UCSD Embedded RTOS Final

Aaron Stafford

Here is a link to a video overview of the project:
https://youtu.be/9gMGHNSOIm0

I ran out of time to get this where I wanted, but learned a lot, and had a lot of fun too. Will likely spend a few more hours cleaning it up. Some lessons learned: It would be useful to set up some unit test, especially to make sure you do not exceed the allocated Heap. I ran into this problem, and the outcome was that tasks would randomly not run, due to the fact they were not allocated enough memory. Also, I enjoyed trying out the PWM to create sounds. I know the board has a DAC, which I would also like to play around with, but for now I wanted to try using 4 timers to create 4 separate, independent sounds.

Tasks configured at the beginning. I was working towards isolating parts of the code, so the tasks for managing the coffee maker were initialized using a function in a separate file. Some were suspended to wait for the system to be 'powered on'

```
/* add threads, ... */
createCoffeeMakerTasks();
BaseType_t status1 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask1", 128, &xBuzzer1, 4, NULL);
BaseType_t status2 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask2", 128, &xBuzzer2, 3, NULL);
BaseType_t status3 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask3", 128, &xBuzzer3, 3, NULL);
BaseType_t status4 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask4", 128, &xBuzzer4, 3, NULL);
vTaskStartScheduler();
/* USER CODE END RTOS_THREADS */
```

```
void createCoffeeMakerTasks() {
    xTaskCreate((TaskFunction_t)vCoffeeMakerResetTask, "CoffeeMakerResetTask", 128, NULL, 2, NULL);
    xTaskCreate((TaskFunction_t)vControlCoffeeMakerTask, "CoffeeMakerControlTask", 128, NULL, 1, NULL);
    xTaskCreate((TaskFunction_t)vOLEDManagerTask, "OLEDManagerTask", 128, NULL, 1, &xOLEDTaskHandle);
    xTaskCreate((TaskFunction_t)vUpdateLEDs, "UpdateIndicatorLEDs", 128, NULL, 1, &xLEDUpdateTaskHandle);
    xTaskCreate((TaskFunction_t)vCoffeeMakerShutdownTask, "CoffeeMakerShutdownTask", 128, NULL, 1, &xShutdownTaskHandle);
    vTaskSuspend(xOLEDTaskHandle);
    vTaskSuspend( xShutdownTaskHandle );
}
```

A one shot timer was used to timeout the different states, while repeating timers were used to check the analog sensor, poll the GPIOs, and time the beat for the sounds

```
//xbuzzer1Timer = xTimerCreate("B1", pdMS_TO_TICKS(1), pdTRUE, NULL, prvBuzzer1);
 xTimeoutTimer = xTimerCreate("Timeout Timer", pdMS_TO_TICKS(1000), pdFALSE, NULL, prvTimeoutTimer);
 xReadTempSensorTimer = xTimerCreate("Temp Sensor Timer", pdMS_TO_TICKS(400), pdTRUE, NULL, prvReadTempSensorTimer);
 xReadWaterLevelSensorTimer = xTimerCreate("Water Level Timer", pdMS_TO_TICKS(400), pdTRUE, NULL, prvReadWaterLevelSensorTimer)
 xBeatTimer = xTimerCreate("Beat Timer", pdMS_TO_TICKS(PERIOD_SIXTEENTH_MS), pdTRUE, NULL, prvBeatTimer);
 xPollGPIOTimer = xTimerCreate("Poll GPIO Timer", pdMS_TO_TICKS(50), pdTRUE, NULL, prvPollGPIOTimer);
//xTimerStart(xbuzzer1Timer,0);
```

Turning on, and timing out – the blue button was used as a power button, and turned the system both on and off using semaphores.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    //check if interrupt triggered by blue pin, enter critical area and increment counter
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    if(GPIO_Pin == BUTTON_EXTI13_Pin) {
        QueueHandle_t semaphoreToSet;
        if(getMachineState() == OFF) {
            semaphoreToSet = xOnSemaphore;
        } else {
            semaphoreToSet = xOffSemaphore;
        }
        xSemaphoreGiveFromISR(semaphoreToSet, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }

}
```

The coffee maker reset task waited for the ‘on’ semaphore and then started the tasks and timers, for the one shot timeout timer, the period was set for the timeout value that corresponded to the current state

```
void vCoffeeMakerResetTask(void const * argument) {
    for(;;) {
        BaseType_t xSemaphoreReceived = xSemaphoreTake(xOnSemaphore, portMAX_DELAY);
        //start timeout timer
        if(machineStatus == OFF) {
            xTimerStart(xBeatTimer,0);
            xTimerStart(xReadWaterLevelSensorTimer,0);
            xTimerStart(xPollGPIOTimer,0);
            vTaskResume(xOLEDTaskHandle);
            vTaskResume(xLEDUpdateTaskHandle);
            vTaskResume(xShutdownTaskHandle);
        }
        machineStatus = ON;
        xTimerChangePeriod( xTimeoutTimer, pdMS_TO_TICKS(ON_DELAY), 50 );
        xSemaphoreGive(xChangeSizeSemaphore);
        HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, SET);


    }
}
```

The timeout timer and coffee maker control tasks managed the states and the timeouts, using semaphores to signal a timeout

```
void prvTimeoutTimer(xTimerHandle xTimer) {
    switch(machineStatus) {
    case ON:
        xSemaphoreGive(xOffSemaphore);
        break;
    case BREWING:
        xSemaphoreGive(xWarmingSemaphore);
        break;
    case WARMING:
        xSemaphoreGive(xOnSemaphore);
        break;
    case OFF:
        break;
    }
}
```

```
void vControlCoffeeMakerTask(void const * argument) {
    BaseType_t xSemaphoreReceived;
    for(;;) {
        switch(machineStatus) {
        case ON:
            //turn off relay
            HAL_GPIO_WritePin(ARD_A0_GPIO_Port, ARD_A0_Pin, RESET);
            if(xSemaphoreTake(xBrewingSemaphore, portMAX_DELAY)==pdTRUE) {
                machineStatus = BREWING;
                HAL_GPIO_WritePin(ARD_A0_GPIO_Port, ARD_A0_Pin, SET);
            }
            break;
        case BREWING:
            //config brew settings
            xTimerChangePeriod( xTimeoutTimer, pdMS_TO_TICKS(BREW_DELAY), 50 );
            xSemaphoreReceived = xSemaphoreTake(xWarmingSemaphore, portMAX_DELAY)
            xTimerChangePeriod( xTimeoutTimer, pdMS_TO_TICKS(WARM_DELAY), 50 );
            machineStatus = WARMING;
            break;
        case WARMING:

            break;
        default:
            break;
        }
    }
}
```

The shutdown task was triggered by the button interrupt or the timeout, and suspends tasks and turns off LEDs

```
void vCoffeeMakerShutdownTask(void const * argument) {
    for(;;) {
        BaseType_t xSemaphoreReceived = xSemaphoreTake(xOffSemaphore, portMAX_DELAY);
        machineStatus = OFF;
        xTimerStop(xReadWaterLevelSensorTimer, 100);
        xTimerStop(xPollGPIOTimer,0);
        HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, RESET);


        HAL_GPIO_WritePin(ARD_D13_GPIO_Port, ARD_D13_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(ARD_D11_GPIO_Port, ARD_D11_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(ARD_D10_GPIO_Port, ARD_D10_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(ARD_D9_GPIO_Port, ARD_D9_Pin, GPIO_PIN_RESET);
        vTaskSuspend(xLEDUpdateTaskHandle);


        xSemaphoreTake(xOLEDMutex,portMAX_DELAY);
        vTaskSuspend(xOLEDTaskHandle);
        ssd1306_Fill(Black);
        ssd1306_UpdateScreen();
        xSemaphoreGive(xOLEDMutex);
        vTaskSuspend( NULL );

    }
}
```

The OLED manager task, is triggered by semaphore to update the OLED when it has new information available. It uses a mutex so that it is not writing to the display when the shutdown task is trying to clear the display.

