)Ni * di;

            errorRpm = desiredRPM - measuredRPM;
            errorRpmSum += errorRpm;

            // Scale errorRpm to Timer CCR
            error    = errorRpm    / MAX_SPEED * (float)(PWM_ARR + 1);
            errorSum = errorRpmSum / MAX_SPEED * (float)(PWM_ARR + 1);

            TimerOutput(Motor,round((Kp*error + Ki*errorSum)));

            if (loopMode == CLT) {
                // Display status for Closed Loop mode /w tuning enabled
                ...
            }
            else {
                // Display status for normal Closed Loop mode
                ...
            }
        }

        totalPulses = 0;
        prevTimerCount = timerCount;
    }

    // Touchpad controls
    switch (TouchInput(MOTOR)) {

    // Motor direction: update and apply
    ...

    // Controller operating mode
    ...

    // Controller tuning (must be in CLT mode to take effect)
    ...

    default: break;
    }
}

// Timer 1 update
void CallbackMotor (void) {
    timerCount++;
    totalPulses = pulsesA+pulsesB;
    pulsesA = 0;
    pulsesB = 0;
}

// Rotary encoder A rising edge
void CallbackEncA (void) {
    pulsesA++;
}

// Rotary encoder B rising edge
void CallbackEncB (void) {
    pulsesB++;
}
```

SPEED MEASUREMENT

Assigned at the end of `Init_Motor()` and used in `Task_Motor()`, calculated variable `rpmScalingFactor` converts `totalPulses` (from the rotary encoders) counted during the timer update interval into a rotational speed in revolutions per minute (RPM):

