#cec32xNotes, cec32x_lab_32_Lab6–state_machine_n_user_input

Lab 6. State Machine and User Input

Introduction

When the control logic of a program with user inputs is complicated, it is a good idea to use a **state machine** to track/transition the state of the program according to the user input. A **state** is a situation of a system depending on previous situations, and it reacts to the current inputs to the system. One state is marked as the **initial state**; this is where the execution of the state machine starts.

In this lab, we implement a simple state machine that transitions based on user inputs from the joystick. **In each state, the user LEDs will light up in a certain pattern to indicate the state of the program is in**. The state can transition to a new state according to the user inputs from the Joystick keys. Specifically, for this lab, there are **four states** in the state machine, including the initial state, with the functionalities and transitions defined below:

- **State 0**. This is the initial state. Usually, we can do some initialization for the state machine here. But for this simple case, we do not do anything other than the automatic transition to State 1.
- **State 1**. In this state, LD_R will be on while LD_G will be off (**one LED solid on** to indicate **State 1**), and the program can only transition to State 2 when reading a pressing of the center key (JOY_C). It does not react to any other keys.
- **State 2**. In this state, both LD_R and LD_G will be on (**two LEDs solid on** to indicate **State 2**). There are two possible transitions from State 2. When the program reads a pressing of the right key (JOY_R), it transitions to State 3; when the program reads a pressing of the left key (JOY_L), it transitions back to State 1.
- **State 3**. In this state, LD_R and LD_G blink alternately (**one LED on alternately** to indicate **State 3**) every ‘’bDelay’’ ms (defined as 100 in the provided code snippet below), and the program can only transition to State 1 when reading a pressing of the left key (JOY_L).

To show the state transition clearly, we flash the LEDs in a specific pattern as shown later.

Note that you need to start with the given ”.ioc’’ file of your HW kit for Lab 3, which has both the red and green LEDs as well as the Joystick keys defined. This is part of your Task 1.

To make the program more readable and maintainable, we need to use functions or macros with meaningful names to hide lower-level implementations. We have been using functions in the past. Here, we use macros to improve the readability of the code. This is Task 2 of the lab.

With the meaningful macros, we can go ahead to program the **state machine**. This is Task 3.

Note also that programming is an iterative process. You write one block of code to implement a single functionality, then debug this block of code. When this block code is correct, you add more code for additional functionalities, then debug more. This approach is especially important for performing Tasks 2 and 3 of this lab. For this lab, much code for the state machine is given, and hence it is a good idea to define the macros used in the state as empty macros so that the project can be built without any error. After that, you can add in the true definitions of the macros piece by piece, until the entire code is finished.

Below are examples of a legitimately defined macro and an empty macro:

```
#define LD_R_ON             HAL_GPIO_WritePin(LD_R_GPIO_Port, LD_R_Pin, LED_ON)
#define JOY_C_IS_PRESSED    // this part is empty
```

Note that `LD_R_GPIO_Port` and `LD_R_Pin` are defined by CubeMX, and the `LED_ON` is defined by us, as seen from Workshop 5, which is used to address the issue of portable programming for different hardware kits.

Tasks of the lab

Task 1. Creating the project (20 points)

There are a few steps involved in creating the project for this lab.

- Make the project folder as described in the handouts of Lab 3. Specifically, you need to create the project folder named `lab06_stateMachine_n_userInput` with two subfolders, one named `usr_src` for the user source files and the other named `F412Discovery` or `L476Discovery` for the board related files.
- Copy the `.ioc` file for your HW kit from Lab 3 to the corresponding board folder.
- Click the `.ioc` file to regenerate the code using CubeMX. Note that you need to review the instructions in Lab 3 to do this step.
- Add the user code in the `main.c` file so that you can redirect to the `usr_tasks.c` file, as we did in Lab 3.
- Copy the `usr_tasks.c` and `usr_tasks.h` files from Lab 3 to this project.
- Set up the build folder in Keil so that the output will be saved in the `pjct_arm\build\xxx` of your HW kit and name the executable as `lab6_xxx`. Here `xxx` is `F412Discovery` or `L476Discovery`
- Build the project.

You may have to pay attention to the following when creating the project:

- You should add ‘’usr_tasks.c’’ to your project and the ‘’usr_src’’ folder to your **Include** path, as shown in the handouts given before. (Note that you have to be able to do these steps without using the handouts as these steps may appear in the final test of the lab.)

- You need to make changes to the ''main.c'' file to add the following:
  - including `usr_tasks.h` under `/* USER CODE BEGIN Includes */`
  - inserting `USR_Task_RunLoop();` under `/* USER CODE BEGIN 3 */`

If there is anything wrong, please contact the TA or instructor ASAP.

