Tags: #cec32x, #lab, cec32x_lab_40–Lab8_state_machine_n_audio

Lab 8. State Machine and Audio

Introduction

In Lab 6, we learned / practiced how a state machine can be used to efficiently handle the complex structure of the program. In this lab, we will use a similar state machine to practice more along this line.

Similar to Lab 6, we use some devices that are populated on both boards, which includes:

- The green and red user LEDs.
- The Joystick keys.
- Digital microphone(s). (The STM32L476G-Discovery board has one, and the STM32F412G-Discovery board has two, which are separated by 21 mm to make a stereo microphone set.)
- Audio codec which can be used to play audio signals to headphones via a stereo headphone jack.

Porting of a project

The original intention of this project is to practice the porting of the project from a demo project. However, after a couple of rounds of the writing of the lab handout, this idea is killed since it can be risky to port a project in a short period of time. (If we get stuck, a few days can be passed without having noticeable progress, which is not a thing we want to see for the lab.) In this lab, the ported code is provided for you so that you can start to build the project. Still, porting a project is an important skill we want to have. So, we move the porting part of the project to the **Dev tool** section under **Modules** on Canvas. Please see the `ece32x_devtool_21_porting_a_project.pdf` file.

Before moving on to discuss the project requirement, let us introduce a new topic, **BSP, board support package**.

Using BSP in the project

We have practiced how to initialize the LEDs and the Joystick keys in previous projects using the HAL functions.

Here, in this lab, we use something on a higher level of abstraction, the BSP, board support package, for the initialization and operation of these devices. Usually, for whatever board we purchase, there should be a BSP coming with the board so that we do not have to do the lower-level setup of the board to get it to work. In the case that the board is produced in-house, there is still a BSP accompanying it as the onboard components have to be tested for functionalities. (There may be a chance for you to write that BSP in the future.)

BSP for controlling the LEDs and Joystick keys

In this lab, we will see how BSP works for the LEDs and Joystick keys. We can see from the provided BSP code (`stm32412g_discovery.c` or `stm321476g_discovery.c` and their corresponding header files) that both boards use the same format to define the control structs for LEDs and Joystick keys. They also use the same names for functions that operate on these I/O devices. This significantly reduces the work to support multiple boards in a single project. This way, we can use the Joystick keys in exactly the same way on the two boards. Considering the fact that the two boards have two user LEDs in common in terms of color (the red LED and the green LED). We can do some further definitions so that the user code will not see any difference between using different boards when using these two LEDs. The corresponding code is provided in `usr_tasks.c`. (See lines 9 and 10 and 25 and 26, respectively.)

Specifically, we use the following BSP functions in this lab:

- `BSP_LED_Init(LED_grn)` for initializing the green LED.
- `BSP_LED_On(LED_grn)` or `BSP_LED_Off(LED_grn)` for turning on / off the green LED.
- `BSP_LED_Toggle(LED_grn)` for toggling the green LED.
- `BSP_JOY_Init(JOY_MODE_GPIO)` for initializing the Joystick keys in the GPIO mode.
- `BSP_JOY_GetState()` for getting the state of the Joystick keys.

Some of these functions are called in the provided `usr_tasks.c` file. Please take a close look at them.

BSP for operating the microphone(s)

Both discovery boards of this lab use MP34DT01TR, a MEMS microphone, to pick up sound signals from the air. The only difference between STM32F412G-Discovery and STM32L476G-Discovery is that the former has two microphones working in stereo and the latter has only one. Since the microphone(s) can pick up the audio signal from the air, these boards can be used in many applications where audio control is needed, such as the sound control of the lights in the office or at home.

The MCUs of these boards connect to the microphones via DFSDM (digital filter for sigma-delta modulator), a peripheral on the MCU, which is designed to connect to microphones. The details of these controls are beyond the scope of this course. Fortunately, there are some good references to learn from.

The BSP code for the operation of the microphone is provided in the attached files of the project.

Specifically, there are a couple of operations needed to use the microphone(s):

- `DFSDM_Init()` for initializing the device—configuring the various registers in DFSDM.
- `HAL_DFSDM_FilterRegularStart_DMA(...)` for setting up the DMA (direct memory access). DMA is a topic that will be covered later in the class. The main point to know right now is that with DMA we can move blocks of data between the MCU and the peripherals without our explicit programming—the DMA controller will be able to move these data in parallel to the other operations of the MCU. This will liberate the MCU from these mundane tasks. Note that the DMA is used to move the data from DFSDM to the receiver buffer defined in `main.c`, which is specific to each board:
  - For F412:

```
int32_t                         LeftRecBuff[2048];
int32_t                         RightRecBuff[2048];
```

  - For L476:

```
int32_t                         RecBuff[2048];
```

- In the user code, when the above receiver buffer is half full or completely full, the DMA system will trigger an interrupt so that we can copy the data to the play buffer used for audio playback and signal processing. The topic of the interrupt will be covered later as well.
- To trigger the required actions after copying the entire block of data to the play buffer, we need to set a flag called `playSound` to `true`, as shown in the `copy_from_rec_to_play()` function called at the end of the `USR_Task_RunLoop()` function in the provided `usr_tasks.c` file.

