SANTA CLARA UNIVERSITY
Electrical and Computer Engineering Department

Real-Time Embedded Systems - ECEN 121
Lab 5

Dylan Thornburg & Kai Hoshide

Concurrency and Buffer Management

Andrew Wolfe

The objective of this project is to build and test FIFO data buffers using timer and interrupt driven activity. There are a few differences from some of the recent labs:

1. I am not going to provide step-by-step directions. You need to think through the problems.
2. There is no output device. To test and demo, you will need to stop the program using breakpoints and examine the contents of memory.
3. To demonstrate robustness, you may need to demo the same program multiple times in a row.
4. You need to explain in your lab report why your concurrency implementation is correct.

Prelab:

Read the lab and submit a prelab that says "I read the lab handout" signed by all group members. This will only be worth 2 points. There is extra work on the lab report this week and it will be worth extra points.

Step 1:

You need to use the CubeMX tool to create and configure a project. Read the lab requirements and then you decide whether to start from scratch or use a copy of one of your existing projects as a starting point.

Your project will need to be configured to use the following resources.

• A 20MHz CPU clock

• Timer TIM3 configured to interrupt every 10ms.
20000000/(xy)=100
xy=200000; x = 200; y =1000

• Timer TIM6 configured to interrupt every 48ms.
20000000/(xy)=125/6
xy=960000; x =960; y =1000

• The center joystick button configured to interrupt when pressed.

Step 2:

You will add at least the following data structures to your project.

An integer variable `Count`

An integer array variable `Buffer[64]`

An integer array variable `Snapshot[64]`

You will also add whatever variables are needed to get the project to work. It is also your responsibility to do whatever is necessary to make sure that there are no data corruptions due to concurrency problems or interrupts occurring at the wrong time or interrupts not occurring at the right time. Note that if you keep the default interrupt priorities, none of these interrupts can pre-empt each other.

Project 1:

You must implement a circular buffer in the `Buffer[]` array. A circular buffer is a first-in-first-out queue. Data elements are added to the end of the queue and removed from the beginning of the queue. If the queue is full, you should not add any more data to it. If the queue is empty, you should not try to remove any elements from it. If you get to the end of the array and the array is not full, the next element should be added to the beginning of the array. That is why it is circular. For Project 1, the `Buffer[]` array should be able to hold 63 unsigned integers when it is full.

Each time timer TIM3 triggers an interrupt (i.e. every 10ms):

- If the queue is not full
    - Insert the current value of `Count` at the end of the queue
    - Increment `Count`

Each time timer TIM6 triggers an interrupt (i.e. every 48ms):

- If there are 4 or more elements in the queue
    - Remove the oldest 4 elements from the queue
- If there are between 1 and 3 elements in the queue
    - Remove all the elements from the queue

Each time the center joystick interrupt is triggered.

- Copy all of the currently-valid values in the queue to the array `Snapshot[]` such that `Snapshot[0]` is the oldest value in the queue and each of the other values follow in order. Fill the remaining elements of `Snapshot[]` with the value 0xfeedbeef.

To demo, you can run the program in the debugger then stop it, either manually after pressing the center joystick or with a breakpoint at the end of the center joystick interrupt routine. Examine

snapshot. It should contain valid, uncorrupted data. To get credit for the demo, you must explain to the TA how you know the data is uncorrupted.

In your lab report you must explain in detail how you know that no data structures are corrupted due to concurrency violations. **We know that the data won't be corrupted since interrupts will be handled by the NVIC. The NVIC stops other interrupts from happening until the first interrupt is concluded.**

CODE for Part 1:

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file    stm32l4xx_it.c
  * @brief   Interrupt Service Routines.
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32l4xx_it.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN TD */

/* USER CODE END TD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
unsigned int Count = 0;
int Buffer[64];
int Snapshot[64];
int Count2 = 0;
int front = 0;
int rear = 0;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

//Function to determine distance between tail and head
int Distance()
{
	if (rear >= front)
	{
		return (rear-front);
	}
	return (rear+ 64 - front);
}
// Function to add data to the circular buffer
void addToBuffer(int data)
{
					if((rear + 1) % 64 == front)
					{
```

```
                        return;
                    }
                    Buffer[rear] = Count;
                    Count++;
                    rear = (rear + 1) % 64;
}