```
measuredRPM = totalPulses * rpmScalingFactor;
```

`rpmScalingFactor` can be expressed as the ratio of the *timer update frequency* (f_{UP}) to the number of *pulses per revolution* (PPR) generated by the two rotary encoders. A factor of 60 converts frequency from a per-second to per-minute time base (i.e. Hz to RPM):

$$rpmScalingFactor = \frac{f_{UP}}{PPR} \times \frac{60\text{s}}{1\text{ min}}$$

- Express f_{UP} by starting with the SYSCLK frequency (see `sysclk.h`) and dividing by the timer pre-scaler, auto-reload period, and repetition count (see constants in `motor.c`). Remember to add 1 to each constant to undo the effect of subtracting 1!
- Express PPR as the pulses per rotation of the rotary encoder multiplied by the gear ratio to the motor drive shaft, then further doubled because we're counting pulses from two encoders. See your Lab User's Guide for these specifications.

Your final code should contain a **documented expression to calculate `rpmScalingFactor`**. To check your work, highlight the hidden value in the box below and copy and paste it into a comment in your code:

Reveal the correct value here: `rpmScalingFactor =` | |
|---|

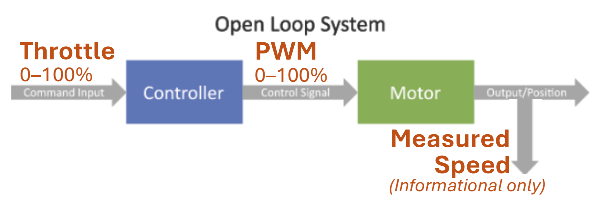
CONTROL SCHEMES

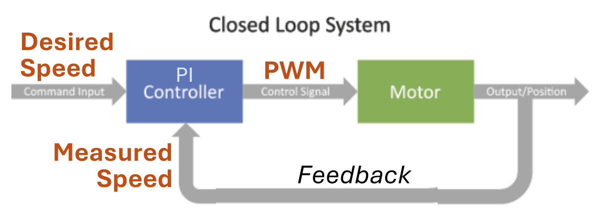
Our Motor app supports two basic control schemes:

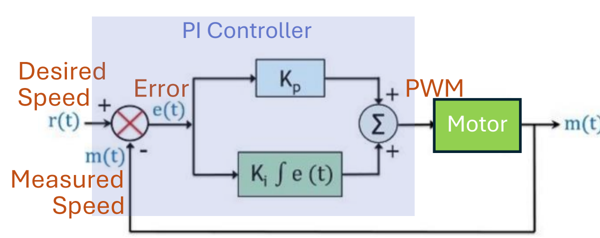
Open Loop—The slide potentiometer acts as a throttle control, which is directly converted to PWM duty cycle between 0% and 100%. The actual motor speed is displayed for informational purposes only.

Closed Loop— Features a *proportional-integral* (PI) controller. The slide potentiometer sets the desired motor speed in RPM. Measured speed is subtracted from desired speed to obtain the speed error. In the proportional path, the most recent error value is scaled by parameter Kp. In the integral path, the sum of all previous error values is scaled by parameter Ki. The outputs of both paths are summed to produce PWM values on an ongoing basis.

Note: To ensure proper motor control, Kp and Ki must be carefully selected. See below for a tuning guide.



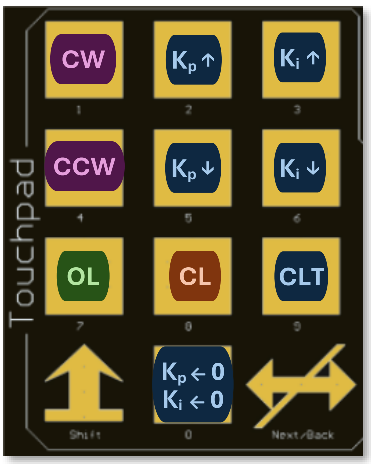




USER INTERFACE

The Motor App includes a user interface (UI) using the on-board display and touchpad:

- Pads 1 & 4 select the motor direction as clockwise (CW) or counterclockwise (CCW). Make sure to apply the new setting after updating the loopMode variable.
- Pads 7–9 select Open Loop (OL), Closed Loop (CL), or Closed Loop Tuning (CLT).
- Pads 2 & 5 and 3 & 6 respectively adjust Kp and Ki up & down by incrementing or decrementing the variables Np and Ni. The variables must not be allowed to decrement below zero.
- Pad 0 immediately clears Kp and Ki to begin the tuning process.
- Adjustments to Kp and Ki are permitted in CLT mode only. Otherwise, those pad presses are ignored.



Touchpad Controls

Sample Display for Each Mode

CONTROLLER TUNING

The following procedure is recommended for tuning the PI controller:

1. Press 9 to enter **Closed Loop Tuning (CLT) mode**.
2. Press 0 to **clear Kp and Ki** to 0. The motor should be stopped.
3. Adjust the target speed to 200 RPM using the slide potentiometer.
   PROTIP: Tuning the motor at its intended operating speed improves controller performance.
4. **Slowly increase Kp**, while listening to the motor sound. Stop when it oscillates periodically.
5. **Reduce Kp to half** (or slightly less than half) of the maximum value reached. The oscillations should stop, but the measured speed will have a significant offset from the target speed.
6. **Slowly increase Ki** to minimize the steady state error (approximately 5 RPM is a good result).
   Note: The measured motor speed may cross the target speed during this process.
7. If necessary, further fine-tune Kp and Ki through experimentation.
8. In your `motor.c` file, update the default values of variables Np and Ni to produce your chosen Kp and Ki values when your program is restarted.

Practice tuning your controller a few times. You'll be asked to show and explain the process to your lab TA during your demo!

BONUS: REAL-TIME CLOCK

In this bonus activity, you're challenged to leverage the MCU real-time clock (RTC) peripheral to create an app that displays the time and date and allows the user to change them using the numeric touchpad.

Because there is no battery on the lab platform, the time and date will be lost whenever the board loses power (glowing button). However, you will need to demonstrate that the RTC will survive MCU reset.

RTC DRIVER

The first step is to build a driver for the real-time clock peripheral.

RTC.H

Under the `Inc` folder, add a new header file named `rtc.h`. Replace its entire contents by copying and pasting the mini-text shown (right above).

RTC.C

Under the `Src` folder, add a new source file named `rtc.c`. Replace its entire contents by copying and pasting the mini-text shown (right).

Complete the code at the highlighted locations, with help from your MCU Reference Manual. The device header file `stm32l552xx.h` is also useful for finding constants for the various register files.

To convert from binary coded decimal (BCD) to ASCII, add the value to character '0'. Subtracting character '0' will convert in the opposite direction. Make sure to do your masks and shifts in the correct order!

rtc.h

```c
#ifndef RTC_H_
#define RTC_H_

void RTC_Enable(void);
void RTC_GetTime(char *time);  // "HH:MM:SS"   ( 9 bytes, including NUL)
void RTC_GetDate(char *date);  // "YYYY-MM-DD" (11 bytes, including NUL)
void RTC_SetTime(const char *time);
void RTC_SetDate(const char *date);

#endif /* RTC_H_ */
```

rtc.c

```c
// Real-time clock (RTC) driver

#include <stdio.h>
#include "stm32l5xx.h"
#include "rtc.h"

// Async/sync pre-scalers (recommended values)
#define PDA (128 - 1)
#define PDS (256 - 1)

// --------------------------------------------------------
// Initialization
// --------------------------------------------------------

void RTC_Enable (void) {
    // Enable clock to PWR block and disable backup domain write protection
    // Refer to MCU RM 9.8.19 and 8.6.1 and device header file
    ?????