```
void vOLEDManagerTask(void const * argument) {
    for(;;) {
        if(xSemaphoreTake(xOLEDUpdateSemaphore, portMAX_DELAY) == pdTRUE) {
            if(xSemaphoreTake(xOLEDMutex,0) == pdTRUE) {
              //clear LCD
              ssd1306_Fill(Black);

              //display the time on the LCD
              displayTime();
              //display the machine State on the LCD
              displayMachineState();
              //display the size selection on the LCD
              displayMode();
              //display the water level on the LCD
              displayWaterLevel();
              //updates the LCD with the new information
              ssd1306_UpdateScreen();
              HAL_Delay(50);
              xSemaphoreGive(xOLEDMutex);
            }
        }
    }
}
```

This timer sets an event group bit so the four instances of the buzzer task can keep time and in sync. The four tasks pass an argument to the same task function. The lowest priority task is responsible for clearing the event bit.

```c
typedef struct {
    TIM_HandleTypeDef *timerHandle;
    uint32_t PWMChannel;
    uint8_t buzzerID;
    uint16_t *buzzerNotes;
    uint16_t *buzzerLengths;
    uint16_t numNotes;
} buzzerDataType;
```

```c
void prvBeatTimer(xTimerHandle xTimer) {
    xEventGroupSetBits(xNoteEventGroup, SIXTEENTH_NOTE_BIT);
}
```

```c
BaseType_t status1 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask1", 128, &xBuzzer1, 4, NULL);
BaseType_t status2 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask2", 128, &xBuzzer2, 3, NULL);
BaseType_t status3 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask3", 128, &xBuzzer3, 3, NULL);
BaseType_t status4 = xTaskCreate((TaskFunction_t)vBuzzerTask, "BuzzerTask4", 128, &xBuzzer4, 3, NULL);
```

Buzzer task – This is used by the four buzzers. It counts the number of 16th notes, and triggers the next note when the correct number of 16ths have passed. Changes the PWM frequency to match the frequency of the note being played.

```c
void vBuzzerTask(void const * argument) {
    TIM_HandleTypeDef *ptimerHandle = ((buzzerDataType *)argument)->timerHandle;
    uint32_t PWMChannel = ((buzzerDataType *)argument)->PWMChannel;
    uint8_t buzzerNumber = ((buzzerDataType *)argument)->buzzerID;
    uint16_t *notes = ((buzzerDataType *)argument)->buzzerNotes;
    uint16_t *delays = ((buzzerDataType *)argument)->buzzerLengths;
    uint16_t numNotes = ((buzzerDataType *)argument)->numNotes;
    note_status_t buzzer_status = PAUSE;
    EventBits_t xEventGroupValue;
    int currentNote = 0;
    int noteLength = delays[currentNote];
    for(;;) {
        uint8_t clearBits = (buzzerNumber == LOW_PRIORITY_BUZZER_ID);
        xEventGroupValue = xEventGroupWaitBits(xNoteEventGroup,
                                               SIXTEENTH_NOTE_BIT,
                                               clearBits, //do not clear bits
                                               pdTRUE, // Wait for all bits
                                               portMAX_DELAY);
        if(buzzer_status == PAUSE) {
            buzzer_status = SOUND_ON;
            int current_arr = calcARR(ptimerHandle, notes[currentNote]);
            ptimerHandle->Instance->ARR = current_arr;
            __HAL_TIM_SET_COMPARE(ptimerHandle, PWMChannel, ((current_arr + 1)/2)-1);
            //ptimerHandle->Instance->CCR2 = ((current_arr + 1)/2)-1;
            HAL_TIM_PWM_Start(ptimerHandle, PWMChannel);
        }
        noteLength--;
        if(noteLength <= 0) {
            currentNote++;
            buzzer_status = PAUSE;
            HAL_TIM_PWM_Stop(ptimerHandle, PWMChannel);
            if(currentNote >= numNotes) {
                currentNote = 0;
                noteLength = delays[currentNote];
                if(buzzerNumber == LOW_PRIORITY_BUZZER_ID) {
                    xTimerStop(xBeatTimer,portMAX_DELAY);
                }
            } else {
                noteLength = delays[currentNote];
            }
        }
    }
}
```

```c
int calcARR(TIM_HandleTypeDef *htim, int frequency) {
    return TIMER_FREQUENCY/(htim->Instance->PSC + 1)/frequency - 1;
}
```