// Function to remove elements from the circular buffer
void removeFromBuffer(int num)
{
                    if (Distance() >=4)
                    {
                        front = (front + 4) % 64;
                    }
                    else
                    {
                        front = rear;
                    }
}

// Function to copy valid values from buffer to snapshot
void copyToSnapshot()
{
                int i = 0;
                int f = front;
                while (f != rear)
                {
                    Snapshot[i] = Buffer[f];
                    i++;
                    f = (f +1) % 64;
                }
                while (i < 64)
                {
                    Snapshot[i] = 0xfeedbeef;
                    i++;
                }
}



/* USER CODE END 0 */

/* External variables --------------------------------------------------------*/
extern LCD_HandleTypeDef hlcd;
extern TIM_HandleTypeDef htim3;
extern TIM_HandleTypeDef htim6;
extern TIM_HandleTypeDef htim16;
/* USER CODE BEGIN EV */

/* USER CODE END EV */

/******************************************************************************/
/*           Cortex-M4 Processor Interruption and Exception Handlers          */
/******************************************************************************/
/**
  * @brief This function handles Non maskable interrupt.
  */
void NMI_Handler(void)
{
  /* USER CODE BEGIN NonMaskableInt_IRQn 0 */

  /* USER CODE END NonMaskableInt_IRQn 0 */
  /* USER CODE BEGIN NonMaskableInt_IRQn 1 */
  while (1)
  {
  }
  /* USER CODE END NonMaskableInt_IRQn 1 */
}

/**
  * @brief This function handles Hard fault interrupt.
  */
void HardFault_Handler(void)
{
  /* USER CODE BEGIN HardFault_IRQn 0 */

  /* USER CODE END HardFault_IRQn 0 */
  while (1)
  {
    /* USER CODE BEGIN W1_HardFault_IRQn 0 */
    /* USER CODE END W1_HardFault_IRQn 0 */
  }
}

/**
  * @brief This function handles Memory management fault.
  */
void MemManage_Handler(void)
{
  /* USER CODE BEGIN MemoryManagement_IRQn 0 */

  /* USER CODE END MemoryManagement_IRQn 0 */
  while (1)
  {
    /* USER CODE BEGIN W1_MemoryManagement_IRQn 0 */
    /* USER CODE END W1_MemoryManagement_IRQn 0 */
  }
}

/**
  * @brief This function handles Prefetch fault, memory access fault.
  */
void BusFault_Handler(void)
{
  /* USER CODE BEGIN BusFault_IRQn 0 */

  /* USER CODE END BusFault_IRQn 0 */
  while (1)
  {
    /* USER CODE BEGIN W1_BusFault_IRQn 0 */
    /* USER CODE END W1_BusFault_IRQn 0 */
  }
}

/**
  * @brief This function handles Undefined instruction or illegal state.
  */
void UsageFault_Handler(void)
{
  /* USER CODE BEGIN UsageFault_IRQn 0 */

  /* USER CODE END UsageFault_IRQn 0 */
  while (1)
  {
    /* USER CODE BEGIN W1_UsageFault_IRQn 0 */
    /* USER CODE END W1_UsageFault_IRQn 0 */
  }
}

/**
  * @brief This function handles System service call via SWI instruction.
  */
void SVC_Handler(void)
{
  /* USER CODE BEGIN SVCall_IRQn 0 */

  /* USER CODE END SVCall_IRQn 0 */
  /* USER CODE BEGIN SVCall_IRQn 1 */
  /* USER CODE END SVCall_IRQn 1 */
}

/**
  * @brief This function handles Debug monitor.
  */
void DebugMon_Handler(void)
{
  /* USER CODE BEGIN DebugMonitor_IRQn 0 */

  /* USER CODE END DebugMonitor_IRQn 0 */
  /* USER CODE BEGIN DebugMonitor_IRQn 1 */

  /* USER CODE END DebugMonitor_IRQn 1 */
}

/**
  * @brief This function handles Pendable request for system service.
  */
void PendSV_Handler(void)
{
  /* USER CODE BEGIN PendSV_IRQn 0 */

  /* USER CODE END PendSV_IRQn 0 */
  /* USER CODE BEGIN PendSV_IRQn 1 */

  /* USER CODE END PendSV_IRQn 1 */
}

/**
  * @brief This function handles System tick timer.
  */
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

  /* USER CODE END SysTick_IRQn 0 */
  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */

  /* USER CODE END SysTick_IRQn 1 */
}

/******************************************************************************/
/* STM32L4xx Peripheral Interrupt Handlers                                    */
/* Add here the Interrupt Handlers for the used peripherals.                  */
/* For the available peripheral interrupt handler names,                      */
/* please refer to the startup file (startup_stm32l4xx.s).                    */
/******************************************************************************/

/**
  * @brief This function handles EXTI line0 interrupt.
  */
void EXTI0_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI0_IRQn 0 */

  /* USER CODE END EXTI0_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(JOY_CENTER_Pin);
  /* USER CODE BEGIN EXTI0_IRQn 1 */
	copyToSnapshot();
	HAL_GPIO_TogglePin(LD_R_GPIO_Port, LD_R_Pin);
  /* USER CODE END EXTI0_IRQn 1 */
}

/**
  * @brief This function handles EXTI line1 interrupt.
  */
void EXTI1_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI1_IRQn 0 */

  /* USER CODE END EXTI1_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(JOY_LEFT_Pin);
  /* USER CODE BEGIN EXTI1_IRQn 1 */

  /* USER CODE END EXTI1_IRQn 1 */
}

/**
  * @brief This function handles TIM1 update interrupt and TIM16 global interrupt.
  */
void TIM1_UP_TIM16_IRQHandler(void)
{
  /* USER CODE BEGIN TIM1_UP_TIM16_IRQn 0 */

  /* USER CODE END TIM1_UP_TIM16_IRQn 0 */
  HAL_TIM_IRQHandler(&htim16);
  /* USER CODE BEGIN TIM1_UP_TIM16_IRQn 1 */
	//copyToSnapshot();
  /* USER CODE END TIM1_UP_TIM16_IRQn 1 */
```

```
}

/**
  * @brief This function handles TIM3 global interrupt.
  */
void TIM3_IRQHandler(void)
{
  /* USER CODE BEGIN TIM3_IRQn 0 */

  /* USER CODE END TIM3_IRQn 0 */
  HAL_TIM_IRQHandler(&htim3);
  /* USER CODE BEGIN TIM3_IRQn 1 */
      addToBuffer(Count);
  /* USER CODE END TIM3_IRQn 1 */
}

/**
  * @brief This function handles TIM6 global interrupt, DAC channel1 and channel2 underrun error interrupts.
  */
void TIM6_DAC_IRQHandler(void)
{
  /* USER CODE BEGIN TIM6_DAC_IRQn 0 */

  /* USER CODE END TIM6_DAC_IRQn 0 */
  HAL_TIM_IRQHandler(&htim6);
  /* USER CODE BEGIN TIM6_DAC_IRQn 1 */
                                if (Count >= 4)
                                {
        removeFromBuffer(4);
    }
                                else if (Count > 0 && Count < 4)
                                {
        removeFromBuffer(Count);
    }
  /* USER CODE END TIM6_DAC_IRQn 1 */
}

/**
  * @brief This function handles LCD global interrupt.
  */
void LCD_IRQHandler(void)
{
  /* USER CODE BEGIN LCD_IRQn 0 */

  /* USER CODE END LCD_IRQn 0 */
  /* USER CODE BEGIN LCD_IRQn 1 */
```

```
 /* USER CODE END LCD_IRQn 1 */
}

/* USER CODE BEGIN 1 */



/* USER CODE END 1 */
```

Project 2:

Create a copy of the prior project and rename it. Follow the steps provided for copying a CubeMX project.

Open the project in CubeMX and disable the center joystick.

Re-generate code and rebuild the project.

Now – add code to the while loop that creates the snapshot of the circular buffer every 1 second. You can use HAL_Delay() to time out the one second.

Your code does not work. You have a potential data corruption problem if you get an interrupt during the snapshot copy. Fix the problem in a way that you cannot ever get data corruption. Figure out how to demo that your code works. To get credit for the demo, you must explain to the TA how you know the data is uncorrupted.

In your lab report you must explain in detail how you know that no data structures are corrupted due to concurrency violations.

No data structures will be corrupted because interrupts will be disabled during Snapshot's execution. They are then reenabled for the Delay Function.

CODE FOR PART 2:

```
while (1)
 {
		//weirdly enough, it worked regardless if we disabled interrupts
		__disable_irq();
		copyToSnapshot();
		__enable_irq();
		HAL_Delay(1000);
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
}
```