    // Enable LSE 32.768kHz, select LSE for RTC clock, and enable RTC clock
    // Refer to MCU RM 9.8.29 and device header file
    ?????

    // Enable APB clock to RTC peripheral
    // Refer to MCU RM 9.8.19 and device header file
    ?????

    // Disable RTC write protection (2 writes)
    // Refer to MCU RM 41.6.10 and 41.3.11, 2nd subsection
    ?????

    // Asynchronous and synchronous pre-scalers
    // Refer to MCU RM 41.3.6 and 41.6.5. Use the PDA and PDS constants.
    ?????
}

// --------------------------------------------------------
// Get time/date from RTC
// --------------------------------------------------------

// Generate time string from RTC, formatted HH:MM:SS (24-hour clock)
// Refer to MCU RM 41.6.1
void RTC_GetTime (char *time) {
    uint32_t tr;
    char ht, hu, mnt, mnu, st, su;

    tr  = RTC->TR;
    ht  = '0' + ((tr & RTC_TR_HT_Msk ) >> RTC_TR_HT_Pos );  // Hour
    hu  = ?????
    ?????

    snprintf(time, 9, "%c%c:%c%c:%c%c", ht, hu, mnt, mnu, st, su);
}

// Generate date string from RTC, formatted YYYY-MM-DD
// Refer to MCU RM 41.6.2
void RTC_GetDate (char *date) {
    uint32_t dr;
    int ym, yc, yt, yu, mt, mu, dt, du;

    dr = RTC->DR;
    ym = '2';
    yc = '0';
    yt = ?????
    ?????

    snprintf(date, ?????);
}

// --------------------------------------------------------
// Set RTC time/date
// --------------------------------------------------------

// Set RTC time from string, formatted HH:MM:SS
void RTC_SetTime (const char *time) {
    uint32_t tr;

    // Enter initialization mode
    // Refer to MCU RM 41.6.4 and 41.3.11, 3rd subsection, items 1-2
    ?????
    while (!(?????)) {}

    tr = 0;
    tr |= (time[0] - '0') << RTC_TR_HT_Pos  & RTC_TR_HT_Msk;   // Hour
    tr |= ?????
    ?????
    RTC->TR = tr;
    RTC->CR &= ~RTC_CR_FMT;  // FMT=0 (24-hour)

    // Return to free-running mode
    // Refer to MCU RM 41.6.4 and 41.3.11, 3rd subsection, item 5
    ?????
}

// Set RTC date from string, formatted YYYY-MM-DD
void RTC_SetDate (const char *date) {
    uint32_t dr;

    // Enter initialization mode
    // Refer to MCU RM 41.6.4 and 41.3.11, 3rd subsection, items 1-2
    ?????

    dr = 0;
    dr |= ?????
    ?????
    RTC->DR = dr;

    // Return to free-running mode
    // Refer to MCU RM 41.6.4 and 41.3.11, 3rd subsection, item 5
    ?????
}
```

CLOCK APP

Using the RTC driver, the Clock application shall:

- Display the RTC time in 24-hour format: HH:MM:SS
- Display the RTC date in ISO format: YYYY-MM-DD
- When pad 1 is pressed, allow the user to set the time in the RTC
- When pad 2 is pressed, allow the user to set the date in the RTC

CLOCK.H

Under the `Inc` folder, add a new header file named `clock.h`. Replace its entire contents by copying and pasting the text shown (right).

clock.h

```c
#ifndef CLOCK_H_
#define CLOCK_H_

void Init_Clock(void);
void Task_Clock(void);

#endif /* CLOCK_H_ */
```

CLOCK.C

Under the `Src` folder, add a new source file named `clock.c`. Replace its entire contents by copying and pasting the skeleton code in the mini-text shown (right).

Complete the highlighted section of code, being careful to account for the format of the time/date string!

clock.c