Note that the magic number of 2048 is the number of audio signal samples the buffer is designed to hold. This is a very often used trick in real-life programming where we handle the signals in batches to save the time for back and forth switching of the context for processing.

BSP for operating the audio codec

An audio codec is included on each board, which can be used to play the audio via a stereo headphone plug connected to the codec in the board. Note that a conventional stereo headphone with a 3.5 mm plug is needed to listen to the audio played by the audio codec.

These two boards use different audio codec devices. The STM32F412G-Discovery board uses WM8994, and the STM32L476G-Discovery board uses CS43L22. The former uses an I2S (inter-IC sound) interface to send the audio data from the MCU to the codec, and the latter uses an SAI (serial audio interface) interface to send the audio data. They both use an I2C (inter-integrated circuit) interface for the control of the codec.

Specifically, we use `Playback_Init()` to initialize the audio codec. We use `play_sound()`, given in the provided `usr_tasks.c` file and shown below, to play the audio signal.

```
void play_sound(void) {
    if(PlaybackStarted == 0) {
        if(0 != audio_drv->Play(AUDIO_I2C_ADDRESS,
                    (uint16_t *) &PlayBuff[0], 4096)) {
            Error_Handler();
        }
#if defined(STM32L476xx)
        if(HAL_OK != HAL_SAI_Transmit_DMA(&SaiHandle,
                    (uint8_t *) &PlayBuff[0], 4096)) {
#elif defined(STM32F412Zx)
        if(HAL_OK != HAL_I2S_Transmit_DMA(&haudio_i2s,
                    (uint16_t *) &PlayBuff[0], 4096)) {
#endif
            Error_Handler();
        }
        PlaybackStarted = 1;
    }
}
```

Note that the `audio_drv->Play(...)` function is used to inform the codec about the intended function—play the audio signal. This is done via the I2C bus. The data to play is sent via the DMA transmission, where both the addresses and the size of the buffer are needed.

We only call this play function after the `playSound` flag is set by the data copy function signaling a new batch of data is available. See the end of the `usr_Task_RunLoop()` function for details.

Note also that the magic number of 4096 is the number of audio signal samples to send to the buffer of the codec. This is double the size of 2048 as the codec is stereo. Also, not like the receive buffer, which is 32 bits, the play buffer is only 16 bit.

After sending the data to the codec, we need to process the audio signal for additional user functionalities, as seen below.

Signal processing and control of the red LED

As we will see later, this project requires us to process the audio signal. Specifically, we need to calculate **the maximum absolute value** of the signal in batch (4096 for the two play channels) every time the play buffer is updated. This value will be used to control the red user LED according to a sound threshold, which is a function of the state.

Note that the above signal processing is contained in a function called `find_audio_max()` in the provided `usr_tasks.c` file. Programming this function is one of the tasks of this lab, as detailed later.

Following the `find_audio_max()` in the provided `usr_tasks.c` file is the `operate_red_LED()` function. Here, you need to program the The control of the red user LED as required below.

Project requirements

In this project, we implement the following user functionalities using the above devices based on provided code.

1. We want to have **three working states**—State 1 to State 3. There should be an initial state, State 0, as well.
2. **In all states**, we need to receive the audio signal (in batches of 2048 samples) from the onboard microphone and play it back to the audio codec. There can be some delay between the received and the playback audio signals. This is not a concern though. We also need to calculate the maximum absolute value of the received audio signal in each batch and use this value to control the red user LED according to a given **sound-level threshold**, which is a function of the state.
3. **In all states**, we need to toggle the green user LED. The toggle rate is a function of the state as well. There are many implementation approaches to do this, and we use a simple one in this project—we use the (super) `while` loop to increase a variable named `loop_num`. When it is greater than a preset **loop threshold**, we toggle the LED and clear the value of `loop_num` 0 so that the process can repeat again.
4. **In State 1**, we toggle the green user LED at a rate of about 1 second (plus / minus 50%) by assigning an appropriate value to the loop threshold. The sound threshold should be set to such a value that a light clap or knock on the table can turn on the red LED; otherwise, it will turn off this LED. (Note that this should be performed in each of the super `while` loop.) You need to figure out the values of these two thresholds by trying out. Note that the value of the loop threshold is a function of the time used for each pass of the super while loop. As such, it is suggested to change this threshold only after all the functionalities are finished to avoid repeated changes.
5. **In State 2**, we toggle the green user LED at a rate of about 0.5 seconds (plus / minus 50%) by assigning anther appropriate value to the loop threshold. The sound threshold should be set to such a value that a heavier clap or knock on the table can turn on the red LED; otherwise, it will turn off this LED. You need to figure out the values of these two thresholds. To test if the sound thresholds for States 1 and 2 are chosen appropriately, we need to be able to turn on the LED in State 1, but not in State 2 with the same level of the sound signal.
6. **In State 3**, we toggle the green user LED at a rate of about 0.25 seconds (plus / minus 50%) by assigning an appropriate value to the loop threshold. The sound threshold should be set to such a value that the red LED will never be turned on no

matter how loud the sound is although we use the same code (approach) in the loop as that in States 1 and 2.