Task 2. Defining Macro definitions (24 points)

In this task, we define the following macros using HAL functions to control LEDs and Joystick keys in `usr_tasks.h`.

```
#define LD_R_ON   HAL_GPIO_WritePin(LD_R_GPIO_Port, LD_R_Pin, LED_ON)
#define LD_G_ON
#define LD_R_OFF
#define LD_G_OFF
#define LD_R_TOGGLE
#define LD_G_TOGGLE

#define JOY_C_IS_PRESSED
#define JOY_L_IS_PRESSED
#define JOY_R_IS_PRESSED
```

Note that you need to fill in the definitions of the last 8 macros above following the given example. The first half of `usr_tasks.h` is given at the end of this manual.

With these definitions, you can go ahead to program Task 3.

Task 3. A simple state machine with C (46 points)

Now, we program the state machine, which is shown in the introduction section given above.

Note that we need to use boolean variables, so make sure you include `stdbool.h` in this file.

You need to use the following code snippet after the macro definitions.

```
    enum progState{State0 = 0, State1, State2, State3, State4};
    enum progState currentState = State1;
    bool stateChanged = true;
    uint32_t bDelay = 100;
```

You need to figure out the meaning of the variables defined above from the context of this lab manual and the provided code. Here we only give *a little explanation* for `stateChanged`. This is a boolean variable used to record the change of the state. If there is a change of the state, we want to flash the LEDs in a certain pattern to signify to the user that there is a state change.

The state machine should be defined in the USR_Task_RunLoop function so that it will be called repeatedly. So, copy and paste the following code to the USR_Task_RunLoop function.

```
void USR_Task_RunLoop(void) {
    switch (currentState) {
    case State0:
        if (stateChanged) {
            currentState = State1;
            stateChanged = true;
        }
        break;

    case State1:
        if (stateChanged) {
            LD_R_G_flash();
            stateChanged = false;
        }

        if (JOY_C_IS_PRESSED) {
            currentState = State2;
            stateChanged = true;
            HAL_Delay(bDelay);
            break;
        }

        LD_R_ON;
        LD_G_OFF;

        break;

    case State2:
        if (stateChanged) {
            // provide your code so that you can flash the LEDs and
            // remove the stateChanged flag.
        }

        if (JOY_L_IS_PRESSED) {
            // provide your code to handle the left Joy key input.
        }
        if (JOY_R_IS_PRESSED) {
            // provide your code to handle the right Joy key input.
        }


        // provide your code for LED control
        break;

    case State3:
        if (stateChanged) {
            // provide your code
            // note that you need also provide your code so that the
            // LEDs will blink alternately.
        }

        // provide your state transition code

        // Note that the LEDs have to blink alternately.
        LD_R_TOGGLE;
        LD_G_TOGGLE;
        HAL_Delay(bDelay);

        break;

    default:
        LD_R_OFF;
        LD_G_OFF;
        break;
    }
}

void LD_R_G_flash(void) {
    for (int i = 0; i < 3; i++) {
        LD_R_OFF;
        LD_G_OFF;
        HAL_Delay(100);
        LD_R_ON;
        LD_G_ON;
        HAL_Delay(100);
    }
    LD_R_OFF;
    LD_G_OFF;
    HAL_Delay(500);
}
```

Note that:

- Many C statements are missing in the above code snippet. You are supposed to code them to make the state machine work as described.
- The `stateChanged` variable is used to avoid the flashing of the LEDs when there is no transition of the state.

- We include `HAL_Delay(bDelay);` in each state transition to prevent sticky keys when a certain key is used to transition in consecutive states. (We don't have this case, but the code is provided just in case you want to play with different transitions.)
- An LED flashing function is provided at the end of the above code. Note that this function is also defined in the `usr_tasks.h` file, as shown below

```
#ifndef __USER_TASKS_H
#define __USER_TASKS_H

#ifdef __cplusplus
extern "C" {
#endif

////////////////////////////////////////////////
#include "main.h"

void USR_Task_RunLoop(void);
void LD_R_G_flash(void);

// Definitions used in static functions
#if defined(STM32L476xx)
    #define LED_ON GPIO_PIN_SET
    #define LED_OFF GPIO_PIN_RESET
#elif defined(STM32F412Zx)
    #define LED_ON GPIO_PIN_RESET
    #define LED_OFF GPIO_PIN_SET
#endif

// Macro definitions
#define LD_R_ON   HAL_GPIO_WritePin(LD_R_GPIO_Port, LD_R_Pin, LED_ON)
// more definitions of macros and other parts of the file ...
}
```

Lab report

10 points will be evaluated based on the cleanliness of the report, which should include the following:

- Your definitions for the macros in Task 2.
- Your code for the state machine in Task 3.