```c
// Clock app

#include <stddef.h>
#include <stdbool.h>
#include <string.h>
#include "clock.h"
#include "rtc.h"
#include "display.h"
#include "touchpad.h"

static char time[9]  =   "HH:MM:SS";
static char date[11] = "YYYY-MM-DD";

static uint32_t entry;  // Touchpad numeric entry

static enum {SHOW, SETDATE, SETTIME} state;

// Facilitate numeric entry of new time or date
bool TimeDateEntry (char *td) {
    bool done;

    DisplayPrint(CLOCK, 1, entry ? "%u" : "", entry);

    done = TouchEntry(CLOCK, &entry);
    if (entry > 999999)
        // Too long; remove last entered digit
        entry /= 10;
    else if (done && entry >= 100000) {
        // Convert 6 digits to ASCII and populate string
        // char td[8] = "HH:MM:SS"
        td[7] = '0' + entry % 10;  entry /= 10;
        ...
        return true;
    }

    return false;  // Incomplete entry
}

// App initialization
void Init_Clock (void) {
    RTC_Enable();
}

// App execution
void Task_Clock (void) {
    switch (state) {

    case SHOW:
        // Display time and date from RTC
        RTC_GetTime(time);
        RTC_GetDate(date);
        DisplayPrint(CLOCK, 0, time);
        DisplayPrint(CLOCK, 1, date);

    switch (TouchInput(CLOCK)) {
    case 1:  state = SETTIME;  entry = 0;  break;
    case 2:  state = SETDATE;  entry = 0;  break;
    default: break;
    }
    break;

    case SETTIME:
        DisplayPrint(CLOCK, 0, "Set time: HHMMSS");

        if (TimeDateEntry(&time[0])) {
            RTC_SetTime(time);
            state = SHOW;
        }
        break;

    case SETDATE:
        DisplayPrint(CLOCK, 0, "Set date: YYMMDD");

        if (TimeDateEntry(&date[2])) {
            RTC_SetDate(date);
            state = SHOW;
        }
        break;
    }
}
```

DISPLAY.H

Open your existing `display.h` file. Add `CLOCK` to the display page enumeration. If necessary, increase the number of display `PAGES`. You can set the default backlight colour for any added pages beyond the initial four in the `dispColor[]` array defined in `display.c`.

MAIN.C

Open your existing `main.c` file. Add an include directive for `clock.h`. Also add calls to `Init_Clock()` and `Task_Clock()` in the appropriate places.

Note: Our RTC driver does not require a housekeeping task to be run in the main loop.

DELIVERABLES

The following sections detail the key deliverables for this laboratory exercise.

LAB DEMO

After you've completed and tested your project, you will demonstrate your final program to your lab TA.

Grading criteria:

- **Environmental monitor:** Accurate display of temperature and humidity in proper format.
- **Motor controller:** Proper display format for each mode. Motor operation via slide pot and touchpad controls. Demonstrate tuning process to TA.
- **Real-time clock (Optional):** Display time and date in proper format. Set time and date. Survive MCU reset. *Up to 10% bonus awarded for successful completion.*

LAB REPORT

Following your demo, your group will submit two files to Brightspace under **Lab 4 Report**:

1. A standalone .pdf file containing the following:
   - A title page containing the current date, course code, lab number, lab section, student names, and student IDs.
   - Source code authored by your group for Lab 4:
     - Complete listings of `enviro.c`, `motor.c`, and (if applicable) `rtc.c` and `clock.c`
     - `SPI_Enable()` in `spi.c` and `TimerEnable()` in `timer.c`
   - A figure of your own creation showing the relationship between the various software components in the composite program created throughout Labs 1-4.
     - Show each component as a box containing its name (i.e. source file without the .s or .c extension) and a list of public functions and structures.
     - Organize the components into driver layer, application layer (tasks), and management layer (main).
     - Add interconnecting lines to show when one component uses the services of another (e.g. an app uses a driver, main code executes an app).
   - Note: There are **no exploration questions** in Lab 4.
2. A .zip file containing your IDE project used to demo. Follow these instructions to generate it:
   - In the STM32CubeIDE, right click your project and select Clean Project.
   - Exit the IDE completely.
   - **Delete the Debug folder within your project folder.**
   - Compress your entire project folder into a single .zip archive.

IMPORTANT: Submit the .pdf and .zip as *two separate files*, added to the same assignment submission. **Do not archive them together**, as this makes it more difficult to grade your report.