Since the initialization of the GPIO pins is done in State 0, as shown in the `usr_tasks.c` file, we do not have to provide other user functions except `USR_Task_RunLoop()` given in this file. Due to this, we can use the same `usr_tasks.h` file of Lab 6, as shown in the given project files. Note that the state machine is defined in the `USR_Task_RunLoop()` function in `usr_tasks.c`. The transitions of the four states of the state machine are given below:

- **State 0**. This is the initial state. Usually, we can do some initialization for the state machine here. For this lab, we initialize the GPIO pins for the two user LEDs and the 5 push buttons for the Joystick keys. The state machine automatically transitions to State 1 after running the above initialization code. The two threshold variables for the LED toggle and sound control should be initialized here as well, as seen in the given code in the `usr_tasks.c` file.
- **State 1**. In this state, the program transitions to State 2 when reading a pressing of the center key (defined as JOY_SEL). (The code is given in the `usr_tasks.c` file.) It transitions to State 3 when reading a pressing of the right key (defined as JOY_RIGHT). (This should be programmed.)
- **State 2**. In this state, the program transitions to State 1 when reading a pressing of the left key (defined as JOY_LEFT). (The code is given in the `usr_tasks.c` file.) It transitions to State 3 when reading a pressing of the right key. (This should be programmed.)
- **State 3**. In this state, the program transitions to State 1 when reading a pressing of the left key. It transitions to State 1 after the above initialization code. (The code is given in the `usr_tasks.c` file.) It transitions to State 2 when reading a pressing of the center key. (This should be programmed.)

Tasks of the lab

Task 1. Cleaning up the provided project

(20 points) There are two subtasks in this task.

Task 1a. Building the ported code

You should be able to build the provided code. If you download the code to your board, you should be able to hear the echo in the headphones when you knock on the table.

Task 1b. Moving the user code to the usr_tasks.c file

As we did in Lab 6, we want to have all the user application-related code in the `usr_tasks.c` file. This way, we don't have to mess around in the original `main.c` file to make it possible to use our user code on different boards. Here, we do it a bit differently.

We make a copy of the `main.c` file to `main_F412.c` or `main_L476.c` before changing anything. This way, we can always go back to the original file to try it out when our modification does not work as needed.

Then, we need to remove the `main.c` from the project and add the `main_F412.c` or `main_L476.c` file. Below, we will refer these files as the `main_hw.c` file for the convenience of discussion.

Now, we open the `main_hw.c` file in Keil. Cut the code in the super `while (1)` loop of the `main_hw.c` file and paste it to the `USR_Task_RunLoop(void)` function in the `usr_tasks.c` file. (This is the code used to handle the audio signal.) You need to replace the cutoff code in the super `while (1)` loop using the `USR_Task_RunLoop(void)` function so that the we can execute the code in the `USR_Task_RunLoop(void)` function defined in the `usr_tasks.c` file. Note that you need to include the `usr_task.h` file in your `main_hw.c` file as well.

To reduce the interference between the audio handling code just copied from the super `while` loop and the existing code in the `usr_tasks.c` file, comment out all the existing code in the `USR_Task_RunLoop()` function.

Note that to build the project, we need to **declare** the variables defined in `main_hw.c` in `usr_tasks.c` to use them. These declarations are provided by these lines starting with `extern`.

There may be some warnings when you build the code. To remove the warnings, you may want to remove these codes that are not used in `main.c` anymore. For example, the line of `unit32_t i;` at the beginning of the function.

Now, you should have the same functionality as Task 1a. If not, please see the lab TA.

Task 2. Programming the state machine

(20 points)

Now, comment out the code copied from the `main_hw.c` file and uncomment the existing code in the `USR_Task_RunLoop(void)` function. Add the missing code to have a fully functioning state machine in terms of state transitions.

No worries about the flashing of the LED or the sound for this step.

Task 3. Programming the signal copy function

(10 points)

Now, look at the commented out code copied from the `main_hw.c` file and finish the missing part of the code in the `copy_from_rec_to_play()` function. The missing part is

labeled as `// Add code to copy to the "right channel"`. You need to copy the left channel signal to the right channel. You need to figure out the patter of the data saving to finish this task.

Task 4. Programming the signal processing function

(20 points)

Next, program the function for finding the maximum absolute value of the audio signal in the play buffer. This is in the `find_audio_max()` file. You need to add your code in the `// Add code here` section.

Task 5. Finding the appropriate thresholds

(20 points)

Test the values of the 6 thresholds in the code to finish the functionalities of the project.

Lab report

(10 points)

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

- Your code snippets.
- In the provided code, we use `BSP_JOY_GetState()` to return a value according to the pressing of the Joystick keys. Please read the source code of this function to figure out what (numerical) value will be returned from this function if the left key is pressed. What value is returned if no key is pressed at all? Write these values in the report.